A Simulator for Concurrent Objects

Jeannette M. Wing and Chun Gong

19 July 1990
CMU-CS-90-150 -2

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

A *concurrent object* is a data structure shared by concurrent processes. This paper describes in detail a simulation package for simulating, testing, and analyzing implementations of concurrent objects. We use linearizability as our fundamental correctness condition, thereby exploiting the semantics of the object's type to enhance the degree of concurrency allowed. The simulator determines for a finite history of operations executed on a concurrent object whether the history is linearizable. The user can give the simulator a finite set of input test histories or let the simulator generate and test an unbounded number.

This research was provided in part by the National Science Foundation under grant CCR-8906483 under the special NSF/DARPA joint initiative on Parallel Computing Theory. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the National Science Foundation, the Defense Advanced Research Projects Agency, or the U.S. Government.

510.7806
[illegible]
1-[illegible]

Keywords: concurrent systems, concurrent objects, linearizability, simulation, testing

A Simulator for Concurrent Objects

Jeannette M. Wing and Chun Gong

July 19, 1990

1. Introduction

A *concurrent object* is a data structure shared by concurrent processes. The traditional approach to implementing a concurrent object is to use critical regions [7], letting only a single process access the object at a time. Critical regions unnecessarily limit the degree of concurrency possible when the type semantics of the object are ignored [11]. 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, non-faulty processes will also be unable to progress [9].

Recently, other approaches for implementing concurrent objects have been proposed [10, 11, 1, 13, 14, 15, 16, 2, 5]. In the design and implementation of a concurrent object, we are faced with two problems:

1. What is our notion of correctness for a system composed of concurrent objects?
2. Given some notion of correctness, how can we show a given implementation is correct?

Answering the first question is one of definition; the second, of method. While there is no general agreement on an answer to the first, 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 [11], generalizes correctness notions that had previously been defined for specific data structures like atomic registers and FIFO queues. It is an intuitively appealing notion of correctness, and also enjoys other properties like locality, which simplifies the proof method, that other notions of correctness do not.

With regard to the second question, we advocate using the complementary techniques of verification and testing. This paper in particular describes an environment for testing implementations of concurrent objects through simulation. Proving linearizability of a data object is a nontrivial task. In [15], the proof of a simple concurrent set consists of five propositions, one lemma and one theorem. The more definitive approach of verification is the subject of other papers [6, 15] including Herlihy and Wing's original POPL paper.

Our simulation package provides a means of finding bugs in implementations, as well as hinting at ways for improving them. We intend our package to serve the following purposes: (1) to detect an implementation is incorrect, (2) to give some additional assurance that an implementation is correct [1] (3) as a basis to do

[1]Testing can show only the presence, not absence of bugs [4].

practical performance analysis of implementations by producing useful statistics for comparing the relative efficiency of different implementations.

In this paper we describe in detail the user interface to the simulator package, the structure of its internal components, and most importantly, the underlying analysis algorithm used to test an implementation. Section 2 defines our terminology, in particular the definition of linearizability; we adopt the notations and definitions as defined in [12], which contains a lengthier discussion and motivating examples. Section 3 describes the structure of the simulation package in detail. Section 4 presents some examples and Section 5 shows part of the package's source code, in particular the algorithm for checking the correctness of a given history.

2. Concurrent Objects and Linearizability

2.1. Model of Computation

A concurrent system consists of 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. 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 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)*. Operations of different processes may be interleaved.

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.

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:

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

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. In this paper, we use operational specifications expressed as (sequential) program code. Herlihy and Wing use the axiomatic style of Larch [8].

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: $<_H \subseteq <_S$.

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.

3. Structure of the Package

3.1. The Simulator

The simulation package consists of several C functions stored in separate files. It uses C Threads [3] to create a user-specifiable number of threads to simulate a MIMD system. C Threads is a run-time library that provides a C language interface to a set of low-level, language-independent primitives for manipulating threads of control. We use these primitives to implement various atomic instructions, e.g., TEST_AND_SET, FETCH_AND_ADD, which in turn are used in the implementation of a concurrent object.

Below we first give a high-level description of our simulation package and then implementation details of each of the components.

3.1.1. User's View of the Package

Figure 1 shows the logical structure of the package, where we use ovals to represent data and rectangles for procedures. Solid lines show control flow; dotted lines show data flow.

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 of processes to run, 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. *Test* creates

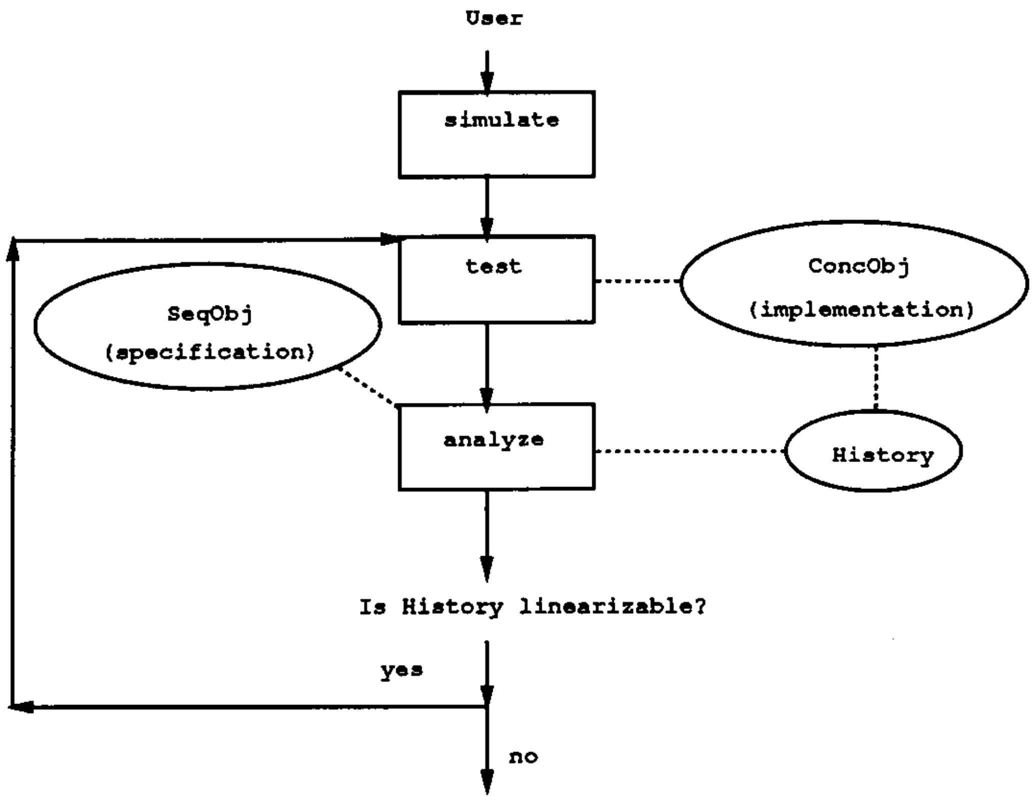

Figure 1: Simulator Package

N processes, and invokes concurrently on their behalf a finite number of operations on *ConcObj*. After all *N* processes terminate, the resulting finite concurrent history is stored in the event list *History* and the *analyze* function is called to determine the linearizability of the history. If an input file of (a finite number of) test cases is 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 even the user's request has not been met. A user can always abort the simulation by typing < *CTRL* >c. >> Below is a user-level specification of the simulation package. The simulator prompts for values for optional parameters, indicated in square brackets, if not given at the command line.

simulate [-c*ConcObj*] [-s*SeqObj*] [-i*input_file*] [-o*output_file*] [-d]
Simulate is the user's interface to the simulator. It accepts the following options:

-c*ConcObj* Simulate the implementation of the concurrent object whose executable is in the file *ConcObj.o*. If *ConcObj* is not a pathname, *simulate* will search in the current working directory for the file *ConcObj.o*. E.g., given that there exists a *queue.o* file to execute, "queue" and "/usr/cgong/sim/bin/queue" and "/sim/bin/queue" are all reasonable values for *ConcObj*. If it fails to find the file, *simulate* terminates, printing an error message.

-s*SeqObj* *SeqObj.o* is the sequential specification against which the simulator will test the linearizability of a history. As with *ConcObj*, *simulate* will search in the current working directory for *SeqObj.o* if a pathname is not given.

-i*input_file* The simulator tests each history in the file *input_file* of input test cases. Appendix A contains the grammar for the input file format. If no input file is specified, the simulator will generate a random set of test cases.

-o*output_file* For each history read from *input_file simulate* will write to *output_file* the history and whether or not it is linearizable. If no input file is specified, then if the simulation conditions are set for the simulator to run indefinitely, only the first non-linearizable history encountered is written to *output_file*; if they are set for the simulator to run for a fixed length of time, each history generated and tested within that time is written, along with its linearizability status, to *output_file*. An output file of histories can be used as an input file of test cases.

-d Use default simulation conditions as set in the working directory's *.init* file. If the user does not choose the -d option, the simulator will prompt the user to set various simulation conditions. These conditions include: the number of concurrent processes to execute, an average number of operations each process should invoke, how long (either in terms of seconds or number of histories generated) to run the simulator, and the names of input and output files to use. Reasonable default values are: 4 processes, 10 operations, 10 seconds (any negative number indicates "infinite"), and if an input file name is provided, "/dev/tty" for the output file name.

test(*input_file* : *file_name*, *output_file* : *file_name*)
Test is the main function of the simulation package. It creates the user-specified (or default) N number of concurrent threads. It tests one history at a time where the history is either taken from *input_file* (the user-specified input test file) or randomly generated. When all N processes terminate, *test* stores the history in a list of *events*, and then calls *analyze* (see below) with this history as a parameter. If *analyze* returns *false*, indicating that the input history is not linearizable, *test* writes the history into *output_file* (the user-specified output file), and terminates; otherwise it repeats the above steps.

bool analyze(*event* $*p$)
This function returns *true* if the history p (i.e., list of events) is linearizable; otherwise it returns *false*. This is the guts of the simulation package; we describe the algorithm used in Section 5.

3.1.2. Implementor's View of the Package

Users of the simulation package should skip this section in which we discuss the implementation in more detail.

simulate [-c*ConcObj*] [-s*SeqObj*] [-i*input_file*] [-o*output_file*] [-d]
This function provides an interactive interface to the simulation package. If not given in the command line, it prompts the user to choose a concurrent object, *ConcObj*, to simulate, and a sequential specification, *SeqObj*, to test against. *Simulate* can read its input from *input_file* or generate its own test cases; it writes its output into *output_file*. It establishes a simulation environment for the concurrent object based on values obtained from prompting the user or default values set in a *.init* file.

Given values for these parameters, *simulate* will first create a simulation environment by linking the concurrent data object *ConcObj.o* and the sequential specification *SeqObj.o* with *test*. It puts the resulting executive in *a.out*. Once the environment is created *simulate* forks a child process to execute *a.out* under the specified or default simulation conditions.

test(*input_file* : *file_name*, *output_file* : *file_name*)
This is the main function of the package. It requires the file name argument *output_file* into which the simulator can write simulation histories. It takes an optional file name argument *input_file* from which

the simulator reads histories to test; if not provided, *test* generates an infinite number of test histories. Termination of *test* also terminates *simulate*.

test assumes the existence of the following functions:

- *Init*() (written by user)
- *End*() (written by user)
- *process*(*outline* $*p$) (provided by package)
- *analyze*(*event* $*p$) (provided by package)

where *Init*() and *End*() are called to initialize and clean up *ConcObj*, *process* is a (sequential) process that invokes operations on *ConcObj*. *test* links these functions together and establishes various user-specifiable simulation conditions. It maintains a global data structure, a *history*, which is represented as a list of *events* with head and tail pointers *hhead* and *htail*. This list records events of the concurrent processes executing operations on *ConcObj*.

test operates as follows:

1. Initializes the concurrent data object the user wants to simulate by calling *Init*(). The user needs to define this function and *End*() when implementing a concurrent object.
2. Defines N process outlines, each of which includes the name of the process and the number of operations the process will invoke.
3. Creates N threads by executing *cthread_fork*(*process*, p_i) for $i = 1 \ldots N$, where p_i is a pointer to $outline_i$.
4. Waits until all N threads terminate. At this point, *simulator* has finished a simulation of one history with N processes. The history is saved in the list of events pointed to by *hhead*.
5. Calls the procedure *analyze(hhead)* to check if the history is linearizable. (a) If yes, clears the working *history*, cleans up the concurrent object by calling *End*(), and then goes back to step 2; (b) otherwise, it cleans up, returns *false*, and terminates.

process(*outline* $*p$)
The user defines a single (sequential) process using *process*(*outline* $*p$) where *outline* is a structure that contains a process name and the number of operations it should invoke. This function assumes the existence of a set of operations that can be invoked on a *ConcObj*. According to the *outline* given, *process* invokes a random sequence of operations on the concurrent object and writes the invocation and response events into the file *output_file* given as an argument to *test*. To prevent this file from being written by several processes simultaneously, it is protected by a mutex variable. A global data structure, also protected by a mutex variable, maintains a record of all processes' operation invocation and response events.

bool analyze(*event* $*p$)
Given a history with a head pointer p, this procedure checks if the history is linearizable. It uses the instances of the *operation* function (see below) for the *ConcObj* to determine if the operation *op* is allowed on the current value of the *ConcObj*.

bool operation(char op, char item, char result, bool b)
If *b* is *true*, this function simulates an operation *op(item)* with the argument *item* on the current value of the *SeqObj* and gets a response value *result'*. If *b* is *false*, this function undoes the operation *op(item)*. After simulating an operation, this function returns a *true* value if *result* = *result'*, which means that the operation completes successfully, i.e., is allowed given the current value of *SeqObj*; otherwise, it returns a *false*. It always returns *true* after undoing an operation.

3.2. Library of Concurrent Objects

Currently we have implemented and simulated the following data objects. A separate paper contains descriptions of the implementations and proofs of correctness [17].

- Bounded FIFO queues.
- Unbounded FIFO queues.
- Bounded priority queues.
- Unbounded priority queues.
- Semiqueue.
- Stuttering queue.
- Set.
- Multiset.
- Register.

As a reminder to implementors of other concurrent objects who wish to use our simulator, implementations of these objects need to include two functions, *Init*() and *End*(), which are called by *test* respectively before and after the test of a single history. The name of the implementation file must start with X- where X indicates the object's type. For example, for a queue, X should be *queue*; for a register X, should be *register*.

4. Examples

In this section we show how the simulator can be used to test both correct and incorrect implementations.

4.1. Simulating a Correct Implementation: FIFO Queue

To show a naive use of the simulator, we give below two sample scripts of a simulation for a bounded FIFO queue whose implementation appears in Appendix B. The first script shows how the user can set some simulation conditions. The second uses default values.

```
bin>simulate -cqueue-unbound -ooutput <CR>
Please provide a sequential specification. [filename]
-: ../lib/spec/queue
Just a moment...
How many processes would you like to run?
-: 4
On the average, how many operations do you want to execute per process?
-: 10
Would you like to simulate your object for a fixed length of time? [y/n]
-: y
For how long? [seconds]
-: 10
Start the simulation by typing <start>.
-: start
Starting simulation...
Time has run out!
bin>
```

At this point, the user could read the file *output* to see what histories within 10 seconds were generated and tested by the simulator (since no input file was specified) and whether or not they are linearizable.

In the second script below, the user manually aborts the simulator at the end by typing < *CTRL* > *c*. Since the implementation is correct and an input test file is not given, were the user not to abort the simulator, it would continue generating and testing concurrent histories forever. Our added comments are prefixed by *.

```
    % simulate -cqueue-bound -squeue -ooutput -d <CR>

    Just a moment...
    Starting simulation...

    /* This is simulation number 1 */
    Q   Deq()    P1      * P1,
    Q   Deq()    P2      * P2, and
    Q   Deq()    P3      * P3 cannot dequeue anything
    Q   Enq(b)   P4      * on an empty queue.
    Q   Ok()     P4      * P4 enqueues b and
    Q   Enq(s)   P4      * starts enqueueing s.
    Q   Ok(b)    P1      * P1 dequeues b.
    Q   Enq(y)   P1
    Q   Ok()     P1      * P1 enqueues y.
    Q   Deq()    P1
    Q   Ok(y)    P2      * P2 dequeues y.
    Q   Enq(w)   P2
    Q   Ok()     P4      * P4 completes enqueueing s.
    Q   Enq(j)   P4      * P4 starts enqueueing j.
    Q   Ok(s)    P3      * P3 dequeues s.
    Q   Enq(u)   P3      * and starts enqueueing u.
    Q   Ok()     P4      * P4 completes enqueueing j.
    Q   Enq(w)   P4
    Q   Ok(w)    P1      * P1 dequeues w
    Q   Ok()     P4      * enqueued by P4.
    <CTRL>c              * P3's enqueue of u is pending.
    %                   * Q's value is [j], [u, j], or [j, u].
```

Here we see that four processes, P1, P2, P3 and P4, execute concurrently on queue object Q. In the first half of the script above we see that P1 returns Q's first element b, enqueued by P4, and that P1 enqueues a y later dequeued by P2. More enqueues and dequeues occur until at the end of this history, Q's abstract value is the set of (sequential) queue values, [j], [u, j], [j, u], corresponding to whether or not P3's enqueue of u occurred or not; if so, since it is concurrent with the enqueue of j, it can appear as either the first or second element in Q.

4.2. Detecting an Implementation is Incorrect: Set

We can use our simulation package to demonstrate that it is possible to detect an implementation exhibits non-linearizable behavior. Lanin and Shasha [15] give a correct implementation of a concurrent set using a sliding array algorithm (see Appendix C). Our intentionally incorrect version given below is unlike the original one in that it does not use locks for the *insert* operation.

The set operations are implemented in terms of the following atomic instructions (which we implemented using the primitives provided by C Threads). The first two are operations on integer variables; the last three on arrays.

int READ(int *x)
ensures Atomically reads and returns the value of the integer pointed to by *x*.

int INC(int *x)
ensures Atomically increments by 1 the value of the integer pointed to by *x* and returns the new value.

char FETCH(char A[], int i)
ensures Atomically reads and returns the value at location *i* of the character array *A*.

bool REMOVE(char A[], int i, char x)
ensures Atomically checks whether the value at location *i* of array *A* is equal to *x*. If so, it sets the value at location *i* to *NULL* and returns *true*. Otherwise, it returns *false*.

bool ADD(char A[], int i, char x)
ensures Atomically checks whether the value at location *i* of array *A* is currently *NULL*. If so, it sets the value at location *i* to *x* and returns *true*. Otherwise, it returns *false*.

The Sliding Array Algorithm: We use an array data structure, containing either characters or a special NULL value. We use the variable *length* to hold the length of the currently used portion of the array.

```
#define S 200     /* Upper bound on size of the array */
char A[S];        /* Array of characters              */
int length;       /* Length counter                   */
```

Member(x) reads the value of *length*, and then scans the array from 1 up to the index equal to length, looking for *x*. If it sees *x*, it returns *true*; otherwise it returns *false*.

```
bool
member(char x) {
  int  mylength, i;
  bool found = false;
  char v;

  mylength = READ(&length);           /* get the current value of index */
  i = 0;
  while ((i < mylength) && !found) { /* scan the array */
    i++;
    v = FETCH(A, i);                  /* get the value at location i */
    found = (v == x);
  }
  if (found) {return(true);}          /* x is in the array */
  else {return(false);}
}
```

Delete(x) behaves just like *member(x)*, except that if it sees *x* in position *i*, it writes *NULL* in position *i*, and returns *true* ; otherwise it returns *false* .

```
bool
delete(char x) {
  int  mylength, i;
  bool found = false, removed = false;
  char v;

  mylength = READ(&length);
  i = 0;
  while (i < mylength && !found) {
    i++;
    v = FETCH(A, i);
    found = (v == x);
  }
  if (found) {                  /* x is in location i */
    removed = REMOVE(A, i, x); /* if x is still there, remove it */
  }
  if (removed) {return(true);}
  else {return(false);}         /* x was removed by other process */
}
```

Insert(x) also scans the array looking for *x* just like *member* and *delete*. If it sees *x*, it terminates returning *false*. Otherwise it increments the length counter, writes *x* in the returned position and returns *true*.

```
bool
insert(char x) {
  int  mylength, i;
  bool found = false, added = false;
  int holes[S],  nholes = 0;
  char v;

  mylength = READ(&length);
  i = 0;
  while (i < mylength && !found) {
    i++;
    v = FETCH(A, i);
    if (v == NULL) {                    /* find one hole in location i */
      nholes++;
      holes[nholes] = i;                /* remember the index */
    }
    found = (v == x);
  }
  if (!found) {                         /* x is not in the array */
    while (!added && nholes > 0) {     /* try to find a hole */
      added = ADD(A, holes[nholes], x); /* put x in the hole */
      nholes--;
    }
    while (!added) {                    /* no holes left */
      i = INC(&length);                 /* get another slot */
      added = ADD(A, i, x);             /* put x in the new slot */
    }
  }
  if (added) {return(true);}
  else {return(false);}
}
```

Note that *delete* leaves "holes" in the array. These holes might be re-used by subsequent *inserts*. An *insert* keeps track of these holes in the initial scan of the array by an array (*holes*) of indices. When *insert* is ready to write the element, it tries the holes in this array one by one (a concurrent *insert* might have filled a hole). If no holes remain, *insert* then performs an increment on the length counter.

When we simulated this set implementation, we got the following non-linearizable history:

```
/* This is the simulation number 84: */    * initially, s = {}
S   insert(y)    P1
S   delete(x)    P2
S   Ok(f)        P2       * cannot delete x from empty set
S   delete(y)    P2
S   Ok(f)        P2
S   delete(p)    P2
S   Ok(f)        P2
S   member(e)    P2
S   Ok(f)        P2       * e is not a member of the empty set
S   delete(d)    P2
S   Ok(f)        P2
S   member(w)    P2
S   Ok(f)        P2
```

```
S   delete(t)    P2
S   Ok(f)        P2
S   insert(s)    P2
S   Ok(t)        P2         * S = {s}
S   insert(v)    P3
S   Ok(t)        P3         * S = {s, v}
S   member(g)    P3
S   Ok(f)        P3
S   insert(l)    P3
S   Ok(t)        P3         * S = {s, v, l}
S   member(u)    P3
S   Ok(f)        P3
S   delete(f)    P3
S   Ok(f)        P3
S   insert(c)    P3
S   Ok(t)        P3         * S = {s, v, l, c}
S   delete(v)    P3
S   Ok(t)        P3         * S = {s, l, c}
S   insert(w)    P3
S   Ok(t)        P3         * S = {s, l, c, w}
S   member(t)    P4
S   Ok(f)        P4
S   insert(a)    P4
S   Ok(t)        P4         * S = {s, l, c, w, a}
S   delete(h)    P4
S   Ok(f)        P4
S   member(w)    P4
S   Ok(t)        P4         * w is a member of the set
S   insert(t)    P4
S   Ok(t)        P4         * S = {s, l, c, w, a, t}
S   insert(i)    P4
S   Ok(t)        P4         * S = {s, l, c, w, a, t, i}
S   delete(l)    P4
S   Ok(t)        P4         * S = {s, c, w, a, t, i}
S   delete(a)    P4
S   Ok(t)        P4         * S = {s, c, w, t, i}
S   insert(z)    P4
S   Ok(t)        P4         * S = {s, c, w, t, i, z}
S   Ok(t)        P1         * S = {s, c, w, t, i, z, y}
S   member(j)    P1
S   Ok(f)        P1
S   insert(e)    P1
S   delete(b)    P3
S   Ok(f)        P3
S   member(i)    P3
S   Ok(t)        P3
S   insert(b)    P3
S   Ok(t)        P3         * S = {s, c, w, t, i, z, y, b}
S   insert(e)    P3
S   Ok(t)        P3         * S = {s, c, w, t, i, z, y, b, e}
S   delete(s)    P4
S   Ok(t)        P4         * S = {c, w, t, i, z, y, b, e}
S   insert(p)    P4
S   Ok(t)        P4         * S = {c, w, t, i, z, y, b, e, p}
S   member(o)    P4
S   Ok(f)        P4
S   delete(n)    P4
S   Ok(f)        P4
S   insert(i)    P4
S   Ok(f)        P4
```

```
S   insert(x)   P4
S   Ok(t)       P1      * Wrong! p1 should not insert e into S.
S   insert(h)   P1
S   Ok(t)       P1
S   Ok(t)       P4
S   delete(q)   P4
S   Ok(f)       P4
/* This history is not linearizable! */
```

We see that process P1 successfully inserts an *e* into S after it was inserted already by P3. This situation can arise since two processes trying to insert the same element *x* at the same time, while scanning the array looking for *x*, may both find that there is no *x*. Both then proceed to insert *x* in the set (one after another). The original (correct) solution associates with each element a lock and requires *insert* of an element first acquire the associated lock, thereby allowing only one process at a time to insert a particular element in the set.

4.3. Detecting an Implementation is Incorrect: Unbounded FIFO Queue

The original implementation of an unbounded queue appeared in [11] and was written in CLU using dynamic arrays as the representation type for queues. We repeat below an incorrect implementation where we use C (dynamic) lists as the representation type [2]. Appendix B contains the correct version.

We store the elements of the queue in a list with head and tail pointers *head, tail*. We keep track of the number of the elements in *back*.

```
typedef struct elts {
        char 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;

reptype queue;
```

The queue operations make use of the following atomic instructions, which as for the set example we also implemented using mutex locks from the C threads package.

slot FETCH_AND_ADD(reptype Q, slot s)
ensures Atomically allocates a new, empty slot to the queue *Q*, increases
Q.back by 1, and returns the old value of *s*, which is a slot.

void STORE(slot s, char x)
ensures Stores *x* in the slot *s*.

[2]We also represent items as characters.

int READ(int *x)
ensures Returns the value of the integer pointed to by x.

char SWAP(slot s, char x)
ensures Stores x in the slot s and returns the old value of the item stored in s.

Enq increments the back pointer by one and stores an element in the empty position.

```
void Enq(char x) {
    slot current;

    current = FETCH-AND-ADD(queue, queue.tail); /* get an empty slot and */
                                                /* increase back  by 1   */
    STORE(current, x);                          /* store x in empty slot */
}
```

Deq fixes its search range and scans the list for a non-NULL element. If it does not find one, it changes its search range and looks again.

```
char Deq() {
  int  i, range;
  char ch;
  slot current;

  while (true) {
    current = queue.head;       /* start from list's first slot */
    i = 1;
    range = READ(&(queue.back)) - 1;  /* search up to back-1 slots */
    while (i <= range) {        /* scan list, looking for an non-NULL element */
      if ( i > 1)
        current = current->next;
      ch = SWAP(current, NULL); /* put a NULL value in ith slot */
      if (ch != NULL) {         /* if char returned is not a NULL value */
        return(ch);             /* return it */
      }
      i++;
      range = READ(&(queue.back)) - 1; /* modify current search range */
    }
  }
}
```

We define a partial order $<_r$ on the items in the list: $x <_r y$ iff the *STORE* operation for x precedes the FETCH_AND_ADD for y. We need to show that:

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 still preserves the first property, 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 $<_r$.)

Running this version of the unbounded FIFO queue through our simulator yielded the following non-linearizable 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 = NULL
Q   Ok()     P4          * Q = [y]
Q   Enq(t)   P4
Q   Enq(e)   P1
Q   Ok()     P4          * Q = [y, t]
Q   Enq(o)   P4
Q   Ok()     P1          * Q = [y, e, t] or Q = [y, t, e]
Q   Deq()    P1
Q   Ok(e)    P2          * Wrong! e cannot be the first element of Q.
Q   Enq(c)   P2
Q   Ok(y)    P1
Q   Ok()     P2
Q   Ok(c)    P3
Q   Ok()     P4

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

5. The Linearizability Analysis Algorithm

For a given data type *T*, we have two objects, *ConcObj* and *SeqObj*, corresponding to a concurrent implementation of an object of type *T* and a sequential implementation. We use *SeqObj* as our (operational) specification for checking whether a history exhibited by *ConcObj* is linearizable.

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 non-linearizable. During this procedure of rearranging the operations of a history, *analyze* might need to undo some operations on *SeqObj* if it finds that a tentative reordering of a subhistory of *H* is non-linearizable.

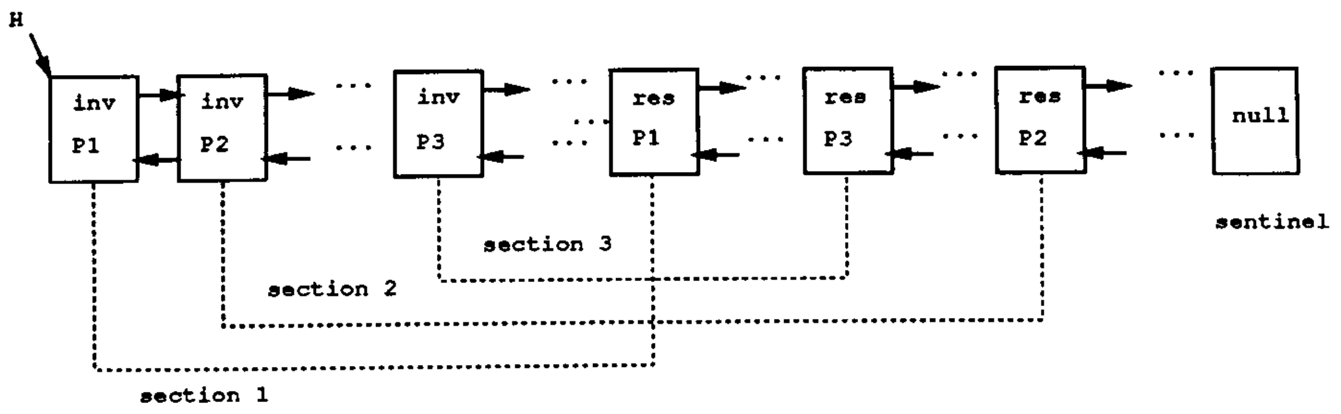

Figure 2: Snapshot of a History

5.1. The Algorithm

A history *H* is stored in a double 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 2 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 the events in between them.

Sections of a history *H* can be ordered by the positions of their first *events* in *H*. Figure 2 shows three sections in the history *H*.

Analyze calls the procedure *search* to search iteratively over events.

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

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

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

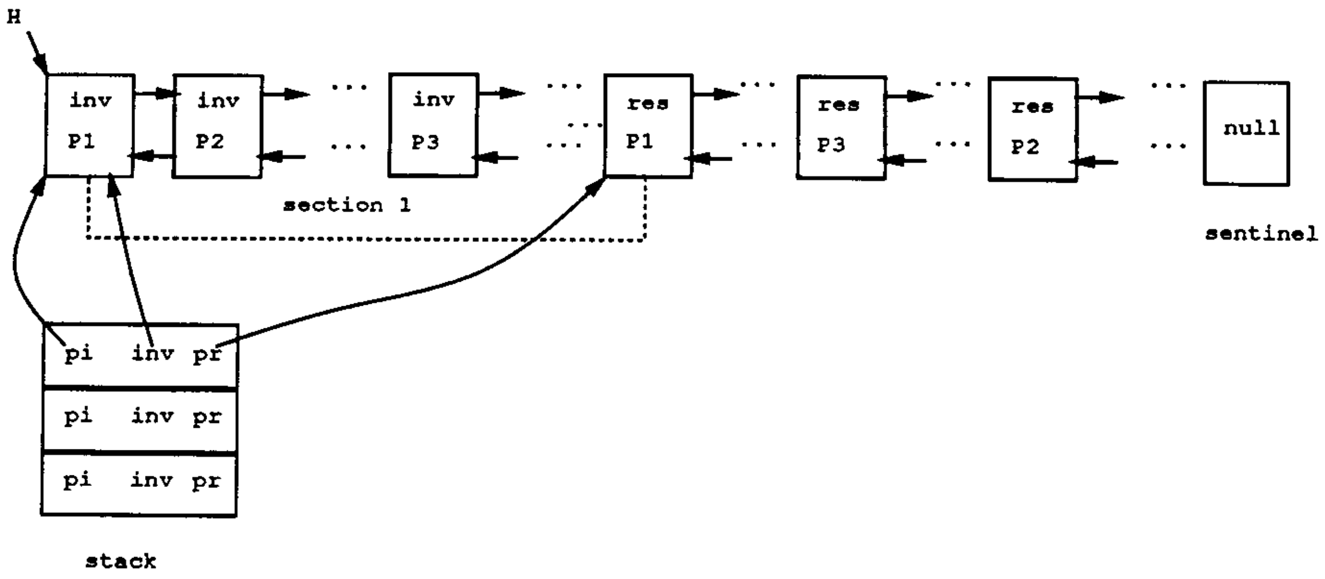

Figure 3: Top of the Stack Tracks Current Section and Operation

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 Figure 3).

```
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*, *res* to remember this operation.

We implemented the stack procedures, push, pop, top, and isempty, with their usual semantics (top is non-mutating), and also the following auxiliary procedures on event lists:

*void op_copy(event * inv, elt_stack * q)*
ensures Makes a local copy the operation whose invocation event is *inv* into *q*.

*bool lift(event * inv, event * resp)*

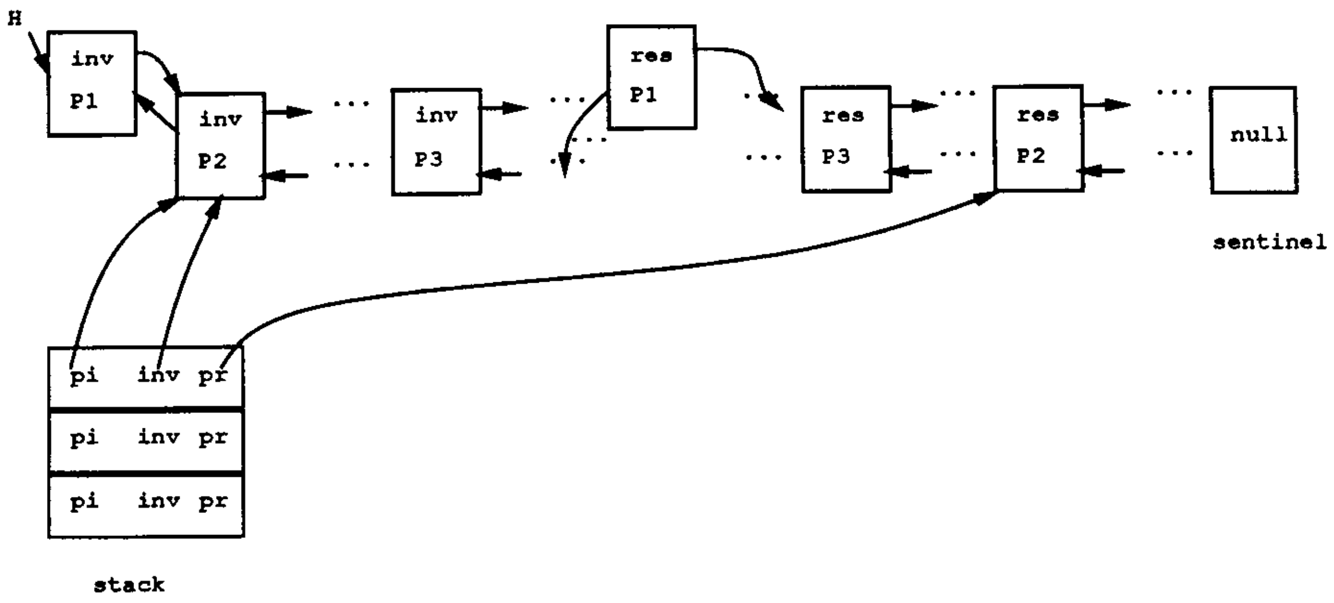

Figure 4: Lifting an Operation in a History

ensures This function "lifts" the invocation and response events *inv* and *resp* from the double linked list (recall that the history is represented as a double linked list of *events*). If the invocation event is the first event in the remaining history *lift* returns *true*, otherwise it returns *false*. To "lift" an event *e* from a double linked list means that *e* is marked as temporarily removed. Later, we might backtrack and need to return *e* to its original position in the history (see Figure 4).

*void unlift(event * inv, event * resp)*
ensures Puts the invocation and response events *inv* and *resp* back in their original positions in the history *H*.

*event * linearization(stacktype * s)*
ensures This function links the operations stored in the stack *s* from bottom to top, creating and returning a linearization of the history *H*.

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

More specifically, we sketch below the gist of the *search* procedure, referring to lines in the code given in Figure 6:

1. Initialize the stack.
2. Locate the current *section* of *H* by the *pi* and *pr* pointers of *current* (lines 3-5) if there is one (see Figure 3); otherwise return a pointer to the linearized history (line 33).

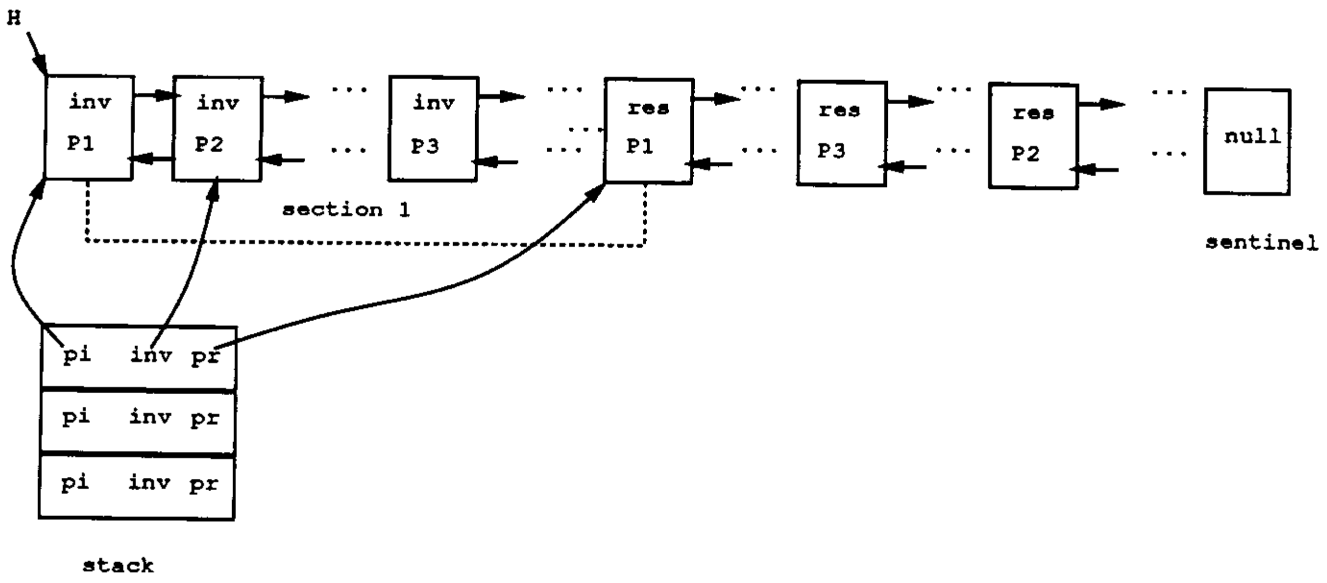

Figure 5: Select Another Operation

3. From the current section, select an operation and store it in *current* (line 6, 8).
4. Simulate this selected operation on the sequential implementation of the object by calling *operation* (line 9).
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 (lines 10-12 (see Figure 4) and go back to 2 (line 13);
   (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 by *current*'s *inv* pointer (line 16-18) 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 (line 20). Otherwise, (1) get the top element of the stack, which contains all information about the previous section and selected operation, by an auxiliary pointer and pop the stack (line 22-23); (2) undo the previous operation and put it back to the history (line 24-25) (see Figure 5); (3) set *current*'s pointers to the previous section and operation (line 26-29); and (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.

5.2. 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_l of H, we maintain the following invariants:

```
/* 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 */
              head = current.pi->next;          /* operation from H. */
           )      /* If it's the first operation of the history, then update head */
           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);             /* backtrack 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 */
  }
```

Figure 6: Source Code for Search Procedure

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_l of H and $<_{H_l} \subseteq <_H$.

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

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

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 3 and only under the condition that the remaining history is empty. At this point, we have processed the entire history successully, identifying a linearization of H operations stored from *stack*[1]. . . . . *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 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 $H - H_l$, 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.

6. 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. This research was provided in part by the National Science Foundation under grant CCR-8906483 under the special NSF/DARPA joint initiative on Parallel Computing Theory.

Appendix A. Grammar for Input File Format

```
<file>        ::=   <history>*
<history>     ::=   <history_1> blank_line
<history_1>   ::=   [<comment new_line>] <event_line>* [<comment>]
<event_line>  ::=   <event> new_line
<event>       ::=   <object_id> <white_space>+ <oper>(<arg_list>) <white_space>+ <process_id>
                  | <object_id> <white_space>+ <resp>(<res_list>) <white_space>+ <process_id>
<comment>     ::= /* ascii_text */
<object_id>   ::=   <id>
<process_id>  ::=   <id>
<oper>        ::=   <id>
<resp>        ::=   <id>
<arg_list>    ::=   <id,>*
<resp_list>   ::=   <id,>*
<id>          ::=   alphaNumeric+
<white_space>::=  blank | tab
```

Appendix B. The Correct Version of the Unbounded Queue

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

typedef slotpt  slot;

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

reptype queue;

mutex_t back_lock, store_lock;

void Enq(char x) {
    slot current;

    current = FETCH-AND-ADD(queue, queue.tail);
    STORE(current, x);
}

char Deq() {
  int  i, range;
  char ch;
  slot current;

  while (true) {
     current = queue.head;
     range = READ((&queue.back)) - 1;
     for (i = 1; i <= range; i++) {
         if ( i > 1) {
           current = current->next;
         }
         ch = SWAP(current, NULL);
         if (ch != NULL) {
            return(ch);
         }
     }
}
```

Appendix C. The Correct Version of Set

```
#define S 200
#define CHAR_A 97

char A[S];
int length;
mutex_t lock[26];

member(x)
     char x;
{
     int mylength, i;
     int found = 0;
     char v;

     get_length(&mylength);
     i = 0;
     while ((i < mylength) && !found) {
          i++;
          atomic_read(i, &v);
          found = (v == x);
     };
     if (found)
          return('t');
     else return('f');
}

delete(x)
     char x;
{
     int mylength, i, found = 0, removed = 0;
     char v;

     get_length(&mylength);
     i = 0;
     while (i < mylength && !found) {
          i++;
          atomic_read(i, &v);
          found = (v == x);
     };
     if (found)
          remove(i, x, &removed);
     if (removed)
        return('t');
     return('f');
}


insert(x)
     char x;
{
     int mylength, i, found = 0, added = 0;
     int holes[S];
     int nholes = 0;
     char v;

     mutex_lock(lock[x - CHAR_A]);
     get_length(&mylength);
```

```
        i = 0;
        while (i < mylength && !found) {
            i++;
            atomic_read(i, &v);
            if (v == NULL) {
                 nholes++;
                 holes[nholes] = i;
            };
            found = (v == x);
        };
        if (!found) {
             while (!added && nholes > 0) {
                  add(holes[nholes], x, &added);
                  nholes--;
             };
             while (!added) {
                 increment_length(&i);
                 add(i, x, &added);
             }
        };
        mutex_unlock(lock[x - CHAR_A]);
        if (added)
            return('t');
        return('f');
    }
```

References

[1] J.E. Burns and G.L. Peterson. Constructing multi-reader atomic values from non-atomic values. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 222–231, 1987.

[2] B. Chor, A. Israeli, , and M. Li. On procesor coordinationusing asynchronous hardware. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 86–97, 1987.

[3] Eric C. Cooper. C threads. Technical Report CMU-CS-88-154, School of Computer Science, Carnegie Mellon University, 1988.

[4] E.W. Dijkstra. *Notes on Structured Programming*, pages 1–82. Academic Press, 1972.

[5] C.S. Ellis. Concurrent search and insertion in 2-3 trees. *Acta Informatica*, 1980.

[6] Chun Gong and Jeannette Wing. A library of concurrent objects and their proofs of correctness. Technical Report CMU-CS-90-151, School of Computer Science, Carnegie Mellon University, 1990.

[7] F.B. Schneider G.R. Andrews. Concepts and notations for concurrent programming. *Computing Surveys*, 15(1):1–43, March 1983.

[8] J.V. Guttag, J.J. Horning, and J.M. Wing. The larch family of specification languages. *IEEE Software*, 2(5):24–36, September 1985.

[9] M.P. Herlihy. Impossibility and universality results for wait-free synchronization. In *Seventh ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, August 1988.

[10] M.P. Herlihy and J.M. Wing. Implementing queues without mutual exclusion. Some queue examples.

[11] M.P. Herlihy and J.M. Wing. Axioms for concurrent objects. In *Fourteenth ACM Symposium on Principles of Programming Languages*, pages 13–26, January 1987.

[12] M.P. Herlihy and J.M. Wing. Linearizability: A correctness condition for concurrent objects. *Transactions on Programming Languages and Systems*, 12(3), July 1990.

[13] L. Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, November 1977.

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

[15] V. Lanin and D. Shasha. Concurrent set manipulation without locking. In *Proceedings of the Seventh ACM Symposium on Principles of Database Systems*, pages 211–220, March 1988.

[16] J.M. Mellor-Crummey. Concurrent queues: Practical $fetch - and - \phi$ algorithms. Technical Report TR 229, Dept. of Computer Science, University of Rochester, November 1987.

[17] J.M. Wing and C. Gong. A library of concurrent objects and their proofs of correctness. Technical Report CMU-CS-90-151, CMU School of Computer Science, July 1990.