A Library of Concurrent Objects and Their Proofs of Correctness

Chun Gong and Jeannette M. Wing

July 20, 1990

1. Focus and Motivation of Paper

A concurrent object is a data structure shared by concurrent processes. It provides a set of primitive operations that are the only means for processes to manipulate the object. 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. Informally, we say an implementation of a concurrent object O is correct if and only if each concurrent history H accepted by O is "equivalent" in some sense to some legal sequential history, where (1) legality is defined in terms of the (sequential) type semantics of the object and (2) the "equivalent" sequential history preserves the real-time ordering of operations in H.

Linearizability, first coined in Herlihy and Wing's 1987 POPL paper [3], 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 a library of concurrent objects that we have designed, implemented, tested and verified. We focus here on the proofs of correctness of the implementations of these objects; a companion paper [8] describes our simulator, used for testing our implementations.

Proving linearizability of a data object is a nontrivial task. In [6], the proof of a simple concurrent set consists of five propositions, one lemma and one theorem. All our proofs are structured similarly by first giving a list of three to five lemmas and then a final theorem. Thus, in the spirit of reuse through program libraries, one of our goals is to supply users with a library of "verified" concurrent objects. The application writer is then freed from having to design, implement, and verify commonly-used abstractions. What we present here is just the beginning of a concurrent object library and we welcome additions from others.

In the remainder of this section we first describe our model of computation, thereby allowing us to define linearizability formally. We proceed to describe our specification language (Larch) used to define the type semantics of an object and then discuss the key property of our proofs.

In the remainder of this paper, we then present for each of the following data types, a specification, an implementation, and a proof of correctness (or cite other work that contains a proof):

- FIFO queue (unbounded and bounded versions).
- Priority queue (unbounded and bounded versions).
- Semiqueue.
- Stuttering queue.
- Set.
- Multiset.
- Register.

1.1. 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:

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

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 the axiomatic style of Larch [2].

1.2. 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.

1.3. Specification Language

We use the Larch Specification Language [2] 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 Bag trait of Figure 1, *emp* and *ins*(*emp*, 5) denote two different bag 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 Bag, one could prove that *del*(*ins*(*ins*(*ins*(*emp*, 4), 3), 3), 3) = *ins*(*ins*(*emp*, 4), 3). The **generated by** clause of Bag asserts that *emp* and *ins* are sufficient operators to generate all values of bags. Formally, it introduces an inductive rule of inference that allows one to prove properties of all terms of sort B.

Larch provides two ways of reusing traits: a trait T can **include** or **assume** another trait T_1. If T_1 is included, then T extends the theory denoted by T_1 by adding more operators and equations explicitly in T. For example, FifoQ of Figure 2 includes Bag and adds two operators, first and rest, and two equations to those of Bag. From FifoQ, one could show that *first*(*ins*(*ins*(*ins*(*emp*, 4), 3), 3)) = 4. If T_1 is assumed, then T may use T_1's operators with their meaning as given in T_1; a further use of T must discharge the assumption of T_1's theory. For example, a trait for priority queues (q.v., Section 3) might assume the existence of a total ordering on the items inserted in the queue. With either kind of reuse, a **with** clause allows renaming of operator and sort identifiers.

We use Larch *interfaces* to describe an object's set of operations. For example, interfaces for the *Enq* and *Deq* operations for Bags and FIFO queues are shown in Figures 1 and 2, respectively. An interface's header is of the form *x :: op(args*)/term(res*)* where x is the object's identifier, and *op(args*)/term(res*)* is the (matching pair of) invocation and response events of the operation being specified. The object's identifier is an implicit argument and return formal (parameter) of each operation. A **requires** clause states

Bag: **trait**
  introduces
    emp: → B
    ins: B, E → B
    del: B, E → B
    isEmp: B → Bool
    isIn: B, E → Bool
  asserts
    B **generated by** [emp, ins]
    for all [b: B, e, e1: E]
      del(emp, e) = emp
      del(ins(b, e), e1) = **if** e = e1 **then** b **else** ins(del(b, e1), e)
      isEmp(emp) = true
      isEmp(ins(b, e)) = false
      isIn(emp, e) = false
      isIn(ins(b, e), e1) = (e = e1) ∨ isIn(b, e1)

b:: Enq(e)/Ok()
  ensures b' = ins(b, e)

b:: Deq()/Ok(e)
  requires ¬ isEmp(b)
  ensures isIn(b, e) ∧ b' = del(b, e)

Figure 1: Bag Trait and Interfaces

FifoQ: **trait**
  includes Bag **with** [Q **for** B]
  introduces
    first: Q → E
    rest: Q → Q
  asserts for all [q: Q, e: E]
    first(ins(q, e)) = **if** isEmp(q) **then** e **else** first(q)
    rest(ins(q, e)) = **if** isEmp(q) **then** emp **else** ins(rest(q), e)

q:: Enq(e)/Ok()
  ensures q' = ins(q, e)

q:: Deq()/Ok(e)
  requires ¬ isEmp(q)
  ensures q' = rest(q) ∧ e = first(q)

Figure 2: FIFO Queue Trait and Interfaces

the precondition that must hold when an operation is invoked. An omitted **requires** clause is interpreted as equivalent to "**requires** true." 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 return formal or a primed argument formal, e.g., q', stands for the value of the object at the end of the operation. For an object x, the absence of the assertion $x' = x$ in the postcondition states that the object's value may change. 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 meaning of *ins* and = in the postcondition for the Bag is given by the Bag trait, but that for the FIFO queue, by the FifoQ trait. Note that the postcondition for the Deq operation for the Bag is weaker than that for the FIFO queue.

1.4. Informal Proof Method

Consider the FIFO queue which has two operations, *Enq* and *Deq*, as specified in Figure 2. Its specification makes sense only when there is a total order relation between the items in queue, i.e., 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 total order relation can we define that will still give meaning to $first$? The answer is that the implementor of a concurrent object needs to define a total order relation; a proof of correctness amounts to showing that the implementation maintains this total order.

As mentioned, 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. Based on our work described in [3], we found that the most difficult part of the verification task is in defining an ordering relation in terms of the representation operations used to implement the (abstract) operations of a concurrent object, and then to argue, informally or formally, that the implementation maintains it. This ordering information is the key insight that a (human) prover must provide for each proof of correctness.

2. FIFO queues

2.1. Specification

See Figure 2.

2.2. Implementations

We provide two implementations, an unbounded version and a bounded version.

2.2.1. Unbounded Version

Figure 3 contains the source code for the queue representation and *Enq* and *Deq* operations. We store the elements of the queue in a list with head and tail pointers, *head* and *tail*, and keep track of the number of the elements in *back* (initialized to 0).

The implementation of each object's operation is given in terms of a set of atomic *instructions*, i.e., representation operations that are indivisible. Our implementation makes use of the following atomic instructions [1]:

slot FETCH_AND_ADD(reptype Q, slot s)
ensures Allocates a new, empty slot (with *item* = *NULL*) to *Q*, increases *Q.back* by 1, and returns the old value of *s*, which is a slot.

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

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

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

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. *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.

2.2.2. 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) is in H, we use $enq(x)$ and $eok(x)$ to refer to its invocation and response event; if the operation Deq() is in H and the value returned by its response event is x, we use $deq(x)$ and $dok(x)$ to refer to its invocation and response event. Here we assume that no two items are equal.
- $enqueue(H) = \{x \mid \text{Both } enq(x) \text{ and } eok(x) \text{ are in } H\}$. Note, this is a partially ordered set with the relation $<_r$ (see below) and we define the concept of *chain* as usual.

[1] We implemented these instructions using mutex locks from the C threads package [1]. The informal specifications of the atomic instructions follow the style of Larch/C interfaces.

```
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;

void Enq(elt x)               /* enqueue an element x */
{
     slot current;

     current = FETCH_AND_ADD(&queue, queue.tail); /* get an empty slot */
     STORE(current, x);                           /* store x in slot */
}

elt  Deq()                    /* Deq will return the first element */
{
     int i, range;
     elt ch;
     slot current;

     while (true) {    /* keep trying till non-NULL value is found */
          current = queue.head;            /* starting from the first slot */
          range = READ(&(queue.back)) - 1;/* search up to back-1 slots */
          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 */
              }
          }
     }
}
```

Figure 3: Code for FIFO Queue (Unbounded Version)

- *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 array:

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

We will be interested in any total order relation consistant 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*. If we replace *range* in line 5 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. Consider the following scenario:

    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 $<_r$.)

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*.

Lemma 1.2 If H is a history accepted by the unbounded FIFO queue, then for all i, $1 \leq i \leq |H|$, *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) ($i < j$), 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 $label(enq(x)) < label(enq(y)) \iff m < n$.

Proof: By the semantics of FETCH_AND_ADD.

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, x_b stored in the lth and nth slots, $1 < l < n < m$, 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, 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. Now we assume that for any complete H with l operations this 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 $1 \leq j < 2(l + 1)$. 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 $H_j = deq()$, $1 \leq j < 2(l + 1)$. The same arguments as above applies with Lemma 1.3 replaced by Lemma 1.5.

2.2.3. Bounded Version

The code for our second implementation of a FIFO queue is given in Figure 4. We use a bounded array to store the elements of the queue and the *back* counter to keep track of the number of elements in the queue. We use modular arithmetic to "fold" an unbounded array 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". 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)

ensures If $e1.tag$ matches *gen*, then $e2$ is set to the value of $e1$; otherwise $e1$ is unchanged. The old value of $e1$ is returned.

void FETCH_AND_MAX(location l, int i)
ensures This operation replaces a memory location l with the maximum of i and l's current value.

Figure 4 contains the source code for the bounded version of the FIFO queue. From the code, we see that 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). *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.

2.2.4. Proof of Correctness for the Bounded Version

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

$$x <_r y \text{ iff FETCH\_AND\_MAX for } x \text{ precedes the READ for } y$$

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 notations:

- 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.

[2] C arrays start indexing from 0.

```
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;

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

     e.item = x;                    /* set the new element's item to x */
     i = READ(&(queue.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(&(queue.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(&(queue.back), i); /* reset the value of back */
}


elt Deq()
{
     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(&(queue.back)) - 1; /* search up to back-1 slots */
          for (i = 0; i <= range; i++) {
              olde = EXCHANGE(&(queue.elts[i % SIZE]), i / SIZE, &e);
                     /* 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 */
     }
}
```

Figure 4: Code for FIFO Queue (Bounded Version)

```
PQueue: trait
   assumes TotalOrder with [E for T] % > denotes the total order relation
   includes Bag with [PQ for B]
   introduces
      best: PQ → E
   asserts for all [q: PQ, e: E]
      best(ins(q, e)) = if isEmp(q)
         then e
         else if e > best(q) then e else best(q)

q:: Enq(e)/Ok()
   ensures q' = ins(q, e)

q:: Deq()/Ok(e)
   requires ¬ isEmp(q)
   ensures e = best(q) ∧ q' = del(q, e)
```

Figure 5: Priority Queue Trait and Interfaces

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, then H is linearizable under the constraint of FIFO queue semantics.

Proof: Similer to the proof of Theorem 1 using Lemmas 1.1 and 2.1-2.3.

3. Priority Queue

3.1. Specification

Figure 5 contains the specification of a priority queue.

3.2. Implementations

Again, we provide two implementations, one unbounded and one bounded.

3.2.1. Unbounded Version

This implementation (Figure 6) is almost the same as the one for the unbounded version of FIFO queue. The only difference is that we assume a relation $<$ (the "priority" ordering) on queue elements. *Deq* must return the maximal ("best") element with respect to $<$ in the queue. We also need another atomic instruction:

elt FETCH_KEY(slot s)
ensures Returns the current item in slot s.

Enq is the same as for the unbounded FIFO queue. *Deq* scans the list, keeping track of the slot of the highest priority element seen so far. If the dequeuer, D, has found a non-NULL element, it returns to that slot and attempts to remove the element (by swapping). If another dequeuer has already removed that element, D tries again.

3.2.2. Proof of Correctness for the Unbounded Version

Similar to that for the unbounded FIFO queue.

3.2.3. Bounded Version

The basic idea is the same one as that for the bounded FIFO queue. Figure 7 contains the code.

3.2.4. Proof of Correctness for the Bounded Version

Similar to that for the bounded FIFO queue.

4. Semiqueues

4.1. Specification

A $Semiqueue_k$ object (Figure 8) 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. It is straightforward to show that if k is one, the object is a FIFO queue (Figure 2) and if k is n, the maximum number of items allowed in the queue, the object is a bag (Figure 1). The motivation for providing this "weaker" queue data type [4] is to give better response time to dequeueing processes, as well as to let more of them proceed concurrently.

4.2. Implementation

Our implementation of a $Semiqueue_k$ (Figure 9) uses a list to store the queue elements and keeps the number of slots in *back*. We use the following atomic instruction:

void ADD(int *p)
ensures:Atomically increases by one the value of the integer pointed to by p.

The code for *Enq* is as for the (unbounded) FIFO and priority queues. 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

```
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;

void Enq(elt x)
{
     slot current;

     current = FETCH_AND_ADD(&queue, queue.tail);
     STORE(current, x);
}

elt Deq()
{
     elt best, ch;
     int i, range;
     slot current, myslot;

     while (true) {            /* keep trying until success */
          myslot = NULL;
          best = NULL;          /* set best to the lowest priority value */
          current = queue.head; /* search from the first slot */
          range = READ(&(queue.back)) - 1; /* up to back-1 slots */
          for (i = 1; i <= range; i++) {
              ch = FETCH_KEY(current);
              if (ch > best) {           /* finds a element with higher */
                                         /* priority in current slot */
                  best = ch;             /* reset best to that element */
                  myslot = current;      /* set myslot to current */
              }
              current = current->next;   /* continues to compare */
          }
          if (best != NULL) {            /* swap out the element */
            ch = SWAP(myslot, NULL);     /* with the highest priority */
            if (ch != NULL) {            /* if success */
                return(ch);              /* return the element */
            }
          }
     }
}
```

Figure 6: Code for Priority Queue (Unbounded Version)

```
typedef struct {
        elt item;
        int tag;
        } entry;

typeder struct rep {
       entry elts[SIZE];
       int  back;
       } reptype;

reptype queue;

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

     e.item = x;
     i = READ(&(queue.back)) + 1;
     while (true) {
          e.tag = i / SIZE;
          olde = EXCHANGE(&(queue.elts[i % SIZE]), -1, &e);
          if (olde->tag == -1) {
              break;
          }
          ++i;
     }
     FETCH_AND_MAX(&(queue.back), i);
}

elt Deq()
{
     entry e, *olde, best;
     int i, range, myslot;

     e.tag = -1;
     e.item = NULL;
     while (true) {
          myslot = -1;
          best.item = NULL;
          best.tag = -1;
          range = READ(&(queue.back)) + 1;
          for (i = 0; i < range; i++) {
              olde = FETCH_ENTRY(i);
              if ((olde->tag == i / SIZE) && (olde->item > best.item)) {
                 best.item = olde->item;
                 best.tag = olde->tag;
                 myslot = i % SIZE;
              }
          }
          if (best.item != NULL) {
              olde = EXCHANGE(&(queue.elts[myslot]), best.tag, &e);
              if (olde->item != NULL) {
                 return(olde->item);
              }
          }
     }
}
```

Figure 7: Code for Priority Queue (Bounded Version)

```
SemiQ: trait
  includes FifoQ, Set with [SetE for C]
  introduces
    prefix: Q, Int → SetE
  asserts for all [q: Q, i: Int]
    prefix(q, i) = if (i = 0 ∨ isEmp(q))
      then {}
      else prefix(rest(q), i-1) ∪ {first(q)}

q:: Enq(e)/Ok()
  ensures q' = ins(q, e)

q:: Deq()/Ok(e)
  requires ¬ isEmp(q)
  ensures q' = del(q, e) ∧ e ∈ prefix(q, k)
```

Figure 8: $Semiqueue_k$

search, *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 13-14), 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 $K - differ$.

4.3. Proof of Correctness for the Semiqueue

The total order relation 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 $S = \{x \mid x \text{ is stored in slot between the first slot and the } m\text{th slot}\}$ contains no chain with length $\geq K$.

```
#define K ...

typedef struct elts {
        elt item;
        struct elts *next;
    } *slotpt;

typedef slotpt  slot;

typedef struct {
   slot head, tail;
   int back;
} reptype;

reptype queue;
static int num_deqd = 0;

void Enq(elt x)
{
     slot current;

     current = FETCH_AND_ADD(&queue, queue.tail);
     STORE(current, x);
}

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

    while (true) {
        current = queue.head;
        range = READ(&(queue.back)); /* range initially is back */
        i = 0;                       /* starting search from first slot */
        while ((i < range) && (i < queue.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. */
            if (differ < K) {
                range += K - differ; /* ok to incr range since there were */
                                    /*  fewer than K items from 1 to range */
            }
        }
    }
}
```

Figure 9: Code for Semiqueue

Proof: Observe that the value of *range* can be changed at only two places, at beginning of the loop (because of some concurrent enqueueing process) and at the end of the loop (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, 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 $queue[l]$, 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 even no chain with length ≥ 2 before $queue[m]$.

Lemma 3.3 If H is a history accepted by 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 $label(eok(x_l)) < label(enq(x)) < j,\ (1 \leq l \leq k)$ 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 $\geq K$ between slot 1 and $m-1$, which contradicts Lemma 3.2.

Theorem 3: If H is a complete history accepted by semiqueue, then H is linearizable.

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

5. Stuttering Queues

5.1. Specification

A *Stuttering$_j$_Queue* object (Figure 10) is like a FIFO queue except that the first item in the queue may be returned as many as j times.

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

q:: Enq(e)/Ok()
  ensures q'.items = ins(q.items, e)

q:: Deq()/Ok(e)
  requires ¬ isEmp(q.items)
  ensures
    q.count < j ⇒ [e = first(q.items) ∧
    [[q'.count = q.count + 1 ∧ q'.items = q.items] ∨
    [q'.count = 0 ∧ q'.items = rest(q.items)]]]
```

Figure 10: *Stuttering$_j$_Queue*

5.2. Implementation

Figure 11 gives an implementation of a *Stuttering$_j$_Queue* queue. We do not use the atomic instructions SWAP and ADD, but instead we use FETCH_AND_ADD, STORE, and SUB:

void SUB(int *p)
ensures Decreases by one the value of the integer pointed to by p.

As with the implementation of 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.

5.3. Proof of Correctness for the Stuttering Queue

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

Lemma 4.1 No item can be dequeued more than *J* times in the *Stuttering$_j$_Queue* 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 *m*th 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.

```
#define J ...

typedef struct elts {
            char item;
            int hold;     /* used to remember the concurrent dequeuers */
            struct elts *next;
        } *slotpt;

typedef slotpt slot;

typedef struct {
            slot head, tail;
            int back;
        } reptype;

reptype queue;

void Enq(elt x)
{
     slot current;

     current = FETCH_AND_ADD(&queue, queue.tail);
     STORE(current, x);
}

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

    while (true) {
        current = queue.head;
        range = READ(&queue.back);
        for (i = 1; i <= range; i++) {
        num = READ_AND_ADD(&(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 deq it again */
            current = current->next; /* try next slot */
        }
    }
}
```

Figure 11: Code for Stuttering Queue

```
Set: trait
  includes Bag with [S for B]
  asserts
    for all s: S, e, e1: E]
      del(emp, e) = emp
      del(ins(s, e), e1) = if e = e1 then del(s,e) else ins(del(s, e1), e)

s: Member(e):/Ok(b)
  ensures b = isIn(s,e)

s:: Delete(e)/Ok(b)
  ensures s' = del(s,e) ∧ b=isIn(s,e)

s:: Insert(e)/Ok()
  ensures s' = ins(s,e)
```

Figure 12: Set Trait and Interfaces

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 still stored in some slot between the first and mth slots and there are less 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 controdiction.

Lemma 4.3 If H is a history accepted by the *Stuttering*$_j$*_Queue* 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 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*$_j$*_Queue* Queue, 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.

6. Set

6.1. Specification

Figure 12 contains the specification for a set.

6.2. Implementation

The original design appeared in [6]. Our implementation is given in Figures 13 and 14. The set operations are implemented in terms of the following atomic instructions. The first two are operations on integer variables; the last three on arrays.

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

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

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

bool REMOVE(char A[], int i, char x)
ensures 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 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. *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*. *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* . *Insert(x)* first get a lock for the item x and then 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, releases the lock and returns *true*. If other process has locked the same item, this process will blocks.

Note that *Delete* leaves "holes" in the array. These holes might be re-used by subsequent *insert*s. 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.

6.3. Proof of Correctness for the Set

See [6].

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

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

     mylength = READ(&length);
     i = 0;
     while ((i < mylength) && !found) {
       i++;
       v = FETCH(A, i);
       found = (v == x);
     }
     if (found) {
        return(true);
     }
     else {
        return(false);
     }
}


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

     mylength = READ(&length);
     i = 0;
     while (i < mylength && !found) {
       i++;
       v = FETCH(A, i);
       found = (v == x);
     }
     if (found) {
       removed = REMOVE(A, i, x);
     }
     if (removed) {
       return(true);
     }
     else {
       return(false);
     }
}
```

Figure 13: Code for Set (Part 1)

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

     mutex_lock(lock[x - CHAR_A]);
     mylength = READ(&length);
        i = 0;
        while (i < mylength && !found) {
            i++;
            v = FETCH(A, i);
            if (v == NULL) {
                 nholes++;
                 holes[nholes] = i;
            }
            found = (v == x);
        }
        if (!found) {
             while (!added && nholes > 0) {
                  added = ADD(A, holes[nholes], x);
                  nholes--;
             }
             while (!added) {
                 i = INC(&length);
                 added = ADD(A, i, x);
             }
        }
        mutex_unlock(lock[x - CHAR_A]);
        if (added) {
            return(true);
        }
        else {
            return(false);
        }
}
```

Figure 14: Code for Set (Part 2)

MultiSet: **trait**
  includes Bag

s: Member(e):/Ok(b)
  ensures b = isIn(s,e)

s:: Delete(e)/Ok(b)
  ensures $\neg$isIn(s',e) $\wedge$ b=isIn(s,e)

s:: Insert(e)/Ok()
  ensures s' = ins(s,e)

Figure 15: Multiple Set Trait and Interfaces

7. Multiple Set

7.1. Specification

Figure 15 gives the specification for a multiple set. Multiple sets are different from bags in that an element may occur multiple times in a multiple set but when it is deleted, all its occurrences are removed (whereas for a bag, only one instance is removed).

7.2. Implementation

The original design appeared in [6]. Ours is given in Figures 16 and 17. We need two more atomic instructions:

element FETCH_KEY(entry A[], i)
ensures Returns the element stored in the location i of array A.

int FETCH_GEN(entry A[], i)
ensures Returns the generation number in the location i of array A.

For the details of the algorithm, please see [6].

7.3. Proof of Correctness for the Multiple Set

See [6].

```
#define S 200

typedef struct {
        element item;
        int gen;
        } entry;

entry A[S];
int length;

bool member(element x)
{
     int mylength, i;
     element v;

     i = -1;
     mylength = READ(&length);
     while (i < mylength) {
        while (i < mylength) {
            i++;
            v = FETCH_KEY(A, i);
            if (v == x) {
               return(true);
            }
        }
        mylength = READ(&length);
     }
     return(false);
}

bool delete(element x)
{
     int mylength, i, g;
     int todo[S], gen[S], ntodo = -1;
     bool removed = false;
     element v;

     mylength = READ(&length);
     for (i = 0; i <= mylength; i++) {
        v = FETCH_KEY(A, i);
        g = FETCH_GEN(A, i);
        if (v == x) {
            ntodo++;
            todo[ntodo] = i;
            gen[ntodo] = g;
        }
     }
     for (i = 0; i <= ntodo; i++) {
         removed = removed || REMOVE(A, todo[i], gen[i]);
     }
     return(removed);
     }
```

Figure 16: Code for Multiple Set (Part 1)

```
bool insert(element x)
{
     int mylength, i;
     bool added = false;
     int holes[S], nholes = -1;
     element v;

     mylength = READ(&length);
     for (i = 0; i <= mylength; i++) {
        v = FETCH_KEY(A, i);
        if (v == NULL) {
            nholes++;
            holes[nholes] = i;
        }
     }
     while (!added && nholes > -1) {
        added = ADD(A, holes[nholes], x);
        nholes--;
     }
     while (!added) {
        i = INC(&length);
        added = ADD(A, i, x);
     }
     return(added);
}
```

Figure 17: Code for Multiple Set (Part 2)

8. Register

8.1. Specification

Figure 18 contains the specification for a register.

8.2. Implementation

The original design appeared in [5]. Our implementation is given in Figures 19 and 20.

8.3. Proof of Correctness for the Register

See [5].

Reg: **trait**
  includes Integer
  introduces
    new: $\rightarrow$ R
    fetch: R $\rightarrow$ Int
    store: R, Int $\rightarrow$ R
    dontcare: $\rightarrow$ V
  asserts
  R **generated by** [new, store]
    for all r: R, i: Int]
      fetch(new) = dontcare
      fetch(store(r,i)) = i

r:: Read()/Ok(v)
  ensures r' = r $\wedge$ v = fetch(r)

r:: Write(v)/Ok()
  ensures r' = store(r,v)

Figure 18: Register Trait and Interfaces

References

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

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

[3] 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.

[4] M.P. Herlihy and J.M. Wing. Specifying graceful degradation. *IEEE Transactions on Parallel and Distributed Computing*, June 1990. to appear.

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

[6] 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.

[7] Lehman and W.E. Weihl. Technical Report xx, MIT Lab. for Computer Science, 1990.

[8] J.M. Wing and C. Gong. A simulator for concurrent objects. Technical Report CMU-CS-90-150, CMU School of Computer Science, July 1990.

```
#define VL 10
#define RL 7

static int reg[RL];
static int v1[VL], v2[VL];

int exp(int i)
{
    int k, j;

    k = 1;
    for (j = 1; j <= i; j++)
        k *= 2;
    return(k);
}


bool compare(int j)
{
     int i, k;

     k = 0;
     for (i = 0; i < VL; i++)
        k = k + v1[i] * exp(i);
     return(k == j);
}

int Read()
{
    int i, tmp, value, exp();
    bool compare();

    do {
        tmp = 0;
        for (i = VL - 1; i >= 0; i--)
            tmp = tmp + v2[i] * exp(i);
        value = 0;
        for (i = 0; i < RL; i++)
            value = value + reg[i] * exp(i);
    } while (!compare(tmp));
    return(value);
}
```

Figure 19: Code for Register (Part 1)

```
void Write(int c)
{
     int i, vt[VL], carry;

     carry = 1;
     for (i = 0; i < VL; i++) {
        vt[i] = v1[i] + carry;
        if (vt[i] > 1) {
            vt[i] = 0;
            carry = 1;
        }
        else carry = 0;
     }
     for (i = VL - 1; i >= 0; i--)
         v1[i] = vt[i];
     i = 0;
     do {
        vt[i] = c % 2;
        c = c / 2;
        i++;
     } while (i < RL);
     for (i = 0; i < RL; i++)
        reg[i] = vt[i];
     for (i = 0; i < VL; i++)
        v2[i] = v1[i];
}
```

Figure 20: Code for Register (Part 2)