The Elusive Atomic Register

AMBUJ K. SINGH

The University of California at Santa Barbara, Santa Barbara, California

JAMES H. ANDERSON

The University of North Carolina at Chapel Hill, Chapel Hill, North Carolina

AND

MOHAMED G. GOUDA

The University of Texas at Austin, Austin, Texas

Abstract. We present a construction of a single-writer, multiple-reader atomic register from single-writer, single-reader atomic registers. The complexity of our construction is asymptotically optimal; $O(M^2 + MN)$ shared single-writer, single-reader safe bits are required to construct a single-writer, M-reader, N-bit atomic register.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features—*concurrent programming structures*; D.4.1 [**Operating Systems**]: Process Management—*concurrency, mutual exclusion, synchronization*; D.4.5 [**Operating Systems**]: Reliability—*fault-tolerance*; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*parallelism and concurrency*

General Terms: Algorithms, Reliability

Additional Key Words and Phrases: Atomic register, linearizability, wait-free synchronization

1. *Introduction*

The currently accepted theory of concurrent computing is deeply rooted in the concept of atomic registers. An *atomic register* is a data object that is read or written by one or more processes according to the following assumption: If several read or write operations of the register are enabled simultaneously in different processes, then these operations are executed in some sequence, one

The work of A. K. Singh was supported in part by National Science Foundation (NSF) grant ECS 83-04734 and Office of Naval Research (ONR) contract N00014-86-K-0182.

The work of J. H. Anderson and M. G. Gouda was supported in part by ONR contract N00014-86-K-0763.

Authors' present addresses: A. K. Singh, Department of Computer Science, The University of California at Santa Barbara, Santa Barbara, CA 93106-5110; J. H. Anderson, Department of Computer Science, The University of North Carolina at Chapel Hill, Chapel Hill, NC 27599; M. G. Gouda, Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712-1188.

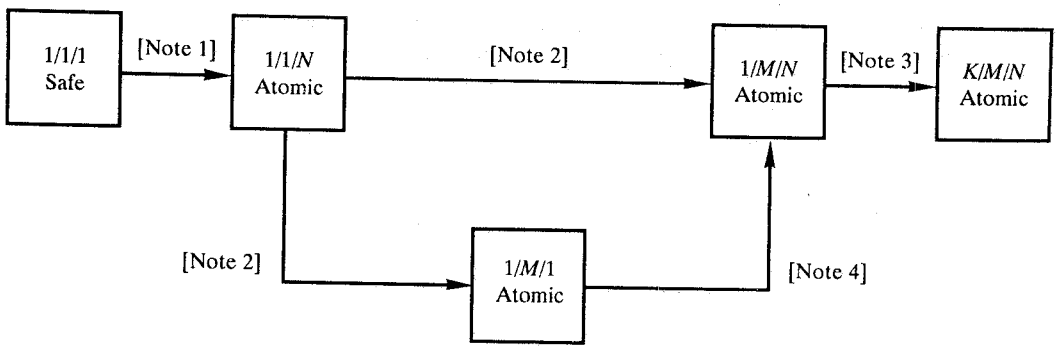
after the other, and not concurrently. This assumption strongly suggests the well-known interleaving semantics of concurrent computations. The validity of this assumption is thus a cornerstone in justifying the present theory of concurrent computing.

One way to check the validity of this assumption is to show that an atomic register can be constructed using a set of more realistic registers that can be read and written concurrently by different processes. In such a construction, a process reads or writes the constructed atomic register by invoking a program; within such a program, only registers of the more realistic kind are read or written. Different programs can be invoked by different processes concurrently; it is required, however, that the net effect resemble that of a serial invocation. The programs are restricted to be wait-free, that is, synchronization primitives, such as P, V, or **await**, and unbounded busy-wait loops are not allowed. This restriction guarantees that a process reads or writes the constructed atomic register in a finite amount of time, regardless of the activities of other processes. (This also means that the read or write of a process is immune to the failure of other processes that also access the register.) The wait-freedom restriction distinguishes the problem of constructing an atomic register from the classic readers-writers problem [Courtois et al., 1971].

Peterson [1983] was the first to suggest the problem of constructing atomic registers from safe registers. A *safe register* is a data object that can be read or written concurrently by different processes; if a read operation overlaps a write operation, then it may return any value from the value domain of the register, and if a read operation does not overlap any write operation, then it obtains the most recently written value. The leap from safe registers to atomic registers is quite large; fortunately, it can be divided into a number of smaller steps. Figure 1 depicts two chains of register constructions that lead from single-writer, single-reader, single-bit safe registers to K-writer, M-reader, N-bit atomic registers. The notation $K/M/N$ denotes a register that can be written by K processes, can be read by M processes, and can store an N-bit value. Each step in the figure is labeled by a reference to the papers in which the given construction is presented.

Henceforth, we concern ourselves only with single-writer atomic registers. The problem of constructing a multiple-reader atomic register from single-reader atomic registers was mentioned as an open problem by Lamport [1986] and by Vitanyi and Awerbuch [1986]. The first solution to the problem was presented by us in [Anderson et al., 1986], where a two-reader construction is given and then generalized to construct an M-reader register from $(M - 1)$-reader registers. This solution, though easy to explain and understand, uses an exponential number of single-reader atomic registers. Subsequently, several solutions with polynomial complexity have been presented [Burns and Peterson, 1987; Kirousis et al., 1987; Newman-Wolfe, 1987; Li et al., 1989], including one given by us in [Singh et al., 1987].

In this paper, we present a construction of a multiple-reader atomic register that is based upon the solution in [Singh et al., 1987]. The presentation of this construction differs from the earlier construction in two respects. First, the solution presented here is of optimal complexity, whereas the one given in [Singh et al., 1987] is not. (Actually, an optimal solution can be attained by combining the solution in [Singh et al., 1987] with constructions by Lamport [1986] and Peterson [1983]—see Singh et al. [1987] for details.) Second, the



NOTES

(1) See Lamport [1986] and Tromp [1989].
(2) See Anderson et al. [1986], Burns and Peterson [1987], Haldar and Vidyasankar [1991], Kirousis et al. [1987], Li et al. [1989], Newman-Wolfe [1987], and Singh et al. [1987].
(3) See Bloom [1988], Li et al. [1989], Peterson and Burns [1987], Schaffer [1988], and Vitanyi and Awerbuch [1986].
(4) See Peterson [1983] and Vidyasankar [1988].

FIG. 1. Two Chains of Register Constructions.

correctness proof presented in this paper is more rigorous and formal (and, we hope, easier to understand).

The rest of the paper is organized as follows. In Section 2, we formally define the problem of constructing an M-reader atomic register from single-reader atomic registers. In Section 3, we present our construction. An informal description of this construction is presented in Section 4 and a formal correctness proof is presented in Section 5; the proof makes use of several lemmas, which are stated and proved in an appendix. Concluding remarks appear in Section 6.

2. *Register Construction*

Register constructions can be defined in a number of different ways. Our choice of definitions is based on simplicity and convenience.

Terminology. In order to avoid confusion, we henceforth capitalize terms such as "Read" and "Write" when they apply to the *constructed* register, and leave them uncapitalized when they apply to the variables used in a construction.

We view the Writer (and each Reader) of a construction as a program that is invoked by a process in order to Write (Read) a value to (from) the register. The program for the Writer has one input parameter indicating the value to be Written; similarly, the program for each Reader has one output parameter indicating the value Read. As an example, see the constructions depicted in Figures 2 and 3.

Each variable of a construction is a single-reader, single-writer atomic register—this restriction arises since our aim is to construct a multiple-reader register from single-reader registers. We also require that all variables be bounded in size. (There is a very simple solution if the variables are unbounded [Vitanyi and Awerbuch, 1986].) As mentioned in the introduction, each program of a construction is "wait-free," that is, synchronization primitives and

```
type WRtype = record old, new : valtype; seq : array[1..M] of 0..2; alt, done : boolean end;
     RRtype = record flag : boolean; seq : 0..2; alt : boolean end
var  WR : array[1..M] of WRtype;
     RW : array[1..M] of 0..2;
     RR[i] : array[i..M] of RRtype for each i in the range 1 ≤ i ≤ M
initialization (∀i : 1 ≤ i ≤ M : RW[i] = WR[i].seq[i] ∧ (∀j : i ≤ i ≤ M : ¬ RR[i, j].flag))

program Writer(val : valtype)
own  new : valtype;
     alt : boolean
var  old : valtype;
     q, seq : array[1..M] of 0..2;
     k : 1..M

initialization (∀i : 1 ≤ i ≤ M : seq[i] = q[i] ⊕ 1 = RW[i])
begin
  0: old, new, alt := new, val, ¬ alt;
  1: for k = 1 to M do read q[k] = RW[k] od;
  2: for k = 1 to M do seq[k] = q[k] ⊕ 1 od
  3: for k = M to 1 do write WR[k] = (old, new, seq[1..M], alt, false) od;
  4: for k = 1 to M do write WR[k] = (old, new, seq[1..M], alt, true) od
end

program Reader(i : 1..M) returns valtype

var  x, y : WRtype;
     v : array[1..i] of RRtype;
     flag : boolean;
     k : 0..N
define (p0 ≡ y.done ∧ x.seq[i] = y.seq[i]) ∧
       (∀k : 0 < k ≤ i  pk ≡ x.seq[i] = y.seq[i] ∧ x.seq[k] = y.seq[k] ∧ x.alt = y.alt ∧
                                  v[k].flag ∧ x.seq[k] = v[k].seq ∧ x.alt = v[k].alt)

initialization x.seq[i] = WR[i].seq[i]
begin
  0: read x = WR[i];
  1: write RW[i] = x.seq[i];
  2: for k = 1 to i do read v[k] = RR[k, i] od;
  3: read y = WR[i];
  4: flag = (∃k : 0 ≤ k ≤ i : pk);
  5: for k = i to M do write RR[i, k] = (flag, y.seq[i], y.alt) od;
  6: if flag then return (y.new) else return (y.old) fi
end
```

FIG. 2. Multiple-reader construction.

busy-wait loops are not allowed. (For a more formal definition of wait-freedom, refer to Anderson and Gouda [1990].)

Next, we define several concepts that are needed to state the correctness condition for a multiple-reader construction. These definitions apply to a given construction.

Definition. A *state* is an assignment of values to the variables of the construction. (Note that each program's "program counter" is considered to be a variable of the construction.) One state is designated as the *initial state*.

Definition. An *event* is an execution of a statement of a program.

Definition. Let t and u be any two states of a construction such that state u is the result of executing some statement at state t. If e is the event corresponding to this statement execution, then we say that e is *enabled* at

```
type: WRtype = record new : valtype; seq : 0..2; alt : boolean end;
      WStype = record old, new : valtype; seq : 0..2; alt, done : boolean end;
      RStype = record flag : boolean; seq : 0.2; alt : boolean end
var   WR : WRtype;
      WS : WStype;
      RW : 0..2;
      RS : RStype
initialization
      ¬ RS.flag

program W(val : valtype)

own   new : valtype;
      alt : boolean
var   old : voltype;
      q, seq : 0..2
begin
      old, new, alt := new, val, ¬ alt;
      read q := RW;
      seq := q ⊕ 1;
      write WS := (old, new, seq, alt, false);
      write WR := (new, seq, alt);
      write WS := (old, new, seq, alt, true)
end

program R returns valtype

var   x, y : WRtype;
      flag : boolean
begin
      read x := WR;
      write RW := x.seq;
      read y := WR;
      flag := x = y;
      write RS := (flag, x.seq, x.alt);
      return(x.new)
end

program S returns valtype

var   x, y : WStype;
      v : RStype;
begin
      read x := WS;
      read v := RS;
      read y := WS;
      if y.done ∨ (x = y ∧ v.flag ∧
                   x.seq = v.seq ∧ x.alt = v.alt) then
        return(y.new)
      else
        return(y.old)
      fi
end
```

FIG. 3. Recursive construction.

state t and we write $t \xrightarrow{e} u$. A *history* of a construction is a sequence $t_0 \xrightarrow{e_0} t_1 \xrightarrow{e_1} \cdots$, where t_0 is the initial state.

Definition. Event e *precedes* another event f in a history iff e occurs before f in the history.

Definition. The set of events in a history corresponding to some complete program execution is called an *operation*. An operation p *precedes* another operation q in a history iff each event of p precedes all events of q.

Observe that the precedes relation is an irreflexive total order on events and an irreflexive partial order on operations.

For the proof of correctness of a construction, it is sufficient to consider only histories in which an initial Write operation precedes all other operations and in which there are no incomplete program executions (that is, operations). From now on, we deal only with such histories. Note that by assuming that

there is an initial Write operation, the value Read by any Read operation can be matched with that Written by some Write operation. This is equivalent to defining the initial state to be equal to that which results after the initial Write operation. Note also that a history with incomplete operations can be extended to one with complete operations; this is possible since all programs are required to be wait-free. By dealing only with complete operations, the value "Read from" or "Written to" the constructed register by an operation is well-defined.

Notation. We denote the ith operation of the Writer, where $i \geq 0$, by $W:i$. (Thus, $W:0$ denotes the initial Write.)

Following Lamport [1986], we define the correctness condition for a construction as follows:

Definition. Let h be any history of a construction. h is said to be *atomic* iff there exists a function ϕ that maps every Read operation in h to some natural number i, where $W:i$ is a Write operation in h, such that the following three conditions hold.

—*Integrity.* For each Read operation r in h, the value Read by r is the same as the value Written by $W:\phi(r)$.
—*Proximity.* For each Read operation r in h, r does not precede the Write operation $W:\phi(r)$ and the Write operation $W:\phi(r+1)$ does not precede r.
—*Precedence.* For any two Read operations r and s in h, if r precedes s, then $\phi(r) \leq \phi(s)$.

Definition. A register construction is *correct* iff all its histories are atomic.

3. *Multi-Reader Construction*

The proposed construction depicted in Figure 2 consists of a Writer program and a program for each Reader i, where $1 \leq i \leq M$. Each shared variable in the construction is of the single-reader kind. The interface between the Writer and Reader i consists of a variable $WR[i]$ that is written by the Writer and read by Reader i and a variable $RW[i]$ that is written by Reader i and read by the Writer. The interface across Readers consists of a set of variables $RR[i,j]$, where $i \leq j$. Variable $RR[i,j]$ is written by Reader i and read by Reader j.[1] An explanation of the field names appearing in the **type** definitions is as follows:

alt. A bit that alternates in value with each operation of the Writer.
done. A bit that distinguishes the two values written by a Write operation to variables $WR[i]$.
new. The "current" value of the constructed register. (*valtype* is the type of the constructed register.)
old. The "previous value of the constructed register.
flag. A bit that indicates whether a Read operation returned the *old* or the *new* value.
seq. A modulo-3 integer "sequence number." ($\oplus$ denotes modulo-3 addition.)

[1] It is possible to eliminate variables $RR[i,i]$ from the construction. However, including these variables simplifies the proof of correctness.

Variables that are local to a program are declared in either a **var** or an **own** block. A variable declared in an **own** block is assumed to retain its value across invocations of the corresponding program. A variable declared in a **var** block is not assumed to retain its value across invocations. The **initialization** assertions that follow the variable declarations serve to define appropriate initial values for the variables of the construction; any state satisfying these assertions is a suitable initial state. (However, recall that, by assumption, the initial Write operation $W:0$ precedes all Read operations.)

In the programs of the construction, we use a special syntax in order to distinguish reads and writes of shared variables from reads and writes of local variables. A program reads a given shared variable Z by executing a statement of the form "**read** $u := Z$," where u is a local variable of the same type as Z. A program writes a shared variable Z by executing a statement of the form "**write** $Z := u$." If variable Z consists of m fields, then u is an m-tuple; the ith component of u is a local variable whose value is to be stored in the ith field of Z. We use similar names (sometimes identical) for the components of u and the fields of Z, so the correspondence should be obvious.

The sequence numbers shared between the Writer and the Readers form the basis of our construction. Included in every value written by the Writer is a set of sequence numbers, one per Reader. During each Write operation, the Writer reads variable $RW[k]$, where $1 \leq k \leq M$, to obtain the most recent sequence number of Reader k. A new sequence number for Reader k is then obtained by incrementing that read from $RW[k]$ using modulo-3 addition. The Writer then writes to the Readers in two passes; in the first pass the Writer writes to the Readers in order from M to 1, and in the second pass this order is reversed. The *done* bit distinguishes the two passes. The value that the Writer writes to each Reader includes both the previous and current value of the constructed register, the aforementioned set of sequence numbers, and the *alt* and *done* bits.

Each Reader i reads two values from the Writer, x and y. Between these reads, the sequence number obtained from the first read is written back to the Writer and a value is read from each Reader k, where $k \leq i$. The values read from the Writer and Readers are used to compute the *flag* bit, which indicates whether the old or new value from y is to be returned. Note that *flag* is assigned a value based upon the expressions $p_0, \ldots, p_t$. These expressions have been introduced as a shorthand, and are defined in the **define** section. Before returning a value, Reader i writes a value to each Reader k, where $k \geq i$. This value includes Reader i's *flag* bit and also the sequence number for Reader i and *alt* bit of the Writer obtained during Reader i's second read from $WR[i]$.

We now calculate the space complexity of our construction by determining the number of shared single-writer, single-reader safe bits required. The size of each shared variable in the construction is as follows:

—$WR[i]$, $1 \leq i \leq M$, consists of $2M + 2N + 2$ bits.
—$RW[i]$, $1 \leq i \leq M$, consists of 2 bits.
—$RR[i, j]$, $1 \leq i \leq j \leq M$, consists of 4 bits.

Using the construction of Vidyasankar [1990], a single-writer, single-reader, B-bit atomic register can be constructed using $3B + 7$ shared single-writer, single-reader safe bits. For our construction, this yields a space complexity of $6M^2 + 6MN + 26M + 19M(M + 1)/2$. It is well known that the lower bound

on space for this problem is $O(M^2 + MN)$ bits [Israeli and Li, 1987]; thus, our construction is asymptotically optimal.

Given the correctness proof in Section 5, our construction establishes the following theorem.

THEOREM. *It is possible to construct a single-writer, M-reader, N-bit atomic register using $O(M^2 + MN)$ shared single-writer, single-reader safe bits.*

4. *Informal Explanation*

In this section, we explain the intuition behind the algorithm by discussing some of the proof obligations that will be established in the next section and in the appendix. Before doing so, however, we introduce some notation that will be useful in the ensuing discussion.

Definition. Let p be an operation of program P and i be a label of a statement in P. If i is not a **for** loop, then $p:i$ denotes the event corresponding to the execution of statement i in operation p. Otherwise, if i is a **for** loop, then $p:i.j$ denotes the event corresponding to the execution of the loop body when the loop counter equals j.

For example, for a Write operation w of the construction in Figure 2, $w:1.i$ denotes the event in which w reads from $RW[i]$ and $w:4.i$ denotes the event in which w writes to $WR[i]$ for the second time. Next, we define three types of control predicates [Lamport, 1980].

Definition. Let p be an operation of program P in some history and let $p:i$ be an event of p. Then, p **at** i is true at a state of the history iff the event $p:i$ is enabled; p **before** i is true at a state iff the state occurs before the event $p:i$; and p **after** i is true at a state iff the state occurs after the event $p:i$.

If n is the label of a **for** loop in program P, then we use p **before** n as a shorthand for $(\forall i::p$ **before** $n.i)$; we use p **after** n as a shorthand for $(\forall i::p$ **after** $n.i)$; and we use p **at** n as a shorthand for $(\exists i::p$ **at** $n.i) \vee (\exists i, j::p$ **after** $n.i \wedge p$ **before** $n.j)$.

Observe that if i is not a **for** loop, then p **at** i implies p **before** i. In particular, p **at** i strengthens p **before** i by also requiring that $p{:}i$ be enabled. On the other hand, if i is a **for** loop, then it is possible to have p **at** $i \wedge \neg(p$ **before** $i)$. This assertion holds when some event of the loop other than the first is enabled. The following assertions are a consequence of the preceding definition; let p and i be as given in the definition.

—p **before** $i \vee p$ **at** $i \vee p$ **after** i
—$\neg(p$ **after** $i) = p$ **at** $i \vee p$ **before** i

As examples of control predicates, observe that for any Write operation w of the construction in Figure 2:

—w **after** $1.i$ denote that w has read the sequence number from Reader i,
—w **before** 4 denotes that the w has not begun its second pass of writes,
—w **after** $4.i$ denotes that w has completed its write to Reader i in the second pass, and
—w **after** 4 denotes that w has completed its second pass of writes.

Definition. Let A and B be two state assertions. The assertion A **unless** B holds iff for every pair of consecutive states in any history, if $A \wedge \neg B$ holds in the first state, then $A \vee B$ holds in the second state.

Our notion of **unless** has been borrowed from the UNITY logic of Chandy and Misra [1988]. Informally, A **unless** B means that once A becomes true, it remains true unless B becomes true. In particular, if A is falsified by some event, then B either is true at the state prior to the occurrence of that event, or is true at the state following the occurrence of that event. There is no requirement that B eventually become true; however, in that case A remains true forever.

Definition. Let p be an operation, and z be any local variable of p. Then, $p!z$ denotes the final value of variable z as assigned by operation p.

Now, we are ready to explain the intuitive idea of the algorithm. Let us examine some Write operation w of the Writer. Assume that w changes the value of the register from *old* to *new*. It is possible to identify two points a and b within the execution of w such that no Reader returns *new* before a and no Reader returns *old* after b. Let us call the interval between a and b the *uncertainty interval* of w.

In establishing the correctness of the construction, the main difficulty is ensuring that the precedence condition (in the definition of an "atomic history" given in Section 2) is not violated during the uncertainty interval of a Write operation such as w. This amounts to proving that new-then-old conflicts do not arise in this interval. Avoiding new-then-old conflicts is difficult because of the fact that only single-reader registers are used in the construction. As a result of this limitation, w must inform the Readers of the new value being Written one at a time. Thus, during the uncertainty interval of w, there exist certain points at which some, but not all, of the Readers have been informed by w that the value *new* is currently being Written. This difficulty will be encountered in *any* construction of a multiple-reader atomic register from single-reader ones.

In our solution, the uncertainty interval begins after the Writer writes to $WR[1]$ in the second pass and ends after the Writer writes to $WR[M]$ in the second pass. In other words, the uncertainty interval exists while the predicate w **at** $4 \wedge \neg w$ **before** 4 is true. Our solution should therefore guarantee the following three properties:

(A0) Reads before the beginning of this interval do not return the new value.
(A1) Reads after the end of this interval do not return the old value.
(A2) Reads during this interval do not result in new-then-old conflicts.

To state these properties precisely, we define a predicate *Cue* that relates the values written by a particular Write operation to the existing value of variable $RR[i, j]$, where $i \leq j$. Informally, predicate $Cue(w, i, j)$ can be thought of as a cue from Reader i to Reader j that Reader i has returned the new value of w. This predicate is defined as follows:

$$Cue(w, i, j) \equiv RR[i, j].flag \wedge RR[i, j].seq = w!seq[i] \wedge RR[i, j].alt = w!alt.$$

According to the Reader program, if both values read from $WR[j]$ by Reader j were written by Write operation w, and if the value read from $RR[i, j]$ satisfies the above predicate, then Reader j will find predicate p_t to be true and subsequently return the value *new*.

Returning to the discussion of the uncertainty interval, by property (A0), we should ensure that $Cue(w, i, j)$ is false before the beginning of the interval. In other words,

$$w \textbf{ after } 1.i \wedge w \textbf{ before } 4 \Rightarrow \neg\, Cue(w, i, j).$$

Control predicate w **after** $1.i$ is added as a conjunct to ensure that the value $w!seq[i]$, which is used in the definition of $Cue(w, i, j)$, is available to Reader i. We prove the above property as Lemma 4 in the appendix. Based on this lemma, Readers do not return the new value before the beginning of the uncertainty interval and property (A0) holds.

Now, consider an operation r of Reader i and assume that both values read from $WR[i]$ by r were written by w. If r begins after the uncertainty interval of w, then r will find p_0 to be true; consequently, r will return the value *new*. Thus, Readers do not return the old value after the end of this interval and hence property (A1) holds.

Finally, we have to ensure that property (A2) holds, that is, that new-then-old conflicts are avoided within the uncertainty interval. Consider successive Read operations by Reader i and Reader j. There are two cases to consider: either $i \leq j$ or $j < i$. The former case is not too difficult. If an operation of Reader i returns the value *new* during the uncertainty interval of w, then, upon completing, it establishes $Cue(w, i, j)$. If we can show that this predicate will remain stable during the uncertainty interval, then Reader j will also return the value *new* and new-then-old conflicts will not arise. The stability of $Cue(w, i, j)$ is captured by the following property,

$$w \textbf{ at } 4 \wedge Cue(w, i, j) \textbf{ unless } w \textbf{ after } 4,$$

which is proved as Lemma 5 in the appendix.

The latter case is more interesting, as there is no direct communication from Reader i to Reader j if $j < i$. In order to avoid new-then-old conflicts in this case, we have to rely upon either the Writer completing its writes to Reader j or some other Reader $k < j$ setting $Cue(w, k, j)$ to true. In other words, we have to show that

$$w \textbf{ at } 4 \wedge Cue(w, l, i) \Rightarrow (\forall j\colon j < i\colon w \textbf{ after } 4.j \vee (\exists k\colon k < j\colon Cue(w, k, j))).$$

Let us examine the above assertion in detail. It states that, during the uncertainty interval of w, if Reader i is cued by some Reader l (and hence, returns the new value), then for any smaller j (to which Reader i does not write any values), either w has written its final values to Reader j (in which case Reader j will return the new value) or Reader j in turn has been cued by some Reader k (in which case, from the stability of $Cue(w, k, j)$, Reader j will again return the new value). This property is proved as Lemma 9 in the appendix. This completes the last remaining obligation, (A2), of the proof that new-then-old conflicts do not occur. Our formal proof in the next section mirrors the intuitive explanation presented here.

5. *Proof of Correctness*

We prove that our construction is correct by defining a function ϕ for a given history, and by showing that the defined ϕ meets the three conditions of integrity, proximity, and precedence defined in Section 2. The following notational conventions and definitions are used in the proof.

Notation. In order to avoid using too many parentheses, we define a binding order for the symbols that we use. The following is a list of these symbols, grouped by binding power; the groups are ordered from highest binding power to lowest.

$[\,], (\,)$
.
$!, :$
$+, -, \oplus$
$=, \neq, <, >, \leq, \geq, \prec, \preccurlyeq$
$\neg$
$\wedge, \vee$
$\Rightarrow$
$\equiv$

Definition. Let e and f be two events in some history. Then, $e \prec f \equiv e$ precedes f, and $e \preccurlyeq f \equiv (e \prec f) \vee (e = f)$.

Definition. Let e be the event corresponding to the execution of the statement **read** $z := Y$ in an operation p, where z is a local variable and Y is a shared variable. If the value that p reads from Y is written by operation q, then we say that operation q *determines* $p!z$.

Observation. Let r and s be any two operations of Reader i and Reader j, respectively, such that $r!y$ is determined by the Write operation $W:m$ and $s!y$ is determined by Write operation $W:n$. If r precedes s and $m > n$, then $m = n + 1$ and $i > j$.

PROOF.

$W:m$ determines $r!y$
$\Rightarrow$ {definition of determines}
$(W:m):3.i \prec r:3$
$\Rightarrow$ {r precedes s}
$(W:m):3.i \prec r:3 \prec s:3$
$\Rightarrow$ {$W:n$ determines $s!y$ and definition of determines}
$(W:m):3.i \prec r:3 \prec s:3 \prec (W:(n+1)):3.j$
$\Rightarrow$ {transitivity of $\prec$}
$(W:m):3.i \prec (W:(n+1)):3.j$
$\Rightarrow$ {Write operations occur sequentially in a history}
$m \leq n+1 \wedge (W:m):3.i \prec (W:(n+1)):3.j$
$\Rightarrow$ {$m > n$}
$m = n+1 \wedge (W:m):3.i \prec (W:(n+1)):3.j$
$\Rightarrow$ {predicate calculus}
$m = n+1 \wedge (W:m):3.i \prec (W:m):3.j$
$\Rightarrow$ {program for Writer}
$m = n+1 \wedge i > j$ □

Definition. Let r be any Read operation, and suppose that Write operation $W:m$ determines $r!y$. Then, $\phi(r)$ is defined as follows:

$$\phi(r) = \begin{cases} m & \text{if } r!\mathit{flag}, \\ m-1 & \text{otherwise.} \end{cases}$$

Observe that $\phi(r)$ is nonnegative. To see this, recall that the initial Write operation $W:0$, by assumption, precedes all Read operations. Thus, if $r!y$ is determined by $W:0$, then $r!y.done \wedge r!x = r!y$ holds. This implies that $r!flag$ holds; hence, $\phi(r) = 0$.

PROOF OF INTEGRITY. Let r be any Read operation, and suppose that Write operation $W:m$ determines $r!y$. If $\phi(r) = m$, then $r!flag$ is true (definition of ϕ) and r returns the value $r!y.new$, that is, the value Written by $W:m$. If $\phi(r) = m - 1$, then $r!flag$ is false (definition of ϕ) and r returns the value $r!y.old$, that is, the value Written by $W:(m-1)$. □

PROOF OF PROXIMITY. Let r be an operation of Reader i, and suppose that Write operation $W:m$ determines $r!y$. Because $W:m$ determines $r!y$, r does not precede $W:m$ and $W:(m+1)$ does not precede r. Thus, if $\phi(r) = m$, then proximity is satisfied.

In the case $\phi(r) = m - 1$, again since r does not precede $W:m$, clearly r does not precede $W:(m-1)$. We next show that $W:m$ does not precede r. From the definition of ϕ, $\phi(r) = m - 1$ implies that $r!flag$ is false. Hence, $r!p_0$ is also false. By the definition of p_0, this implies that $r!y.done$ is false or $r!x.seq[i] \neq r!y.seq[i]$. If $r!y.done$ is false, then $r:3 \prec (W:m):4.i$. If $r!x.seq[i] \neq r!y.seq[i]$, then $r:0 \prec (W:m):3.i$. Thus, in either case $W:m$ does not precede r. □

PROOF OF PRECEDENCE. The proof of precedence is quite complicated and consists of a somewhat lengthy case analysis, most of which has been relegated to the appendix. Here, we make use of Lemmas 6, 10, and 11, which are proved there. These three lemmas are based on Lemma 4, 5, and 9 discussed in the informal description in the previous section.

Let r be any operation of Reader i, and s be any operation of Reader j such that r precedes s. Our proof obligation is to show that $\phi(r) \leq \phi(s)$.

Assume that Write operations $W:m$ and $W:n$ determine $r!y$ and $s!y$, respectively. Observe the following:

$$
\begin{aligned}
& \phi(r) \leq \phi(s) \\
= \; & \{m \leq n-1 \vee m \geq n\} \\
& (m \leq n-1 \Rightarrow \phi(r) \leq \phi(s)) \wedge (m \geq n \Rightarrow \phi(r) \leq \phi(s)) \\
= \; & \{\text{by definition of } \phi,\ \phi(r) \leq m \text{ and } n-1 \leq \phi(s);\ \text{thus,} \\
& \quad m \leq n-1 \Rightarrow \phi(r) \leq \phi(s)\} \\
& m \geq n \Rightarrow \phi(r) \leq \phi(s) \\
= \; & \{(m \geq n) = (m > n \vee m = n)\} \\
& (m > n \Rightarrow \phi(r) \leq \phi(s)) \wedge (m = n \Rightarrow \phi(r) \leq \phi(s)) \\
= \; & \{\text{from observation proved earlier,} \\
& \quad r \text{ precedes } s \Rightarrow ((m > n) = (m = n+1 \wedge i > j))\} \\
& ((m = n+1 \wedge i > j) \Rightarrow \phi(r) \leq \phi(s)) \wedge (m = n \Rightarrow \phi(r) \leq \phi(s)) \\
= \; & \{\text{by definition of } \phi,\ m = n+1 \Rightarrow ((\neg r!flag \wedge s!flag) = (\phi(r) \leq \phi(s)))\} \\
& ((m = n+1 \wedge i > j) \Rightarrow \neg r!flag \wedge s!flag) \wedge (m = n \Rightarrow \phi(r) \leq \phi(s)) \\
= \; & \{\text{Lemma 10 in the appendix}\} \\
& m = n \Rightarrow \phi(r) \leq \phi(s) \\
= \; & \{\text{by definition of } \phi,\ \phi(r) = m-1 \vee \phi(r) = m\} \\
& ((\phi(r) = m-1 \wedge m = n) \Rightarrow \phi(r) \leq \phi(s)) \wedge \\
& ((\phi(r) = m \wedge m = n) \Rightarrow \phi(r) \leq \phi(s))
\end{aligned}
$$

$=$ {by definition of ϕ, $n - 1 \leq \phi(s)$; thus, $\phi(r) = m - 1 \wedge m = n \Rightarrow \phi(r) \leq \phi(s)$}
$(\phi(r) = m \wedge m = n) \Rightarrow \phi(r) \leq \phi(s)$
$=$ {by definition of ϕ, $\phi(s) \leq n$}
$(\phi(r) = m \wedge m = n) \Rightarrow \phi(s) = n$
$=$ {by definition of ϕ and *flag*, $(\phi(r) = m) = (\exists k : k \leq i : r!p_k)$ and $(\phi(s) = n) = (\exists k : k \leq j : s!p_k)$}
$((\exists k : k \leq i : r!p_k) \wedge m = n) \Rightarrow (\exists k : k \leq j : s!p_k)$
$=$ {$i \leq j \vee i > j$}
$((i \leq j \wedge (\exists k : k \leq i : r!p_k) \wedge m = n) \Rightarrow (\exists k : k \leq j : s!p_k)) \wedge$
$((i > j \wedge (\exists k : k \leq i : r!p_k) \wedge m = n) \Rightarrow (\exists k : k \leq j : s!p_k))$
$=$ {Lemmas 6 and 11 in the appendix}
true □

6. *Discussion*

We have shown that a single-writer, M-reader, N-bit atomic register can be constructed in a wait-free manner using single-writer, single-reader atomic registers. Our construction requires $O(M^2 + MN)$ shared single-writer, single-reader safe bits, which is asymptotically optimal.

Our definition of atomicity is equivalent to that given by Misra [1986]. His axioms for atomicity in essence require that all read and write operations be shrunk to a point; such a shrinking of operations is possible iff a function ϕ that meets the three conditions of our definition exists. Recently, Herlihy and Wing [1990] have extended the idea of atomicity to arbitrary abstract data types by defining the concept of *linearizability*. Though akin to serializability, the usual correctness criterion for concurrent execution of transactions, there are some subtle differences between the two concepts. One important distinction is that linearizability is a local correctness condition whereas serializability is not. We refer the reader to Herlihy and Wing [1990] for further details.

In order to prove the correctness of a multiple-reader atomic register construction, a function ϕ that meets the three conditions of integrity, proximity, and precedence has to be defined for every possible history. This leads to long, and somewhat tedious proofs, mainly because such a proof must take into account all possible ways in which Reads and Writes can overlap. To keep the resulting case analysis in our proof to a minimum, we chose the function ϕ to be very simple; it depends only on the boolean variable *flag*. If our proof appears formidable, in spite of this simplification, then it is because we have been very formal in our reasoning, so as to leave no doubt about the validity of the proof. However, given the length of our proof, it seems reasonable to inquire whether there exist other approaches to constructing an M-reader register that facilitate simpler correctness arguments. We briefly describe such an approach next.

The main idea behind this approach is to develop a construction of an M-reader ($M \geq 2$) register from a collection of $(M - 1)$-reader registers. An M-reader register can then be implemented from single-reader registers by recursively applying the construction, first to replace all $(M - 1)$-reader registers by $(M - 2)$-reader ones, then to replace all $(M - 2)$-reader registers by $(M - 3)$-reader ones, and so on.

A simple implementation of an M-reader register from $(M-1)$-reader registers is depicted in Figure 3. In this construction, the Writer and Reader 1 are denoted as W and R, respectively. The remaining $(M-1)$ Readers are denoted S_1 through S_{M-1}; these Readers execute the program called S. The construction uses two single-reader shared variables, WR and RW, and two $(M-1)$-reader shared variables, WS and RS. Each variable's name indicates those programs that read and write it, respectively. For example, WS is written by the Writer W and is read by Readers S_1 through S_{M-1}. Observe that each of the Readers S_1 through S_{M-1} writes no shared variables. This fact is crucial and is exploited in recursive applications of the construction. The local variables used in the construction are similar to those used in the construction in Figure 2.

The primary advantage of the above recursive construction is that the proof of correctness can be simplified by assuming that there are only two Readers, namely R and S. Specifically we can ignore the possible interleavings that may arise among Readers S_1 through S_{M-1} when considering the proofs of integrity, proximity, and precedence. This follows quite easily for the proofs of integrity and proximity, as these conditions relate an individual Read operation with operations of the Writer. As for the proof of precedence, note that the precedence condition only restricts the values returned by Read operations that are in a strict precedence relationship with one another. Because Readers S_1 through S_{M-1} execute identical programs and write no shared variables, the only precedence relationships that are of interest are the following: a Read operation of R precedes a Read operation of some S_j; a Read operation of R precedes another Read operation of R; a Read operation of some S_j precedes a Read operation of R; and a Read operation of some S_j precedes a Read operation of some S_k. These cases are precisely those that arise in the special case when $M = 2$. Thus, the proof of correctness for the M-reader register reduces to the much simpler task of proving the correctness of a two-reader construction consisting of Readers R and S.

A formal proof of correctness for the construction of Figure 3 appears in Anderson et al. [1986], where this construction was first presented. In the remainder of this section, we informally describe how the construction works for two Readers by comparing it to the general construction given earlier in Figure 2. (In the general construction considered here, we assume that variables $RR[i, i]$, where $1 \le i \le M$, are removed; see the footnote at the beginning of Section 3. Hence, in the Reader program, the loop index in statement 2 is from 1 to $i-1$, and the loop index in statement 5 is from $i+1$ to M.) We begin our comparison by considering a variant of the two-reader version of the general construction in Figure 2. This variant is shown in Figure 4(a). In this variant, some statements have been combined into larger atomic statements, which we denote by enclosing them within angle brackets "⟨"and "⟩." Note also that we have renamed the programs and variables to coincide with the names given in Figure 3. We have also moved the assignment to $seq[1]$ so that it occurs immediately after the read from RW. These changes to our original construction clearly do not affect the construction's correctness. Thus, because our original construction is correct, the construction shown in Figure 4(a) is also correct.

Now, consider the code for W in Figure 4(a). The fifth atomic statement of W always assigns the value true to $WR.done$. Hence, the *done* field of WR can

```
program W(val : valtype)
begin
  old, new, alt := new, val, ¬ alt;
  read q[1] := RW;
  seq[1] := q[1] ⊕ 1;
  ⟨read q[2] := SW;
  seq[2] := q[2] ⊕ 1;
  write WS := (old, new, seq[1], seq[2], alt, false)⟩;
  ⟨write WR := (old, new, seq[1], seq[2], alt, false);
  write WR := (old, new, seq[1], seq[2], alt, true)⟩;
  write WS := (old, new, seq[1], seq[2], alt, true)
end
```

```
program R returns valtype
begin
  read x := WR;
  write RW := x.seq[1];
  read y := WR;
  flag := p0;
  write RS := (flag, y.seq[1], y.alt);
  if flag then return(y.new)
          else return(y.old) fi
end
```

```
program S returns valtype
begin
  ⟨read x := WS;
  write SW := x.seq[2];
  read v := RS;
  read y := WS⟩;
  flag := p0 ∨ p1;
  if flag then return(y.new)
          else return(y.old) fi
end
```

(a)

```
program W(val : valtype)
begin
  old, new, alt := new, val, ¬ alt;
  read q[1] := RW;
  seq[1] := q[1] ⊕ 1;
  write WS := (old, new, seq[1], alt, false);
  write WR := (old, new, seq[1], alt);
  write WS := (old, new, seq[1], alt, true)
end
```

```
program S returns valtype
begin
  ⟨read x := WS;
  read v := RS;
  read y := WS⟩;
  flag := y.done ∨ (x = y ∧ v[1].flag ∧
    x.seq[1] = v[1].seq ∧ x.alt = v[1].alt);
  if flag then return(y.new)
          else return (y.old) fi
end
```

(b)

FIG. 4. Two intermediate constructions.

be removed without affecting the construction's correctness. With this change, the W's fifth atomic statement can be replaced by a single write to WR.

Next, consider Reader S. The first atomic statement of S assigns the same values to local variables x and y. Thus, in Reader S's calculation of *flag*, which depends on p_0 and p_1 as defined in Figure 2, $x.seq[2] = y.seq[2]$ is a tautology and can be removed. With this change, the $seq[2]$ fields of the construction serve no useful purpose and hence all can be removed. Finally, the condition $x.seq[1] = y.seq[1] \wedge x.alt = y.alt$ in Reader S's computation of *flag* is also a tautology. Instead of removing it, we choose to replace it by another tautology $x = y$. These changes, which clearly do not affect the construction's correctness, yield the code given in Figure 4(b). In this figure, we have not shown Reader R, as its code has not been changed.

Note that, in the construction of Figure 4(b), the only statement that reads or writes multiple shared variables is the first statement of Reader S—all other such statements have been eliminated. It turns out that with a few slight changes to the code for Reader R, the first statement of Reader S can be broken into three separate statements. The required changes to Reader R are as follows: Reader R computes its *flag* by $flag := x = y$ instead of $flag :=$

$y.done \wedge x.seq[1] = y.seq[1]$; Reader R assigns ($flag$, $x.seq[1]$, $x.alt$) to RS rather that ($flag$, $y.seq[1]$, $y.alt$); and Reader R always returns $x.new$. With the last change, it turns out that the Writer no longer needs to write the old value to Reader R. The resulting construction is the same as that given in Figure 3. This completes our comparison of the two constructions.

Although the construction of Figure 3 has a simpler correctness proof than that of Figure 2, this simplicity comes at a price. In particular, as shown in [Anderson et al., 1986], this construction of a multiple-reader register requires a number of bits that is exponential in the number of Readers.

Appendix. Remainder of Correctness Proof

Here we prove the lemmas that were used in the proof given in Section 5. As outlined in the informal discussion of the algorithm in Section 4, Lemmas 4, 5, and 9 relate the *uncertainty interval* of a Write operation to the values of variables $RR[i, j]$, which are used for communication between Readers. Lemmas 1, 2, and 3 state some elementary results that are used in proving subsequent lemmas. Lemmas 7 and 8 are used to prove some results that are used in the proof of Lemma 9. Lemmas 6, 10, and 11 consider the values returned by two successive Read operations and are used in the proof of precedence presented in Section 5.

A few words concerning the structure of the proofs are in order at this juncture. In most of our proofs, the reasoning is based upon sequences of states or events. (In other words, we assume a total order on all reads and writes of internal variables in the construction.) Wherever possible, the proofs have been simplified through the introduction of invariants. In establishing these invariants, we usually proceed by a case analysis on the order of events that may "affect" one another. For example, consider a Write operation w and a Read operation r of Reader i. Operation w reads a sequence number from Reader i and operation r writes a sequence number to the Writer. So, it is possible to do a case analysis on whether w's read of the sequence number occurs before r's write of the sequence number. On the whole, the proofs are not difficult, but, due to the numerous interleavings of events that may potentially occur, they are rather lengthy. The following definitions will be used in the proofs; the first is repeated from Section 4.

Definition. Let w be a Write operation, and let $1 \leq i \leq j \leq M$. Then,

$$Cue(w, i, j) \equiv RR[i, j].flag \wedge RR[i, j].seq = w!seq[i] \wedge RR[i, j].alt = w!alt.$$

Definition. Consider the history $t_0 \xrightarrow{e_0} \cdots t_i \xrightarrow{e_i} t_{i+1} \xrightarrow{e_{i+1}} \cdots$. We say that t_i is the state *prior to* the event e_i and t_{i+1} is the state *following* e_i. Similarly, e_i is the event *prior to* the state t_{i+1} and e_{i+1} is the event *following* the state t_{i+1}.

LEMMA 1. *Let r be an operation of Reader i such that $r!p_k$ holds for some $k \leq i$. Let w be the Write operation that determines $r!y$. Then, $w:1.i \prec r:1$. Moreover, if $k \neq 0$, then $Cue(w, k, i)$ holds at the state prior to the event $r:2.k$.*

PROOF. By the program for the Writer, $w:1.i \prec w:3.i$. Because w determines $r!y$, $w:3.i \prec r:3$. Therefore, by transitivity, $w:1.i \prec r:3$. This implies that $w!q[i]$ is determined by either r or some predecessor of r.

We now show that r does not determine $w!q[i]$. Because $r!p_k$ holds, $r!x.seq[i] = r!y.seq[i]$. Because w determines $r!y$, $r!y.seq[i] = w!seq[i]$. Therefore, by transitivity, we have

$$r!x.seq[i] = w!seq[i]. \tag{1}$$

If r determines $w!q[i]$, then $w!q[i] = r!x.seq[i]$. As w assigns $seq[i] := q[i] \oplus 1$, this implies that $w!seq[i] = r!x.seq[i] \oplus 1$, contrary to (1). Therefore, r does not determine $w!q[i]$. Thus, we conclude that $w!q[i]$ is determined by a predecessor of r, that is, $w:1.i \prec r:1$.

For the second part of the proof, assume that $k \neq 0$. Let t be the state prior to the event $r:2.k$. Because $r!p_k$ is true and because $k > 0$, the following assertion holds at state t.

$$RR[k,i].flag \wedge RR[k,i].seq = r!x.seq[k] \wedge RR[k,i].alt = r!x.alt.$$

Because $k > 0$, we have, by the definition of p_k, $r!x.seq[k] = r!y.seq[k]$ and $r!x.alt = r!y.alt$. Because w determines $r!y$, this implies that $r!x.seq[k] = w!seq[k]$ and $r!x.alt = w!alt$. Thus, by transitivity, the following assertion holds at state t.

$$RR[k,i].flag \wedge RR[k,i].seq = w!seq[k] \wedge RR[k,i].alt = w!alt$$

Therefore, from the definition of *Cue*, $Cue(w,k,i)$ holds at state t. □

The following lemma relates the sequence numbers read by two consecutive operations of the same Reader i. It states that these values do not differ by more than one. The intuition is as follows: In order for Reader i's sequence number to be incremented by 1, it must first be written by Reader i to $RW[i]$, then be read by the Writer, then be incremented by the Writer and written to $WR[i]$, and finally be read again by Reader i. Between two reads from $WR[i]$ by two consecutive Read operations, this complete sequence of events can happen at most once.

LEMMA 2. *Let r and s be consecutive operations by Reader i. Then,*

$$s!x.seq[i] = r!x.seq[i] \vee s!x.seq[i] = r!x.seq[i] \oplus 1.$$

PROOF. We prove the lemma by first showing that the following assertion is an invariant.

$$B \equiv WR[i].seq[i] = RW[i] \vee WR[i].seq[i] = RW[i] \oplus 1.$$

To prove that B is an invariant, we consider the assertions $B0, \ldots, B4$ defined below and show that $B0 \vee \cdots \vee B4$ is an invariant. In these assertions, we refer to the local variables $q[i]$ and $seq[i]$ of the Writer and $x.seq[i]$ of Reader i.

$$\begin{array}{lllll}
B0 \equiv RW[i] & = q[i] \oplus 1 & = seq[i] & = WR[i]seq[i] & = x.seq[i] \\
B1 \equiv RW[i] & = q[i] & = seq[i] & = WR[i].seq[i] & = x.seq[i] \\
B2 \equiv RW[i] \oplus 1 & = q[i] \oplus 1 & = seq[i] & = WR[i].seq[i] \oplus 1 & = x.seq[i] \oplus 1 \\
B3 \equiv RW[i] \oplus 1 & = q[i] \oplus 1 & = seq[i] & = WR[i].seq[i] & = x.seq[i] \oplus 1 \\
B4 \equiv RW[i] \oplus 1 & = q[i] \oplus 1 & = seq[i] & = WR[i].seq[i] & = x.seq[i].
\end{array}$$

To see that $B0 \vee \cdots \vee B4$ is an invariant, observe the following:

—$B0$ is initially true, and the only statement that can possibly falsify it is the **read** by the Writer from $RW[i]$. But, executing this statement when $B0$ is true establishes $B1$.
—The only statement that can possibly falsify $B1$ is the assignment to $seq[i]$ by the Writer. But, executing this statement when $B1$ is true establishes $B2$.
—The only statement that can possibly falsify $B2$ is the first **write** by the Writer to $WR[i]$. But, executing this statement when $B2$ is true establishes $B3$.
—The only statement that can possibly falsify $B3$ is the **read** by Reader i from $WR[i]$. But, executing this statement when $B3$ is true establishes $B4$.
—The only statement that can possibly falsify $B4$ is the **write** by Reader i to $RW[i]$. But, executing this statement when $B4$ is true establishes $B0$.

Thus, we conclude that $B0 \vee \cdots \vee B4$ is an invariant. This also implies that B is an invariant since $(B0 \vee \cdots \vee B4) \Rightarrow B$.

We now use invariant B to show that the lemma holds. Our proof obligation is as follows:

$$s!x.seq[i] = r!x.seq[i] \vee s!x.seq[i] = r!x.seq[i] \oplus 1.$$

Let t denote the state prior to the event $s:0$. Because r and s are consecutive, the value of $RW[i]$ at state t equals $r!x.seq[i]$, and the value of $WR[i].seq[i]$ at state t equals $s!x.seq[i]$. Since B is an invariant, either

—$WR[i].seq[i] = RW[i]$ at state t, in which case $s!x.seq[i] = r!x.seq[i]$, or
—$WR[i].seq[i] = RW[i] \oplus 1$ at state t, in which case $s!x.seq[i] = r!x.seq[i] \oplus 1$. □

The following lemma relates the value written to $RR[i, j]$ by an operation r of Reader i to the values written by an "overlapping" or succeeding Write operation w. The proof of this lemma makes use of Lemma 2.

LEMMA 3. *Let r be an operation of Reader i, and let w be a Write operation such that $r:1 \prec w:1.i$. Furthermore, let t be any state at which (w **after** $1.i$) holds. If the value appearing in $RR[i, j]$, $i \leq j$, at state t is written by r, then $Cue(w, i, j)$ is false at t.*

PROOF. Let t be any state for which w **after** $1.i$ holds and i and j be indices such that $i \leq j$. Assume that $r:1 \prec w:1.i$ and that the value appearing in $RR[i, j]$ at state t is written by r. Our proof obligation is to show that $Cue(w, i, j)$ is false at t.

We first show that $r!x.seq[i] \neq w!seq[i]$. Let e be the event prior to state t. Because w **after** $1.i$ holds at t, $w:1.i \preccurlyeq e$. Since $r:1 \prec w:1.i$, we have, $r:1 \prec w:1.i \preccurlyeq e$. Therefore, $w!q[i]$ is determined by either r or some successor s of r. In the former case, $w!q[i] = r!x.seq[i]$. As w assigns $seq[i] := q[i] \oplus 1$, this implies that $w!seq[i] \neq r!x.seq[i]$. In the latter case, $s:1 \prec w:1.i$, and therefore, $s:1 \prec w:1.i \preccurlyeq e$. Because r writes the value appearing in $RR[i, j]$ at state t and t is the state following event e, r and s are consecutive operations of Reader i. Hence, by Lemma 2, $w!q[i]$ equals $r!x.seq[i]$ or $r!x.seq[i] \oplus 1$. Therefore, $w!seq[i]$ equals $r!x.seq[i] \oplus 1$ or $r!x.seq[i] \oplus 2$. As $\oplus$ is modulo-3 addition, this implies that $w!seq[i] \neq r!x.seq[i]$. Thus, in both cases, $w!seq[i] \neq r!x.seq[i]$.

Because r writes the value appearing in $RR[i, j]$ at state t,

$$RR[i, j].flag = r!flag \wedge RR[i, j].seq = r!y.seq[i] \tag{2}$$

holds at state t. Consider the two values $r!y.seq[i]$ and $w!seq[i]$. If they are equal then, because $w!seq[i] \neq r!x.seq[i]$, we have $r!x.seq[i] \neq r!y.seq[i]$ and consequently, $r!flag$ is false. Therefore, by (2), $RR[i, j].flag$ is false at t and hence, $Cue(w, i, j)$ is false at t. If on the other hand, $r!y.seq[i] \neq w!seq[i]$ then by (2), $RR[i, j].seq \neq w!seq[i]$ at t and therefore, $Cue(w, i, j)$ is false at t. □

The following lemma ensures that the new value is not returned from a Write operation before its uncertainty interval. It states that Reader i does not cue Reader j to return the new value unless the Writer has begun the second pass of writes to the Readers. The proof makes use of Lemmas 1 and 3.

LEMMA 4. *Let w be any Write operation and $i \leq j$. Then,*

$$w \textbf{ after } 1.i \wedge w \textbf{ before } 4 \Rightarrow \neg Cue(w, i, j).$$

PROOF. We prove the lemma by induction on i. We assume the result for all indices less than i and prove it for i. Consider the state interval over which w **after** $1.i \wedge w$ **before** 4 holds. We need to show that $Cue(w, i, j)$ is false during this interval for all $j \geq i$.

Consider a state t in the interval in question and assume that $Cue(w, i, j)$ is false for all $j \geq i$ at all states in the interval that occur before t. (Note that t could be the first state in the interval.) We show that $Cue(w, i, j)$ is also false at t for $j \geq i$. If $RR[i, j].flag$ is false at t, then $Cue(w, i, j)$ is clearly false at t. So, in the remainder of the proof, assume that $RR[i, j].flag$ is true at t.

Since $RR[i, j].flag$ is false initially, there exists an operation r (of Reader i) that writes the value appearing in $RR[i, j]$ at t. Consider the events $w : 1.i$ and $r : 1$. Either $r : 1 \prec w : 1.i$ or $w : 1.i \prec r : 1$. If $r : 1 \prec w : 1.i$ then, by Lemma 3, $Cue(w, i, j)$ is false at t, as required. So, in the remainder of the proof, assume that $w : 1.i \prec r : 1$.

Let e be the event prior to state t. By the program for Reader i, $r : 1 \prec r : 3 \prec r : 5.j$. Because r writes the value appearing in $RR[i, j]$ at state t, $r : 5.j \preccurlyeq e$. Because w **before** 4 holds at t, $e \prec w : 4.1$. Thus,

$$w : 1.i \prec r : 1 \prec r : 3 \prec r : 5.j \preccurlyeq e \prec w : 4.1. \tag{3}$$

Therefore, $r!y$ is determined by w or the Write operation immediately preceding w. In the latter case, $r!y.alt \neq w!alt$, and therefore, $Cue(w, i, j)$ is false at t. In the remainder of the proof, we consider the case in which $r!y$ is determined by w.

By (3), $r : 3 \prec w : 4.i$, and therefore, $r!y.done$ is false. Hence, by the definition of p_0, $r!p_0$ is false. We now show that $(\exists l : 0 < l \leq i : r!p_l)$ is false as well. Because r writes the value appearing in $RR[i, j]$ at t, this implies that $RR[i, j].flag$ is false at t and hence $Cue(w, i, j)$ is false at t.

Consider any l in the range $0 < l \leq i$. From the program of the Writer, $w : 1.l \preccurlyeq w : 1.i$. By assumption, $w : 1.i \prec r : 1$. From the program for Reader i, $r : 1 \prec r : 2.l \prec r : 3$. Thus, by (3),

$$w : 1.l \preccurlyeq w : 1.i \prec r : 1 \prec r : 2.l \prec r : 3 \prec r : 5.j \preccurlyeq e \prec w : 4.1. \tag{4}$$

Consider the state prior to the event $r:2.l$. From the above precedence assertion, w **after** $1.l \wedge w$ **before** 4 holds at this state. If $l = i$, then $Cue(w, l, i)$ is false at this state; this follows from our assumption that $Cue(w, i, j)$ is false for all $j \geq i$ at all states that occur before t in the interval over which w **after** $1.i \wedge w$ **before** 4 holds. If $0 < l < i$, then $Cue(w, l, i)$ is false at this state by the induction hypothesis. Thus, in either case, $Cue(w, l, i)$ is false at this state. Since w determines $r!y$, by the contrapositive of Lemma 1, $r!p_l$ is false. This establishes our remaining proof obligation. □

The following lemma asserts the stability of $Cue(w, i, j)$ during the uncertainty interval. It states that once $Cue(w, i, j)$ becomes true during this interval, it remains true until the interval ends. The proof makes use of Lemma 3.

LEMMA 5. *Let w be any Write operation and let $i \leq j$. Then,*

$$w \textbf{ at } 4 \wedge Cue(w, i, j) \textbf{ unless } w \textbf{ after } 4.$$

PROOF. The stated safety property is preserved trivially by each event of the Writer and all Readers different from Reader i. We show that it is also preserved by each event of Reader i. Let s be any operation of Reader i. Consider the event $s:5.j$. This is the only event of s that may falsify the predicate $Cue(w, i, j)$. Let t be the state prior to this event and let u be the state following this event. Assume that w **at** $4 \wedge Cue(w, i, j)$ holds at t. Then, our proof obligation is to show that $Cue(w, i, j)$ holds at u (w **at** 4 holds at u trivially). By the program for Reader i, the following assertion holds at u:

$$RR[i, j].flag = s!flag \wedge RR[i, j].seq = s!y.seq[i] \wedge RR[i, j].alt = s!y.alt.$$

Thus, to prove that $Cue(w, i, j)$ holds for u, it suffices to prove the following:

$$s!flag \wedge s!y.seq[i] = w!seq[i] \wedge s!y.alt = w!alt. \tag{5}$$

Because $Cue(w, i, j)$ is false initially ($RR[i, j].flag$ is initially false), the value appearing in $RR[i, j]$ at state i is written by some operation r (of Reader i) that immediately precedes s. Consider the events $w:1.i$ and $r:1$. By the contrapositive of Lemma 3, $\neg(w$ **after** $1.i)$ holds at t or $w:1.i \prec r:1$. But, by assumption, w **at** 4 holds at t. Therefore, $w:1.i \prec r:1$.

Now, we show that w determines $r!y$. By the program for Reader i, $r:1 \prec r:3 \prec r:5.j$. Since r precedes s and w **at** 4 holds at t (i.e., the state prior to $s:5.j$), we have $r:5.j \prec s:5.j \prec w:4.M$. Thus,

$$w:1.i \prec r:1 \prec r:3 \prec r:5.j \prec s:5.j \prec w:4.M. \tag{6}$$

Therefore, $r!y$ is determined by either w or the Write operation immediately preceding w. In the latter case, $r!y.alt \neq w!alt$, and therefore, $RR[i, j].alt \neq w!alt$ at t. Consequently, $Cue(w, i, j)$ is false at t. This is contrary to our assumption. Therefore, w determines $r!y$.

From the program for the Writer, $w:0 \prec w:3.i$. Because w determines $r!y$, $w:3.i \prec r:3$. Since r precedes s, $r:3 \prec s:0 \prec s:3 \prec s:5.j$. Therefore, applying (6),

$$w:0 \prec w:3.i \prec r:3 \prec s:0 \prec s:3 \prec s:5.j \prec w:4.M. \tag{7}$$

Therefore, w determines both $s!x$ and $s!y$. The latter implies that $s!y.seq[i] = w!seq[i]$ and $s!y.alt = w!alt$. This meets two out of our three proof obligations

(eq. 5) and we are left with the proof obligation that $s!flag$ holds. This is proved next.

Because $Cue(w, i, j)$ holds at t, and r writes the value appearing in $RR[i, j]$ at state t, $r!flag$ holds. Since w determines $r!y$, this implies that $Cue(w, i, i)$ holds at the state following the event $r:5.i$. Consequently, since r and s are consecutive operations of Reader i, $Cue(w, i, i)$ also holds at the state prior to the event $s:2.i$. By definition of Cue, this implies that the following assertion holds at that state:

$$RR[i,i].flag \wedge RR[i,i].seq = w!seq[i] \wedge RR[i,i].alt = w!alt.$$

By the program for Reader i, $RR[i, i] = s!v[i]$ also holds at that state. Therefore,

$$s!v[i].flag \wedge s!v[i].seq = w!seq[i] \wedge s!v[i].alt = w!alt. \tag{8}$$

Because w determines both $s!x$ and $s!y$ (shown earlier as a consequence of (7)).

$$s!x.seq[i] = s!y.seq[i] \wedge s!x.alt = s!y.alt \wedge s!x.seq[i] = w!seq[i] \wedge s!x.alt = w!alt.$$

Therefore, using (8) and the definition of p_i, $s!p_i$ is true. Consequently, $s!flag$ holds, which was our final proof obligation. □

The following lemma considers the case in which an operation of Reader i precedes an operation of Reader j where $i \leq j$. This lemma formalizes the following property: If the y variables of both Reads are determined by the same Write operation—implying that each assigns its y variable during or after the uncertainty interval of that Write operation—and if the first read returns the new value, then the second Read also returns the new value. This lemma is based on Lemmas 1, 4, and 5, and is in turn used in the proof of precedence.

LEMMA 6. *Let r be any operation of Reader i and s be any operation of Reader j such that $i \leq j$ and r precedes s. Assume that both r!y and s!y are determined by the same Write operation. Then,*

$$(\exists k : k \leq i : r!p_k) \Rightarrow (\exists l : l \leq j : s!p_l).$$

PROOF. Assume that Write operation w determines both $r!y$ and $s!y$ and that $r!p_k$ holds for some $k \leq i$. Our proof obligation is to show that $s!p_l$ holds for some $l \leq j$.

We first establish that w determines both $s!x$ and $s!y$. Because $i \leq j$, we have $w:3.j \preccurlyeq w:3.i$. Because w determines $r!y$, $w:3.i \prec r:3$. Because r precedes s, $r:3 \prec s:0$. By the program for Reader j, $s:0 \prec s:3$. Therefore,

$$w:3.j \preccurlyeq w:3.i \prec r:3 \prec s:0 \prec s:3. \tag{9}$$

Because w determines $s!y$, by (9), w determines both $s!x$ and $s!y$.

Now, consider the events $w:4.j$ and $s:3$. Either $w:4.j \prec s:3$ or $s:3 \prec w:4.j$. We first dispose of the former case by showing that $s!p_0$ is true. Because w determines both $s!x$ and $s!y$, we have $s!x.seq[j] = s!y.seq[j]$. Moreover, since $w:4.j \prec s:3$, $s!y.done$ is true. Therefore, by the definition of p_0, $s!p_0$ is true. In the remainder of the proof, we assume that $s:3 \prec w:4.j$.

By (9), we have,

$$w:3.j \preceq w:3.i \prec r:3 \prec s:0 \prec s:3 \prec w:4.j. \tag{10}$$

By assumption, $r!p_k$ holds for some $k \leq i$. Thus, by Lemma 1, $w:1.i \prec r:1$. From the program for Reader i, $r:1 \prec r:3$. Therefore, by (10),

$$w:1.i \prec r:1 \prec r:3 \prec s:0 \prec s:3 \prec w:4.j. \tag{11}$$

Next, we show that $Cue(w, i, j)$ is true at the state prior to the event $s:2.i$. Let t be the state following the event $r:5.j$. Because $r!p_k$ holds, $r!flag$ is true. Therefore, the following assertion holds at state t:

$$RR[i,j].flag \wedge RR[i,j].seq = r!y.seq[i] \wedge RR[i,j].alt = r!y.alt.$$

Because w determines $r!y$, we have $r!y.seq[i] = w!seq[i]$ and $r!y.alt = w!alt$. Thus, the following assertion holds at state t:

$$RR[i,j].flag \wedge RR[i,j].seq = w!seq[i] \wedge RR[i,j].alt = w!alt.$$

Hence, $Cue(w, i, j)$ is true at state t. Thus, by the contrapositive of Lemma 4, $\neg(w$ **after** $1.i) \vee \neg(w$ **before** $4)$ holds at state t.

By the program for Reader i, we have $r:3 \prec r:5.j$. Since r precedes s, we have $r:5.j \prec s:0$. Thus, by (11),

$$w:1.i \prec r:1 \prec r:3 \prec r:5.j \prec s:0 \prec s:3 \prec w:4.j.$$

Therefore, w **after** $1.i$ holds at state t. Consequently, $\neg(w$ **before** $4)$ holds at t, that is, w **at** 4 or w **after** 4 holds at t. But, by the above precedence assertion w **after** 4 does not hold at t. Thus, w **at** 4 holds at t, that is, $w:3.1 \prec r:5.j$. Therefore,

$$w:3.1 \prec r:5.j \prec s:0 \prec s:3 \prec w:4.j.$$

Observe that w **at** 4 holds for all states between $r:5.j$ and $s:3$. Thus, by Lemma 5, $Cue(w, i, j)$ holds for all states in that interval. In particular, it holds at the state prior to $s:2.i$.

Now, by the program for Reader j, $RR[i, j] = s!v[i]$ holds at the state prior to the event $s:2.i$. This implies that the following assertion holds:

$$s!v[i].flag \wedge s!v[i].seq = w!seq[i] \wedge s!v[i].alt = w!alt.$$

On account of (10) and the fact that $s!x$ and $s!y$ are both determined by w, $s!x = s!y$, $s!x.seq[i] = w!seq[i]$, and $s!x.alt = w!alt$. Therefore,

$$\begin{aligned} s!x.seq[j] = s!y.seq[j] \wedge s!x.seq[i] = s!y.seq[i] \wedge s!x.alt = s!y.alt \\ \wedge s!v[i].flag \wedge s!v[i].seq = s!x.seq[i] \\ \wedge s!v[i].alt = s!x.alt. \end{aligned}$$

Hence, by the definition of p_i, $s!p_i$ is true, which is our proof obligation. □

Because the Writer writes to higher numbered Readers first, a lemma is needed that ensures that new-then-old conflicts do not arise in a history in which an operation of a higher numbered Reader is followed by an operation of a lower numbered one. The required property is given later in Lemma 9. Lemmas 7 and 8, given next, take care of subcases arising in the proof of Lemma 9. Lemma 7 states that if Reader i has cued Reader j of the new value

during the uncertainty interval of Write operation w, then it did so either on account of reading the final value from the Writer (i.e., w **after** $4.i$ holds) or on account of being cued in turn by some other Reader (i.e., $(\exists k : k < i : Cue(w, k, i))$ holds.) The proof of this lemma makes use of Lemmas 1, 3, 4, and 5.

LEMMA 7. *Let w be any Write operation and $i \leq j$. Then,*

$$w \textbf{ at } 4 \wedge Cue(w, i, j) \Rightarrow w \textbf{ after } 4.i \vee (\exists k : k < i : Cue(w, k, i)).$$

PROOF. Consider any Reader i and the state interval over which w **at** 4 holds. We need to show that the property

$$(\forall j : j \geq i : Cue(w, i, j) \Rightarrow w \textbf{ after } 4.i \vee (\exists k : k < i : Cue(w, k, i))) \quad (12)$$

holds at all states during this interval. Consider a state t in the interval in question and assume that property (12) holds at all states in the interval that occur before t. (Note that t could be the first state in the interval.) We show that the property also holds at state t.

Assume that $Cue(w, i, j) \wedge \neg(w \textbf{ after } 4.i)$ holds at state t for some $j \geq i$. Our proof obligation is to show that $Cue(w, k, i)$ holds at state t for some $k < i$.

Since $RR[i, j].flag$ is false in the initial state, $Cue(w, i, j)$ is initially false. Therefore, assume that the value appearing in $RR[i, j]$ at state t is written by operation r (of Reader i). Consider the events $w : 1.i$ and $r : 1$. By the contrapositive of Lemma 3, $\neg(w \textbf{ after } 1.i)$ holds at t or $w : 1.i \prec r : 1$. But, by assumption, w **at** 4 holds at t. Therefore, $w : 1.i \prec r : 1$.

By the program for Reader i, $r : 1 \prec r : 5.j$. Because r writes the value appearing in $RR[i, j]$ at state t, $r : 5.j \preccurlyeq e$, where e is the event prior to state t. Because $\neg(w \textbf{ after } 4.i)$ holds at t, $e \prec w : 4.i$. Therefore,

$$w : 1.i \prec r : 1 \prec r : 5.j \preccurlyeq e \prec w : 4.i. \quad (13)$$

Therefore, $r!y$ is determined by either w or the immediate precedessor of w —but, the immediate predecessor of w does not determine $r!y$, as $r!y.alt = w!alt$ on account of $Cue(w, i, j)$ being true at state t. So, w determines $r!y$.

Because $Cue(w, i, j)$ holds at state t, $RR[i, j].flag$ also holds at t. Hence, because r writes the value appearing in $RR[i, j]$ at t, $r!flag$ holds. By the program for Reader i, this implies that $r!p_k$ is true for some k where $k \leq i$. We now show that $k > 0$. Observe that assertion (13) implies that $r : 3 \prec e \prec w : 4.i$. Thus, because w determines $r!y$, $r!y.done$ is false, which implies that $r!p_0$ is false. Thus, $k > 0$.

Because $k \leq i$, we have $w : 1.k \preccurlyeq w : 1.i$. By the program for Reader i, $r : 1 \prec r : 2.k \prec r : 5.j$. Thus, by (13), we have

$$w : 1.k \preccurlyeq w : 1.i \prec r : 1 \prec r : 2.k \prec r : 5.j \preccurlyeq e \prec w : 4.i. \quad (14)$$

Let u denote the state prior to the event $r : 2.k$. Because $r!p_k$ holds and because $k > 0$, by Lemma 1, $Cue(w, k, i)$ holds at state u. Thus, by the contrapositive of Lemma 4, $\neg(w \textbf{ after } 1.k) \vee \neg(w \textbf{ before } 4)$ holds at u. But, by assertion (14), w **after** $1.k$ holds at u. Consequently, $\neg(w \textbf{ before } 4)$ holds at u, that is, w **at** 4 or w **after** 4 holds at u. But, by (14), w **after** 4 does not hold at u. Thus, w **at** 4 holds at u, which implies that $w : 3.1 \prec r : 2.k$. Thus, by

assertion (14), we have,

$$w:3.1 \prec r:2.k \prec r:5.j \preccurlyeq e \prec w:4.i. \tag{15}$$

Observe that w **at** 4 holds for all states between $r:2.k$ and $w:4.i$. Since $Cue(w, k, i)$ holds at state u, by Lemma 5, $Cue(w, k, i)$ holds for all states in this interval. In particular, it holds, at state t (the state following event e). This establishes our proof obligation if $k < i$.

In the case that $k = i$, $Cue(w, i, i)$ holds at state u. From (15), u lies within the interval over which w **at** 4 holds and u occurs before t. Consequently, from our assumption that property (12) holds at all states prior to t in this interval, w **after** $4.i$ holds at state u or $Cue(w, k, i)$ holds at state u, for some $k < i$. However, by (15), w **after** $4.i$ does not hold at state u. Therefore, $Cue(w, k, i)$ holds at state u, for some $k < i$. Consequently, by applying Lemma 5 as in the previous paragraph, $Cue(w, k, i)$ also holds at state t, as desired. □

According to the next lemma, if Reader i has cued Reader j during the uncertainty interval of Write operation w, then it has also cued other Readers with indices between i and j. The proof follows essentially from the fact that Reader i writes to other Readers in the order of increasing indices. This lemma makes use of Lemmas 3, 4, and 5 and is in turn used in the proof of Lemma 9.

LEMMA 8. *Let w be any Write operation and $i \le j$. Then,*

$$w \textbf{ at } 4 \wedge Cue(w, i, j) \Rightarrow (\forall k : i \le k \le j : Cue(w, i, k)).$$

PROOF. Assume that w **at** $4 \wedge Cue(w, i, j)$ holds at some state t. Let $i \le k < j$. We show that $Cue(w, i, k)$ holds at state t.

Because $RR[i, j].flag$ is false in the initial state, $Cue(w, i, j)$ is initially false. Therefore, assume that the value appearing in $RR[i, j]$ at state t is written by operation r (of Reader i).

Let u be the state following the event $r:5.k$. Since $Cue(w, i, j)$ holds at t and since r writes identical values to $RR[i, j]$ and $RR[i, k]$, $Cue(w, i, k)$ holds at u.

Now, consider the events $w:1.i$ and $r:1$. By contrapositive of Lemma 3, $\neg(w$ **after** $1.i)$ holds at t or $w:1.i \prec r:1$. But, by assumption, w **at** 4 holds at t. Therefore, $w:1.i \prec r:1$. By the program for Reader i, $r:1 \prec r:5.k \prec r:5.j$. Because r writes the value appearing in $RR[i, j]$ at state t, $r:5.j \preccurlyeq e$, where e denotes the event prior to state t. Since w **at** 4 holds at t, $e \prec w:4.M$. Therefore,

$$w:1.i \prec r:5.k \prec e \prec w:4.M. \tag{16}$$

Since w **after** $1.i \wedge Cue(w, i, k)$ holds at state u, by Lemma 4, $\neg\, w$ **before** 4 holds at u. In other words, w **at** 4 or w **after** 4 holds at u. But, by (16), w **after** 4 does not hold at state u. Thus, w **at** 4 holds at u. Consequently, by Lemma 5, $Cue(w, i, k)$ holds at all states in the interval between $r:5.k$ and $w:4.M$ and in particular at state t. This establishes our proof obligation. □

Lemma 9 relates the value of $RR[l, i]$ to that of $RR[k, j]$ where $j < i$. It states that if Reader l has cued Reader i during the uncertainty interval of Write operation w, then all Readers with indices lower than i either have been informed of the new value by the Writer (i.e., w **after** $4.j$ holds) or they in turn

have been cued by some other Reader (i.e., $(\exists k : k \leq j : Cue(w, k, j))$ holds). The proof of this lemma makes use of Lemmas 7 and 8.

LEMMA 9. *Let w be any Write operation and $l \leq i$. Then,*

$$w \textbf{ at } 4 \wedge Cue(w, l, i) \Rightarrow (\forall j : j \leq i : w \textbf{ after } 4.j \vee (\exists k : k \leq j : Cue(w, k, j))).$$

PROOF. Assume that w **at** $4 \wedge Cue(w, l, i)$ holds at some state t. For brevity, let $P(j)$ be shorthand for $(\exists k : k \leq j : Cue(w, k, j))$. Then, we have to show that $(\forall j : j \leq i : w$ **after** $4.j \vee P(j))$ holds at state t. The proof is by induction. The inductive step is given by the following assertion, which, as shown later, holds at state t.

$$w \textbf{ at } 4 \wedge Cue(w, l, i) \Rightarrow (w \textbf{ after } 4.l \wedge (\forall j : l \leq j \leq i : P(j))) \vee$$
$$(\exists l_1, i_1 : l_1 \leq i_1 < l \leq i : w \textbf{ at } 4 \wedge Cue(w, l_1, i_1) \wedge (\forall j : i_1 \leq j \leq i : P(j))).$$

Given our assumption that w **at** $4 \wedge Cue(w, l, i)$ holds at t, this inductive step implies that one of the disjuncts of the consequent holds at t. If the second disjunct holds, then the inductive step can be applied again, this time using w **at** $4 \wedge Cue(w, l_1, i_1)$ as the antecedent. Because the number of Readers is finite and because the Reader indices (l_1, i_1) appearing in the second disjunct of the consequent are smaller than the indices (l, i), this inductive step can be applied in this way only a finite number of times. Thus, after some finite number of applications, we eventually obtain a consequent in which the first disjunct holds. Since the assertions $P(j)$ are being accumulated for $i_1 \leq j \leq i$ in the second disjunct of each application of the inductive step, this implies that following assertion holds when the induction terminates.

$$(\exists m : m \leq l : w \textbf{ after } 4.m \wedge (\forall j : m \leq j \leq i : P(j))).$$

Because $j \leq m \wedge w$ **after** $4.m \Rightarrow w$ **after** $4.j$, we can assert the following at state t:

$$(\exists m : m \leq l : (\forall j : j \leq m : w \textbf{ after } 4.j) \wedge (\forall j : m \leq j \leq i : P(j))).$$

Combining the ranges of the two universal quantifications, we get

$$(\exists m : m \leq l : (\forall j : j \leq i : w \textbf{ after } 4.j \vee P(j))).$$

at state t. This implies that

$$(\forall j : j \leq i : w \textbf{ after } 4.j \vee P(j))$$

holds at state t, which is the required proof obligation. The proof of the inductive step is as follows (All state predicates refer to state t.):

w **at** $4 \wedge Cue(w, l, i)$
$\Rightarrow$ {Lemma 7}
w **after** $4.l \vee (\exists i_1 : i_1 < l : Cue(w, i_1, l))$
$\Rightarrow$ {by assumption, w **at** 4}
w **after** $4.l \vee (\exists i_1 : i_1 < l : w \textbf{ at } 4 \wedge Cue(w, i_1, l))$
$\Rightarrow$ {Lemma 8}
w **after** $4.l \vee (\exists i_1 : i_1 < l : (\forall j : i_1 \leq j \leq l : Cue(w, i_1, j)))$
$\Rightarrow$ {definition of $P(j)$}
w **after** $4.l \vee (\exists i_1 : i_1 < l : (\forall j : i_1 \leq j \leq l : P(j)))$

$\Rightarrow$ {by assumption, w **at** 4 $\wedge$ $Cue(w, l, i)$}
(w **after** 4.$l \vee (\exists i_1 : i_1 < l : (\forall j : i_1 \le j \le l : P(j)))) \wedge w$ **at** 4 $\wedge$ $Cue(w, l, i)$
$\Rightarrow$ {Lemma 8}
(w **after** 4.$l \vee (\exists i_1 : i_1 < l : (\forall j : i_1 \le j \le l : P(j)))) \wedge$
$(\forall j : l \le j \le i : Cue(w, l, j))$
$\Rightarrow$ {definition of $P(j)$}
(w **after** 4.$l \vee (\exists i_1 : i_1 < l : (\forall j : i_1 \le j \le l : P(j)))) \wedge (\forall j : l \le j \le i : P(j))$
$\Rightarrow$ {predicate calculus}
(w **after** 4.$l \wedge (\forall j : l \le j \le i : P(j))) \vee (\exists i_1 : i_1 < l : (\forall j : i_1 \le j \le i : P(j)))$
$\Rightarrow$ {predicate calculus and $l \le i$}
(w **after** 4.$l \wedge (\forall j : l \le j \le i : P(j))) \vee$
$(\exists i_1 : i_1 < l \le i : P(i_1)) \wedge (\forall j : i_1 \le j \le i : P(j)))$
$\Rightarrow$ {definition of $P(i_1)$}
(w **after** 4.$l \wedge (\forall j : l \le j \le i : P(j))) \vee$
$(\exists l_1, i_1 : l_1 \le i_1 < l \le i : Cue(w, l_1, i_1) \wedge (\forall j : i_1 \le j \le i : P(j)))$
$\Rightarrow$ {by assumption, w **at** 4}
(w **after** 4.$l \wedge (\forall j : l \le j \le i : P(j))) \vee$
$(\exists l_1, i_1 : l_1 \le i_1 < l \le i : w$ **at** 4 $\wedge$ $Cue(w, l_1, i_1) \wedge (\forall j : i_1 \le j \le i : P(j)))$
□

The following lemma considers two successive Read operations r and s such that r, though it precedes s, reads from a more recent Write operation than s. Because of the order in which the Writer writes to the Readers, this situation can arise only if the index of r is greater than that of s. The lemma ensures that r computes its *flag* to be false and that s computes its *flag* to be true. This lemma is based on Lemmas 1 and 4, and is in turn used in the proof of precedence.

LEMMA 10. *Let r be any operation of Reader i and s be any operation of Reader j such that $i > j$ and r precedes s. Assume that $r!y$ and $s!y$ are determined by Write operations $W : m$ and $W : (m - 1)$, respectively. Then, $r!flag$ is false and $s!flag$ is true.*

PROOF. Let w denote $W : (m - 1)$ and w' denote $W : m$. Because w' determines $r!y, w' : 3.i \prec r : 3$. Because r precedes s, $r : 3 \prec s : 0$. By the program for Reader j, $s : 0 \prec s : 3$. Because w determines $s!y$ and w precedes w', $s : 3 \prec w' : 3.j$. Therefore,

$$w' : 3.i \prec r : 3 \prec s : 0 \prec s : 3 \prec w' : 3.j. \tag{17}$$

Hence, $r!y.done$ is false, $s!y.done$ is true, and $s!x.seq[j] = s!y.seq[j]$. Thus, by the definition of p_0, $r!p_0$ is false and $s!p_0$ is true. Because $s!p_0$ is true, from the program of Reader j, $s!flag$ is true. This meets a half of our proof obligation. In the remainder of the proof, we show that $r!p_k$ is false for each k in the range $0 < k \le i$. This implies our remaining proof obligation, namely that $r!flag$ is false.

Consider the two events $r : 1$ and $w' : 1.i$. Either $r : 1 \prec w' : 1.i$ or $w' : 1.i \prec r : 1$. In the former case, by the contrapositive of Lemma 1, $r!p_k$ is false for each k as desired. So, assume that $w' : 1.i \prec r : 1$ in the remainder of the proof.

Because $k \le i$, we have $w' : 1.k \preccurlyeq w' : 1.i$. By the program for Reader i, $r : 1 \prec r : 2.k \prec r : 3$. Thus, using (17), we have

$$w' : 1.k \preccurlyeq w' : 1.i \prec r : 1 \prec r : 2.k \prec r : 3 \prec s : 0 \prec s : 3 \prec w' : 3.j.$$

Note that w' **after** $1.k \wedge w'$ **before** 4 holds at the state prior to $r:2.k$. Hence, from Lemma 4, $\neg\, Cue(w', k, i)$ holds at that state. Therefore, by the contrapositive of Lemma 1, $r!p_k$ is false, which is the required proof obligation. □

The final lemma in the proof of the construction considers the case in which an operation of Reader i precedes an operation of Reader j where $i > j$. This lemma is the counterpart of Lemma 6, which considers the case $i \leq j$. This lemma is based on Lemmas 1, 4, 5, and 9, and is in turn used in the proof of precedence.

LEMMA 11. *Let r be any operation of Reader i and s be any operation of Reader j such that $i > j$ and r precedes s. Assume that both $r!y$ and $s!y$ are determined by the same Write operation. Then,*

$$(\exists k : k \leq i : r!p_k) \Rightarrow (\exists l : l \leq j : s!p_l).$$

PROOF. Assume that Write operation w determines both $r!y$ and $s!y$ and that $r!p_k$ holds for some $k \leq i$. Our proof obligation is to show that $s!p_l$ holds for some $l \leq j$.

Consider the two events $w:4.i$ and $r:3$. Either $w:4.i \prec r:3$ or $r:3 \prec w:4.i$. We first dispose of the former case. Because $j < i$, we have $w:4.j \prec w:4.i$. Because r precedes s, $r:3 \prec s:0$. By the program for Reader j, $s:0 \prec s:3$. Therefore,

$$w:4.j \prec w:4.i \prec r:3 \prec s:0 \prec s:3. \tag{18}$$

Thus, since w determines $s!y$, $s!y.done$ is true and $s!x.seq[j] = s!y.seq[j]$. Therefore, by the definition of p_0, $s!p_0$ is true, which establishes our proof obligation.

In the remainder of the proof, assume that $r:3 \prec w:4.i$. Because $r!y$ is determined by w, this implies that $r!y.done$ is false. By the definition of p_0, this implies that $r!p_0$ is false; therefore, $k > 0$. Because $r!p_k$ holds, by Lemma 1, $w:1.i \prec r:1$. By the program for Reader i, $r:1 \prec r:2.k \prec r:3$. Therefore,

$$w:1.i \prec r:1 \prec r:2.k \prec r:3 \prec w:4.i.$$

Let t denote the state prior to the event $r:2.k$. We now show that w **at** 4 holds at t. Because $r!p_k$ is true and because $k > 0$, by Lemma 1, $Cue(w, k, i)$ holds at state t. Hence, by the contrapositive of Lemma 4, $\neg(w$ **after** $1.k) \vee \neg(w$ **before** $4)$ holds at t. Because $k \leq i$, we have $w:1.k \preccurlyeq w:1.i$. Thus, by the previous precedence assertion, we have the following.

$$w:1.k \preccurlyeq w:1.i \prec r:1 \prec r:2.k \prec r:3 \prec w:4.i.$$

Therefore, w **after** $1.k$ holds at state t. Consequently, $\neg(w$ **before** $4)$ holds at t, that is, w **at** 4 or w **after** 4 holds at t. But, by the above precedence assertion, w **after** 4 does not hold at t. Thus, w **at** 4 holds at t.

Since w **at** $4 \wedge Cue(w, k, i)$ holds at state t and $j < i$, by Lemma 9, w **after** $4.j$ holds at t or $Cue(w, m, j)$ holds at t for some $m \leq j$. In the former case, we have $w:4.j \prec r:2.k$. Because r precedes s, this implies that $w:4.j \prec s:0 \prec s:3$. By repeating the reasoning following (18), this implies that $s!p_0$ holds, as required. In the remainder of the proof, assume that $Cue(w, m, j)$ holds at t.

Because $i > j \geq 1$, we have $w:3.i \prec w:3.j \preccurlyeq w:3.1$. Since w **at** 4 holds at state t, we have $w:3.1 \prec r:2.k$. By the program for Reader i, $r:2.k \prec r:3$.

Therefore,

$$w:3.i \prec w:3.j \preccurlyeq w:3.1 \prec r:2.k \prec r:3. \tag{19}$$

Consider the two events $w:4.j$ and $s:3$. Either $w:4.j \prec s:3$ or $s:3 \prec w:4.j$. We first dispose of the former case. Because r precedes s, $r:3 \prec s:0$. Hence, by (19), we have $w:3.j \prec s:0$. Since w determines $s!y$ and $w:4.j \prec s:3$, $s!y.done$ is true. Moreover, since $w:3.j \prec s:0$, $s!x.seq[j] = s!y.seq[j]$. Therefore, by the definition of p_0, $s!p_0$ is true, which establishes our proof obligation.

In the remainder of the proof, assume that $s:3 \prec w:4.j$. Because r precedes s, $r:3 \prec s:0 \prec s:3$. Thus, by (19), the following assertion holds.

$$w:3.i \prec w:3.j \preccurlyeq w:3.1 \prec r:2.k \prec r:3 \prec s:0 \prec s:3 \prec w:4.j. \tag{20}$$

Therefore, $s!x = s!y$.

By assertion (20), w **at** 4 holds for all states in the interval between $r:2.k$ and $s:3$. Recall that $Cue(w, m, j)$ holds at state t, that is, the state prior to the event $r:2.k$. Therefore, by Lemma 5, $Cue(w, m, j)$ holds for all states in this interval. In particular, it holds at the state following the event $s:2.m$. By the program for Reader j, $RR[m, j] = s!v[m]$ also holds at that state. Therefore,

$$s!v[m].flag \wedge s!v[m].seq = w!seq[m] \wedge s!v[m].alt = w!alt.$$

Because w determines $s!y$, $s!y.seq[m] = w!seq[m]$ and $s!y.alt = w!alt$. Since $s!x = s!y$, we have $s!x.seq[j] = s!y.seq[j]$, $s!x.seq[m] = s!y.seq[m]$, and $s!x.alt = s!y.alt$. Therefore,

$$\begin{aligned} s!x.seq[j] = s!y.seq[j] \wedge s!x.seq[m] = s!y.seq[m] \wedge s!x.alt = s!y.alt \\ \wedge s!v[m].flag \wedge s!x.seq[m] = s!v[m].seq \\ \wedge s!x.alt = s!v[m].alt. \end{aligned}$$

Consequently, by the definition of p_m, $s!p_m$ is true, which is our proof obligation. □

ACKNOWLEDGMENTS. We would like to thank Ted Herman and the UT Distributed Systems Discussion Group for their comments on this paper. We would also like to thank Nancy Lynch and the anonymous referees for their helpful comments.

REFERENCES

ANDERSON, J., AND GOUDA, M. 1990. The virtue of patience: Concurrent programming with and without waiting, Tech. Rep. TR.90.23. Dept. Comput. Sci., Univ. Texas at Austin, Austin, Tex., July.

ANDERSON, J., SINGH, A., AND GOUDA, M. 1986. The elusive atomic register, Tech. Rep. TR.86.29. Dept. Comput. Sci., Univ. Texas at Austin, Austin, Tex.

BLOOM, B. 1988. Constructing two-writer atomic registers. *IEEE Trans. Comput. 37*, 12 (Dec.) 1506–1514.

BURNS, J. E., AND PETERSON, G. L. 1987. Constructing multi-reader atomic values from non-atomic values. In *Proceedings of the 6th Annual ACM Symposium on Principle of Distributed Computing*, (Vancouver, B.C., Canada, Aug. 10–12). ACM, New York, 1987, pp. 222–231.

CHANDY, K., AND MISRA, J. 1988. *Parallel Program Design:A Foundation, Addison Wesley.*

COURTOIS, P. J., HEYMANS, F., AND PARNAS, D. L. 1971. Concurrent control with readers and writers *Commun. ACM 14*, 10 (Oct.), 667–668.

HALDAR, S., AND VIDYASANKAR, K. 1991. Counterexamples to a one writer multireader atomic variable construction of Burns and Peterson. Tech. Rep. #9106. Dept. Comput. Sci. Memorial Univ. Newfoundland, St. John's, Canada, July.

HERLIHY, M., AND WING, J. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Prog. Lang. Syst., 12*, 3 (July), 463–492.

ISRAELI, A., AND LI, M. 1987. Bounded time-stamps. In *Proceedings of the 28th IEEE Symposium on Foundations of Computer Science*. IEEE, New York, pp. 371–382.

KIROUSIS, L., KRANAKIS, E., AND VITANYI, P. 1987. Atomic multireader register. In *Proceedings of the 2nd International Workshop on Distributed Computing*. Lecture Notes in Computer Science, vol. 312, Springer-Verlag, New York, pp. 278–296.

LAMPORT, L. 1980. The "Hoare logic" of concurrent programs. *Acta Inf., 14*, 1, 21–37.

LAMPORT, L. 1986. On interprocess communication, parts I and II. *Dist. Comput. 1*, 77–101.

LI, M., TROMP, J., AND VITANYI, P. 1989. How to construct wait-free variables. In *Proceedings of International Colloquium on Automata, Languages, and Programming*. Lecture Notes in Computer Science, vol. 372. Springer-Verlag, New York, pp. 488–505.

MISRA, J. 1986. Axioms for memory access in asynchronous hardware systems, *ACM Trans. Prog. Lang. Syst., 8*, 1 (Jan.), 142–153.

NEWMAN-WOLF, R. 1987. A protocol for wait-free, atomic, multi-reader shared variables. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing* (Vancouver, B.C., Canada, Aug. 10–12). ACM, New York, pp. 232–248.

PETERSON, G. L. 1983. Concurrent reading while writing. *ACM Trans. Program. Lang. Syst. 5*, (Jan.) 46–55.

PETERSON, G., AND BURNS, J. 1987. Concurrent reading while writing II: The multi-writer case. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*. IEEE, New York, 383–392.

SCHAFFER, R. 1988. On the correctness of atomic multi-writer registers. Tech. Rep. MIT/LCS/TM-364. MIT Laboratory for Computer Science, June.

SINGH, A. K., ANDERSON, J. H., AND GOUDA, M. G. 1987. The elusive atomic register revisited. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing* (Vancouver, B.C., Canada, Aug. 10–12). ACM, New York, pp. 206–221.

TROMP, J. 1989. How to construct an atomic variable. In *Proceedings of the 3rd International Workshop on Distributed Algorithms*. Lecture Notes in Computer Science, vol. 392. Springer-Verlag, New York, pp. 292–302.

VIDYASANKAR, K. 1988. Converting Lamport's regular registers to atomic registers. *Inf. Proc. Lett., 28*, 6, 287–290.

VIDYASANKAR, K. 1990. Concurrent reading while writing revisted. *J. Dist. Comput. 4*, 81–85.

VITANYI, P., AND AWERBUCH, B. 1986. Atomic shared register access by asynchronous hardware. In *Proceedings of the 27th Annual Symposium on Foundations of Comupter Science*. IEEE, New York, pp. 233–243.

RECEIVED JUNE 1987; REVISED JUNE 1992; ACCEPTED JANUARY 1993