Multi-Writer Composite Registers*

James H. Anderson†

Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, NC 27599-3175

July 1989
Revised March 1991, December 1992, August 1993

Abstract

A *composite register* is an array-like shared data object that is partitioned into a number of components. An operation of such a register either writes a value to a single component, or reads the values of all components. A composite register reduces to an ordinary atomic register when there is only one component. In this paper, we show that a composite register with multiple writers per component can be implemented in a wait-free manner from a composite register with a single writer per component. It has been previously shown that registers of the latter kind can be implemented from atomic registers without waiting. Thus, our results establish that any composite register can be implemented in a wait-free manner from atomic registers. We show that our construction has comparable space complexity and better time complexity than other constructions that have been presented in the literature.

Keywords: atomicity, atomic register, composite register, concurrency, interleaving semantics, linearizability, shared variable, snapshot

CR Categories: D.4.1, D.4.2, F.3.1

*Preliminary version was presented at the Ninth Annual ACM Symposium on Principles of Distributed Computing [2].

†Much of the work described herein was completed while the author was with the University of Texas at Austin and the University of Maryland at College Park. This work was supported at the University of Texas by ONR Contract N00014-89-J-1913, and at the University of Maryland by NSF Contract CCR 9109497 and by an award from the University of Maryland General Research Board.

1 Introduction

The wait-free implementation of concurrent shared data objects is a subject that has received much attention in the concurrent programming literature. A *shared data object* is a data structure that is accessed by a collection of processes by means of a fixed set of operations. An implementation of a shared data object is *wait-free* iff the operations of the data object are implemented without any unbounded busy-waiting loops or idle-waiting primitives. Wait-free implementations are preferable to those that employ mutually exclusive "critical sections" because they are inherently resilient to halting failures. In particular, a process that halts while accessing a wait-free shared data object cannot prevent subsequent accesses by other processes. Also, because such an object can be accessed concurrently by any number of the processes that share it, wait-free implementations allow processes to execute with maximum parallelism.

The notion of an atomic register is of fundamental importance in the study of wait-free shared data objects [24, 25, 28, 30]. An *atomic register* is a shared data object that can either be read or written (but not both) in a single operation. An atomic register can be characterized by the number of processes that can read or write it, and the number of bits that it stores. The simplest atomic register can be read by one process, can be written by one process, and can store a one-bit value; the most complicated can be read or written by several processes and can store any number of bits. It has been shown in a series of papers that the most complicated atomic register can be implemented without waiting in terms of the simplest [6, 13, 14, 20, 21, 25, 26, 29, 30, 31, 32, 33, 34]. This work shows that, using only atomic registers of the simplest kind, the classical readers-writers problem [17] can be solved without requiring either readers or writers to wait.

In this paper, we consider a shared data object, called a *composite register*, that extends the notion of an atomic register. The notion of a composite register was first introduced by Anderson [2, 3, 4], and is similar to the atomic snapshot memory of Afek et al. [1]. A composite register is an array-like shared data object that is partitioned into a number of components. An operation of a composite register either writes a value to a single component, or reads the values of all components. Note that a composite register differs significantly from an atomic register: a write operation of a composite register only overwrites a portion of the register, namely the contents of a particular component, while leaving the rest of the register unchanged. By contrast, a write operation of an atomic register overwrites the previous contents of the entire register.

In this paper, we show that composite registers can be implemented from atomic registers without waiting. Specifically, we show that a composite register with multiple writers per component — hereafter called a *multi-writer* composite register — can be constructed in a wait-free manner from a composite register with one writer per component — hereafter called a *single-writer* composite register. As explained below, wait-free constructions of single-writer composite registers from atomic registers have been given previously by several authors. Along with these previous constructions, the results of this paper show that multi-writer composite registers can be implemented in a wait-free manner using only atomic registers. It follows from this result that atomic registers can be used to implement a shared memory that can be read in its entirety in a single "snapshot" operation, without using mutual exclusion.

The problem of constructing a composite register from atomic registers was first considered by us in [2, 3, 4] and by Afek et al. in [1]. The construction given in this paper is based on the multi-writer construction of [2, 4]. More recently, several constructions have been presented by Kirousis et al. [22, 23], for the special case in which there is only one reader.

Our approach in constructing a composite register from atomic registers differs from that of Afek et al. in several respects. Afek et al. presented two constructions, one that implements a single-writer composite register from multi-reader, single-writer atomic registers, and another that implements a multi-writer composite register from multi-reader, multi-writer atomic registers. We have also given constructions for both the single- and multi-writer cases. Like that of Afek et al., our single-writer composite register construction, which is given in [2, 3], is based on multi-reader, single-writer atomic registers. However, our multi-writer construction differs from theirs in that it is based on a single-writer composite register. By employing the single-writer constructions previously mentioned, our multi-writer construction shows that a multi-writer composite register can be implemented using only single-writer atomic registers. As such, our construction solves the problem of implementing a multi-writer atomic register (the case in which there is only one component) from single-writer ones.

The construction of this paper and the multi-writer construction of Afek et al. also differ in complexity. It is assumed in [1] that the constructed composite register is shared by N processes, and that each process may read the register or write any component. Under these assumptions, our multi-writer construction has comparable space complexity and better time complexity than the multi-writer construction of Afek et al. In fact, the time complexity of our construction is (asymptotically) the same as the single-writer composite register used in the construction. Thus, because the time complexity of the single-writer construction in [1] is $\Theta(N^2)$, our construction shows that multi-writer composite registers can also be implemented with time complexity that is $\Theta(N^2)$.[1] The multi-writer construction of Afek et al. has time complexity that is $\Theta(N^3)$.

Composite registers are quite powerful and can be used to implement a number of interesting shared data objects without waiting. For example, as shown in [7, 8, 10], composite registers can be used to implement wait-free shared data objects with "pseudo" read-modify-write (PRMW) operations. A PRMW operation is similar to a "true" read-modify-write (RMW) operation in that it modifies the value of a shared variable[2] based upon the original value of that variable. However, unlike RMW operations, a PRMW operation does not return the value of the variable that it modifies. An operation that increments a shared variable without returning its value is an example of a PRMW operation. It is shown in [7, 8, 10] that composite registers can be used to implement without waiting any shared data object that can either be read, written, or modified by a commutative PRMW operation. As explained in Section 5, these results have been recently extended in [9], where it is shown that composite registers can be used to implement an even larger class of objects. Such results stand in sharp contrast to those of [5, 18], where it is shown that RMW operations cannot, in general, be implemented from atomic registers without waiting.

The rest of the paper is organized as follows. In Section 2, we formally define the problem of constructing a composite register of one type from a composite register of a simpler type, and in Section 3 we present a

[1]At about the same time as this paper was accepted for publication, a $\Theta(N \log N)$ single-writer construction was presented by Attiya and Rachman in [11]. By using their construction as the basis for ours, we get a multi-writer construction with time complexity $\Theta(N \log N)$. If other, more efficient single-writer constructions are developed, then our construction can be used to obtain a corresponding improvement for the multi-writer case.

[2]The term *variable* is used to denote an arbitrary data item. The term *register* is used when referring to a particular shared data object, such as an atomic register or composite register.

sufficient condition for proving that a construction is correct. Then, in Section 4, we present our construction along with its proof of correctness. Concluding remarks appear in Section 5.

2 Composite Register Construction

In this section, we consider the problem of constructing a composite register of one type from composite registers of a simpler type, and give the conditions that such a construction must satisfy to be correct. (The simpler composite register used in such a construction could, for instance, have fewer components, fewer readers, etc.)

A construction consists of a set of procedures along with a set of variables. Each procedure has the following form:

```
procedure name(inputs)
private var ...
begin
    body;
    return(outputs)
end
```

where *name* is the name of the procedure, either *Reader* or *Writer*, *inputs* is an optional list of input parameters, *outputs* is an optional list of output parameters, and *body* is a program fragment comprised of atomic statements. One may think of each procedure as being resident to a particular process. A Reader procedure is invoked by a process to read the values of all components of the constructed register, and a Writer procedure is invoked by a process to write a value to a particular component of the register. Each Reader procedure has one output parameter for each component of the constructed register, and each Writer procedure has an input parameter indicating the value to be written. We assume that each process invokes its resident procedures in a serial manner.

For convenience, we designate a composite register construction by a 4-tuple $C/B/W/R$, where C is the number of components, B is the number of bits per component, W is the number of Writers per component, and R is the number of Readers. (Thus, a $1/B/W/R$ composite register is an ordinary atomic register.) The structure of a $C/B/W/R$ composite register construction is depicted in Figure 1. Note that this figure only depicts the Writer procedures for component i. For an example of a Reader or Writer procedure, see Figure 3.

Each variable of a construction is either private or shared. A *private variable* is defined only within the scope of a single procedure, whereas a *shared variable* is defined globally and may be accessed within more than one procedure. (Each procedure's program counter is considered to be a private variable.) A construction is required to satisfy the following two restrictions.

- *Atomicity Restriction:* Each shared variable is required to be of the same type as the simpler composite register used in the construction. Note that this restriction constrains those statements that access shared variables.

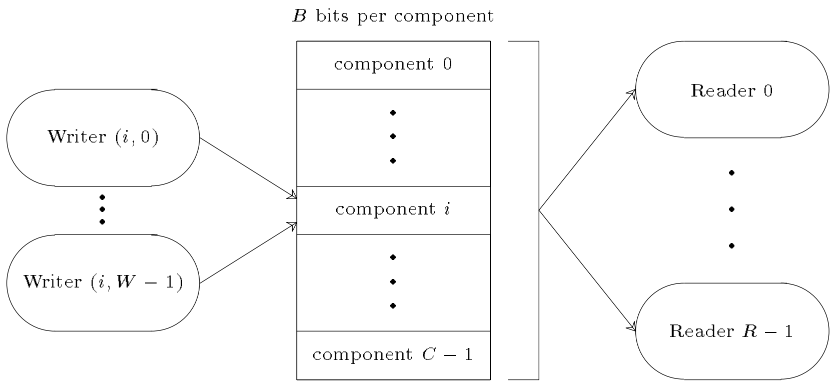

Figure 1: $C/B/W/R$ composite register structure.

- *Wait-Freedom Restriction*: As mentioned in the introduction, each procedure is required to be "wait-free," i.e., idle-waiting primitives and unbounded busy-waiting loops are not allowed. (A more formal definition of wait-freedom is given in [5].)

We now define several concepts that are needed to state the correctness condition for a construction. These definitions apply to a given construction. A *state* is an assignment of values to all variables (private and shared) of the construction. One or more states are designated as *initial states*. An *event* is an execution of a statement of a procedure. We use $s \xrightarrow{e} t$ to denote the fact that state t is reached from state s via the occurrence of event e. A *history* of the construction is a sequence (either finite or infinite) $s_0 \xrightarrow{e_0} s_1 \xrightarrow{e_1} \cdots$ where s_0 is an initial state. It is important to note that a given statement may be executed many times in a history; each such execution corresponds to a distinct event. Event e *precedes* another event f in a history iff e occurs before f in the history.

The sequence of events in a history corresponding to some procedure invocation is called an *operation*. Note that every event in a history is part of some operation of the constructed register. Thus, our notion of a history abstracts from those "external activities" of the processes sharing the constructed register that do not directly affect that register. An operation p *precedes* another operation q in a history iff each event of p precedes all events of q. An operation of a Reader (Writer) procedure is called *Read operation* (*Write operation*).[3] A Write operation of component k of the constructed composite register, where $0 \leq k < C$, is called a *k-Write operation*.

As mentioned above, each Reader procedure has an output parameter for each component of the constructed register, which is used to return the value read from that component; the value read by a Read operation from a component is called the *output value* of the operation for that component. As also mentioned above, each Writer procedure has an input parameter that specifies the value to be written to the constructed register; the value written to the constructed register by a Write operation is called the *input value* of that operation.

An operation of a procedure P in a history is *complete* iff the last event of the operation occurs as the

[3]In order to avoid confusion, we henceforth capitalize the terms "Read" and "Write" when referring to the operations of the constructed register, and leave them uncapitalized when referring to the variables used in the construction.

result of executing the **return** statement of P. A history is *well-formed* iff each operation in the history is complete.

Given this terminology, we are now in a position to state the correctness condition for a construction. In order to avoid special cases in the correctness condition, we make the following assumption concerning the initial Write operations.

Initial Writes: For each k, where $0 \leq k < C$, there exists a k-Write operation that precedes each other k-Write operation and all Read operations. $\Box$

The correctness condition is based upon the notion of "linearizability." Linearizability provides the illusion that each operation is executed instantaneously, despite the fact that it is actually executed as a sequence of events. Intuitively, a history is linearizable if every operation in the history "appears" to take effect at some point between its first and last events. It can be shown that the following definition is equivalent to the more general definition of linearizability given by Herlihy and Wing in [19], when restricted to the special case of constructing a composite register.

Linearizable Histories: Let h be a well-formed history of a construction. History h is *linearizable* iff the precedence relation on operations (which is a partial order) can be extended[4] to a total order $\sqsubset$ where for each Read operation r in h and each k in the range $0 \leq k < C$, the output value of r for component k is the same as the input value of the k-Write operation v defined as follows: $v \sqsubset r \wedge \neg(\exists w : w \text{ is a } k\text{-Write} : v \sqsubset w \sqsubset r)$. $\Box$

Note that the Write operation v in the definition above exists by our assumption concerning the initial Writes. A construction of a composite register is *correct* iff it satisfies the Atomicity and Wait-Freedom restrictions and each of its well-formed histories is linearizable.

3 Shrinking Lemma

The correctness condition given in Section 2, while intuitive, is rather difficult to use. We now present a lemma that gives a set of conditions that are sufficient for establishing that a history is linearizable. Intuitively, a history is linearizable if each operation in the history can be shrunk to a point; that is, there exists a point between the first and last events of each operation at which that operation appears to take effect. For this reason, the following lemma is referred to as the "Shrinking Lemma."

Shrinking Lemma: A well-formed history h is linearizable if for each k, where $0 \leq k < C$, there exists a function ϕ_k that maps every Read operation and k-Write operation in h to some natural number, such that the following five conditions hold.

- *Uniqueness*: For each pair of distinct k-Write operations v and w in h, $\phi_k(v) \neq \phi_k(w)$. Furthermore, if v precedes w, then $\phi_k(v) < \phi_k(w)$.

[4]A relation R over a set S *extends* another relation R' over S iff for each x and y in S, $xR'y \Rightarrow xRy$.

- *Integrity*: For each Read operation r in h, and for each k in the range $0 \leq k < C$, there exists a k-Write operation w in h such that $\phi_k(r) = \phi_k(w)$. Furthermore, the output value of r for component k is the same as the input value of w.

- *Proximity*: For each Read operation r in h and each k-Write operation w in h, if r precedes w then $\phi_k(r) < \phi_k(w)$, and if w precedes r then $\phi_k(w) \leq \phi_k(r)$.

- *Read Precedence*: For each pair of Read operations r and s in h, if $(\exists k :: \phi_k(r) < \phi_k(s))$ or if r precedes s, then $(\forall k :: \phi_k(r) \leq \phi_k(s))$.

- *Write Precedence*: For each Read operation r in h, and each j-Write operation v and k-Write operation w in h, where $0 \leq j < C$ and $0 \leq k < C$, if v precedes w and $\phi_k(w) \leq \phi_k(r)$, then $\phi_j(v) \leq \phi_j(r)$. $\Box$

Uniqueness totally orders the Write operations on a given component in accordance with the partial precedence ordering defined by h. According to Integrity, the output value of a Read operation for a given component must equal the input value of some Write operation for that component. This condition prohibits a Read operation from returning a predetermined value for some component. Proximity ensures that a Read operation does not return a value from the "future," or one from the "far past" that has subsequently been "overwritten" (i.e., each output value of a Read operation must be the input value of a Write operation in close proximity). Read Precedence disallows two Read operations from obtaining inconsistent snapshots. Write Precedence orders Write operations of one component with respect to Write operations of another component. Conditions similar to Integrity, Proximity, and Read Precedence have been used elsewhere as a correctness condition for atomic register constructions; see, for example, the Integrity, Safety, and Precedence conditions in [32], Proposition 3 in [25], and the definition of an atomic run and the Shrinking Function Theorem in [12].

The correctness proof for the Shrinking Lemma is given in [3]. The proof is somewhat tedious, but is not hard. First, the precedence relation on operations in history h is augmented by adding pairs of operations. These added pairs of operations are defined based upon the five conditions of the lemma. Then, the resulting relation is shown to be an irreflexive partial order. Finally, it is shown that any extension of this relation to an irreflexive total order satisfies the condition given in the definition of a linearizable history in Section 2.

4 $C/B/W/R$ Construction

In this section, we present a construction of a $C/B/W/R$ composite register. The construction is based upon a single $CW/B'/1/CW + R$ composite register, where, as explained below, $B' = \Theta(B + W \log W)$. This single-writer composite register is used to record input values (and associated information) of Write operations of the constructed register.

The basic idea behind the construction is as follows. Each Write operation appends an integer "tag" to its input value. A Read operation returns a value for component k of the constructed register by comparing the tags of input values stored in that component, choosing the input value with the maximum tag. Note that, because all input values and tags are stored in a single-writer composite register, a Read operation can obtain all information it needs to compute its C return values in a single snapshot.

To ensure that tags can be stored over some bounded range, a mechanism is needed for distinguishing "new" tags from "old" ones (so that new tags are not confused with old ones in the event that tags "wrap around"). We use sequence numbers to distinguish old tags from new ones. In addition to choosing a tag, each Write operation chooses a sequence number and makes copies of other Writers' sequence numbers. The sequence number chosen by a Write operation is selected so as to be distinct from all corresponding copies. Once the sequence number associated with a given tag has been copied "several" times by the same Writer, that tag is considered old.

A more detailed description of the construction is presented next in Section 4.1. A discussion of space and time complexity follows in Section 4.2. Then, in Section 4.3, a correctness proof is given. Finally, in Section 4.4, we show that the tags used in the construction can be stored over a bounded range.

4.1 Informal Description

The architecture of our $C/B/W/R$ construction is shown in Figure 2. This figure depicts only the W Writers for component k. The construction itself is given in Figure 3. Other than auxiliary variables, the only shared variable used in the construction is variable Q, a CW-component composite register. Each pair of variables $Q[k,m]$ and $Q[k,W+m]$, where $0 \leq k < C$ and $0 \leq m < W$, corresponds to a single component of Q and is written by Writer (k,m). All $CW+R$ Reader and Writer procedures read Q. Unless indicated otherwise, the term "component" is henceforth assumed to refer to the constructed composite register; we call each $Q[k,j]$ an "element" of Q. (Thus, each component of the constructed register consists of $2W$ elements.)

As seen in Figures 2 and 3, each element of Q consists of the following fields: *val*, *tag*, *done*, *flag*, *seq*, *count*, and *phi*. The *val* field is used to record the input value of a Write operation. The *tag* field is used to identify the most recently-written input value. We initially assume that each *tag* field is an integer, but later show that the *tag* fields can be restricted to range over $0..8W-2$. The *done* field is boolean and is used to distinguish between the two writes to Q by a Write operation. The *flag*, *seq*, and *count* fields are used to bound the size of the *tag* fields. This is explained in detail below.

The *phi* field is an auxiliary variable. (To emphasize that this field is auxiliary, we have enclosed it in parentheses in Figure 2.) Note also that each Reader procedure has a private auxiliary variable *phi*, and each Writer procedure has two private auxiliary variables *phi0* and *phi1*. These auxiliary variables are used in defining the functions $\phi_0, \ldots, \phi_{C-1}$ of the Shrinking Lemma. The values assigned to *phi0* and *phi1* are determined by using a shared integer auxiliary array P. We stress that these auxiliary variables are used only to facilitate the proof of correctness, and have no bearing on the correctness of the construction (no auxiliary variable's value is ever assigned to a nonauxiliary variable or tested in any control statement).

Before considering the Reader and Writer procedures depicted in Figure 3, several comments concerning notation are in order. The initialization requirement is defined by the **initialization** sections given with the shared variable declarations and within each procedure (if any). Each initial state of the construction is required to satisfy this initialization requirement. (If a given variable is not included in any **initialization** section, then its initial value is arbitrary.) To make the construction easier to understand, the keywords **read** and **write** are used to distinguish reads and writes of (nonauxiliary) shared variables from reads and writes of private variables. We use $\oplus$ to denote modulo-$2W$ addition. Finally, each labeled sequence of statements

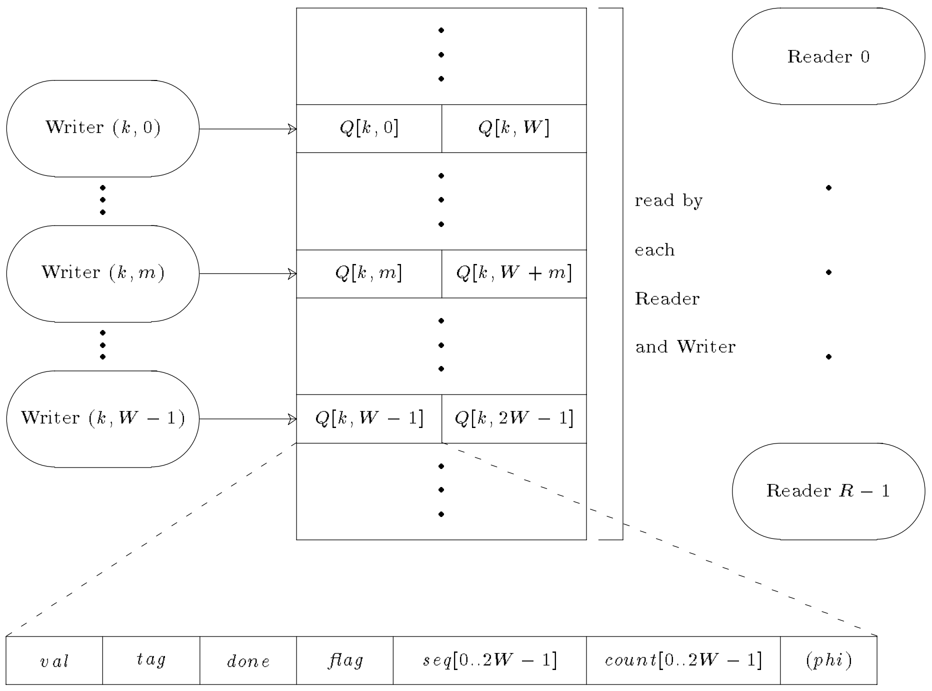

Figure 2: $C/B/W/R$ construction architecture.

is assumed to be a single atomic statement.

Each Write operation writes to a particular element of Q; we call a Write operation that writes to $Q[k, j]$ a (k, j)-*Write operation*. The Write operations for a given component follow a protocol that is similar to that used in the multi-writer atomic register construction of Vitanyi and Awerbuch [34]. Each Write operation for a particular component appends a "tag" to its input value; a Write operation computes its tag by incrementing the value of the maximum tag that it reads from the elements of Q corresponding to its component. A Read operation returns the value from the element of that component with the maximum tag (ties are broken using the indices of the Writers). The C values returned by a Read operation constitute a consistent snapshot since all of the elements of Q are read in a single statement.

Each Write operation consists of two phases: in each phase, Q is read and then written. As we shall see later, a Read operation determines its output value for component k based solely on the values of those elements $Q[k, j]$ such that $Q[k, j].done$ holds. Thus, if a k-Write operation is between phases when a Read operation reads Q, then the element of Q that is written by that Write operation is ignored when the Read operation determines its output value for component k. As a result, the value of component k is well-defined only if there exists an element $Q[k, j]$ such that $Q[k, j].done$ holds. To ensure that this is always the case, successive operations of the same Writer write to different elements of Q. In particular, the operations of

```
type $valtype$ = a $B$-bit value;
     $Qtype$  = record
                   $val : valtype$;
                   $tag$ : integer;   /* As shown in Section 4.4, a range of $0..8W-2$ suffices */
                   $done,\ flag$ : boolean;
                   $seq$ : array[$0..2W-1$] of $0..2W$;
                   $count$ : array[$0..2W-1$] of $0..3$;
                   $phi$ : integer   /* Auxiliary variable */
                end
shared var
     $Q$ : array[$0..C-1$][$0..2W-1$] of $Qtype$;      /* Single-writer composite register */
     $P$ : array[$0..C-1$] of integer      /* Auxiliary variable */
initialization
     $(\forall k, m : 0 \le k < C \ \wedge\ 0 \le m < 2W : Q[k,m].tag = 0 \ \wedge\ Q[k,m].done \ \wedge\ \neg Q[k,m].flag\ \wedge$
     $0 \le Q[k,m].phi < P[k] \ \wedge\ (\forall n : 0 \le n < 2W : Q[k,m].seq[n] = m \ \wedge\ Q[k,m].count[n] = 0))$

procedure $Reader(j : 0..R-1)$ returns array[$0..C-1$] of $valtype$
private var
      $x : Qtype$;
      $k : 0..C-1$;
      $n : 0..2W-1$;
      $max$ : array[$0..C-1$] of $0..2W-1$;
      $val$ : array[$0..C-1$] of $valtype$;
      $phi$ : array[$0..C-1$] of integer    /* Auxiliary variable */
begin
   0:  read $x := Q$;
   1:  for $k = 0$ to $C-1$ do
           select $max[k]$ such that $ALIVE(x, k, max[k]) \wedge$               /* Note: $ALIVE(x,k,n)$ holds for some $n$ by
                 $(\forall n : ALIVE(x,k,n) : (x[k,n].tag,\ n) \le (x[k, max[k]].tag,\ max[k]))$;        Lemma 6 of Section 4.3 */

           $val[k],\ phi[k] := x[k, max[k]].val,\ x[k, max[k]].phi$
       od;
   2:  return($val[0],\ \ldots,\ val[C-1]$)
end
```

Figure 3: $C/B/W/R$ construction.

```
procedure $Writer(k : 0..C-1;\ m : 0..W-1;\ val : valtype)$
private var
        $y,\ z : Qtype$;
        $i,\ max,\ n : 0..2W-1$;
        $tag$ : integer;    /* As shown in Section 4.4, a range of $0..8W-2$ suffices */
        $flag$ : boolean;
        $seq$ : array$[0..2W-1]$ of $0..2W$;
        $count$ : array$[0..2W-1]$ of $0..3$;
        $phi0,\ phi1$ : integer   /* Auxiliary variables */
initialization
        $(\forall n : 0 \le n < 2W : y[k,n].tag = z[k,n].tag = 0)$
begin
   0:   if $i = m$ then $i := W + m$ else $i := m$ fi;  $phi0,\ P[k] := P[k],\ P[k]+1$;

        /* First Phase: update $val$, $seq[0..2W-1]$, and $count[0..2W-1]$ fields */
   1:   read $y := Q$;
   2:   select $seq[i]$ such that $(\forall n : 0 \le n < 2W : seq[i] \ne y[k,n].seq[i])$;
   3:   for $n = 0$ to $2W-1$ skip $i$ do
            $seq[n] := y[k,n].seq[n]$;
            if $seq[n] \ne y[k, i \oplus W].seq[n]$ then $count[n] := 0$ else $count[n] := min(3, y[k, i \oplus W].count[n] + 1)$ fi
        od;
        /* The following fields are left unchanged by first phase */
   4:   $tag,\ flag,\ count[i],\ phi1 := y[k,i].tag,\ y[k,i].flag,\ 0,\ y[k,i].phi$;
   5:   write $Q[k,i] := (val,\ tag,\ false,\ flag,\ seq[0..2W-1],\ count[0..2W-1],\ phi1)$;

        /* Second Phase: update $tag$, $flag$, and $phi$ fields */
   6:   read $z := Q$;
   7:   $flag := (\exists n : 0 \le n < 2W \ \wedge\ n \ne i : z[k,n].count[i] \ge 2 \ \wedge\ z[k,n].seq[i] = seq[i])$;
   8:   select $max$ such that $ALIVE(z,k,max) \ \wedge$                    /* Note: $ALIVE(z,k,n)$ holds for some $n$ by
                  $(\forall n : ALIVE(z,k,n) : (z[k,n].tag,\ n) \le (z[k,max].tag,\ max))$;          Lemma 6 of Section 4.3 */
   9:   $tag := z[k,max].tag + 1$;
  10:   $phi1,\ P[k] := P[k],\ P[k]+1$;
        write $Q[k,i] := (val,\ tag,\ true,\ flag,\ seq[0..2W-1],\ count[0..2W-1],\ phi1)$;
  11:   return
end
```

Figure 3: $C/B/W/R$ construction (continued).

Writer (k,m), where $0 \le k < C$ and $0 \le m < W$, alternate between writing to $Q[k,m]$ and $Q[k,m+W]$. This is why each component of the constructed register consists of $2W$ elements of Q, instead of just W. This strategy guarantees that it is always the case that the *done* field is true for at least half of the elements of each component.

In order to determine which element of a component has the maximum tag, it is necessary only to consider those elements that have been written "recently." By exploiting this fact, it is possible to bound the size of the *tag* fields. We say that a recently-written element is "alive." The alive elements are identified by including a number of additional fields in each element of Q. One of these fields is an array $seq[0..2W-1]$ of "sequence numbers." The field $Q[k,j].seq[j]$ holds the "primary" sequence number of the most recent (k,j)-Write operation. The field $Q[k,j].seq[n]$, where $n \neq j$, is a copy of $Q[k,n].seq[n]$ (which records the primary sequence numbers of (k,n)-Write operations) as read by the most recent (k,j)-Write operation.

Each Write operation updates its *seq* fields during its first phase. A (k,j)-Write operation changes the value of its primary sequence number, $Q[k,j].seq[j]$, so that the resulting value is distinct from all copies of it. (Observe that, because each sequence number ranges over $0..2W$, it is possible for a Write operation to choose a value for its primary sequence number differing from its previous value and any of the $2W-1$ copies of it.) At the same time, the Write operation updates its copy of each other primary sequence number to correspond to the current value of that sequence number as stored in Q. More specifically, each (k,j)-Write operation tries to make the value of $Q[k,j].seq[n]$ equal to that of $Q[k,n].seq[n]$, for each $n \neq j$.

If the primary sequence number for a given element of Q remains unchanged for a sufficiently long period of time, then it will eventually be copied by some Write operation. An element of Q is no longer "alive" once its primary sequence number has been copied by "several" successive operations of the same Writer. This condition is detected by means of the *flag* and *count* fields in Q. Each Write operation updates the *count* fields of its element of Q during its first phase, and updates the *flag* field of its element in its second phase, while its tag value is being computed. The bit $Q[k,j].flag$ is set by a (k,j)-Write operation if it detects that its own primary sequence number, as recorded in $Q[k,j].seq[j]$, has been copied by at least three successive operations of some other Writer. The field $Q[k,j].count[n]$, where $n \neq j$, is incremented (until a maximum value of 3) by a (k,j)-Write operation if the value it reads from $Q[k,n].seq[n]$ equals the value read from $Q[k,n].seq[n]$ by the preceding operation of the same Writer (note that the preceding operation is a $(k, j \oplus W)$-Write operation).

As shown in Section 4.4, the tags of the alive elements of a given component are within a range of size $4W$. Therefore, we can restrict the size of each tag to range over $0..8W-2$. (If the smallest alive tag value is 0, then the largest is $4W-1$. If the smallest is $4W-1$, then the largest is $8W-2$.) The maximum tag for the alive elements can then be determined with respect to this range. To see that the *tag* fields can be restricted to range over *some* bounded range, consider a (k,j)-Write operation w. Note that the maximum tag for the alive elements of component k can increase by a "large" amount between w's second read and second write of Q (i.e., while w is computing its tag) only if a "large" number of k-Write operations occur in this interval. But, in this case, the primary sequence number for w will be read by "several" successive operations of some Writer, and $Q[k,j]$ will not be alive after w's second write to Q. So, if $Q[k,j]$ *is* alive after w's second write to Q, then the value of its *tag* field will differ from that of some other alive element by only a "small" amount.

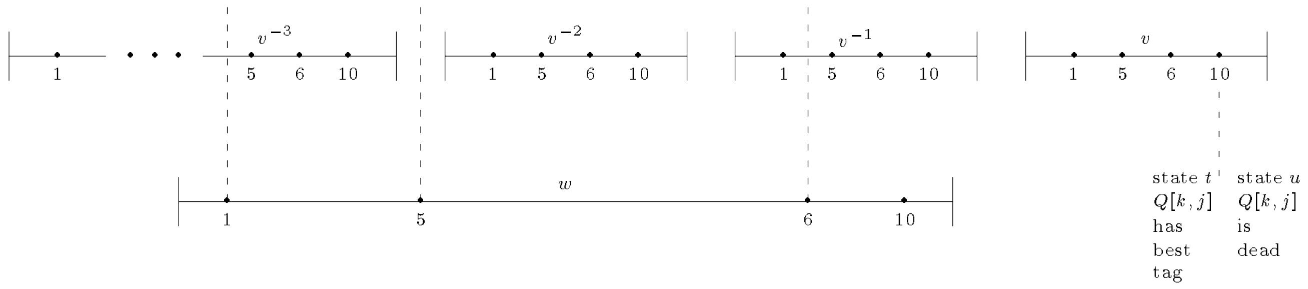

Figure 4: Example history.

The alive elements of Q are determined formally by the predicate *ALIVE*, which is defined as follows.

Definition of *ALIVE***:** Let $0 \le k < C$ and $0 \le j < 2W$. Then,

$$ALIVE(Q,k,j) \equiv Q[k,j].done \ \wedge \ \neg Q[k,j].flag \ \wedge$$
$$(\forall n : n \neq j \ \wedge \ Q[k,n].done : Q[k,n].count[j] = 3 \ \Rightarrow \ Q[k,n].seq[j] \neq Q[k,j].seq[j]) \ . \ \Box$$

According to the first conjunct in this definition, $ALIVE(Q,k,j)$ fails to hold if the last Write operation to write to $Q[k,j]$ has done so only once, i.e., is between phases. According to the second conjunct, $ALIVE(Q,k,j)$ fails to hold if the last Write operation to update the bit $Q[k,j].flag$ detected that its own primary sequence number, i.e., that recorded in $Q[k,j].seq[j]$, was read by at least three successive operations of some other Writer (see statement 7 of the Writer procedure). According to the third conjunct, $ALIVE(Q,k,j)$ fails to hold if at least four successive operations of some Writer have read the current value of $Q[k,j].seq[j]$.

We complete our explanation of the $ALIVE(Q,k,j)$ predicate by explaining more precisely the role of the *flag* and *count* fields. For the sake of this discussion, let us refer to the maximum tag value taken from the alive elements of a component as the "best" tag value for that component. As seen in Section 4.3, to show that each Read operation gets a consistent snapshot, one of the key proof obligations is that of showing that each component's best tag value never decreases. Intuitively, this property ensures that if a Read operation r precedes another Read operation s, then the return value of r for component k will not be "more recent" than that of s. In establishing this proof obligation, one important case that arises occurs when the element containing the previous best tag value for a component becomes dead.

This case is illustrated in Figure 4. In this and subsequent figures, operations are denoted by line segments, with "time" running from left to right. An event is denoted by a point along a line segment, labeled by the corresponding statement number. In Figure 4, t and u are consecutive states. At state t, element $Q[k,j]$, which was last written by Write operation w, has the best tag value for component k. State u is reached from t via a write to $Q[k,n]$, $n \neq j$, by Write operation v. This write establishes $Q[k,n].done \ \wedge \ Q[k,n].count[j] = 3 \ \wedge \ Q[k,n].seq[j] = Q[k,j].seq[j]$, which implies that $Q[k,j]$ is no longer alive at state u. Note that v is the fourth consecutive operation of the same Writer to copy w's primary sequence number; call v's three predecessors v^{-3}, v^{-2}, and v^{-1}, respectively. As illustrated in Figure 4, v^{-3} may have obtained w's sequence number "by accident," i.e., by copying an old sequence number from

$Q[k,j]$ that w has subsequently recycled. However, it can be shown that v^{-2} and v^{-1} must have actually read from $Q[k,j]$ after it was written by w. Note that, because $Q[k,j]$ is alive at state t, w computes its $flag$ variable to be false. Thus, it must be the case that v^{-1} (the third consecutive operation to copy w's sequence number) performs its first write to Q *after* w's second read from Q. This implies that v^{-1} and v compute their tag values after w's second read from Q. From this, it is possible to show that the tag value for v^{-1} is at least that of w, and the tag value for v is bigger than that of w. The latter can be used to show that the best tag value for component k increases from state t to state u.

One final point bears mentioning before leaving this example. Observe that, if v^{-1} had performed its first write to Q before w's second read, then w would have computed its $flag$ variable to be true. Intuitively, this corresponds to the situation in which w is overwritten by a concurrent k-Write operation. Note that, in this case, because v^{-2} is completely contained within w, in linearizing the operations, w can be "shrunk" to occur at a point immediately prior to v^{-2}. Had the algorithm been such that w set its $flag$ upon finding that its sequence number had been copied once or twice, rather than three times, then the existence of such an operation contained within w would not have been guaranteed.

The method described above for bounding the *tag* fields is very similar to the one employed in the atomic register construction of Li, Tromp, and Vitanyi in [26]. This construction is also based on the protocol of Vitanyi and Awerbuch described above. The method for bounding the *tag* fields employed by Li et al. is similar to the one described above in that "newer" operations mark the tag values of "older" ones. Although their implementation of markers differs from that described above, the two implementations are very similar: in both cases, a tag value is declared "old" (no longer alive) once it has been marked several times by some Writer.

4.2 Complexity

The space and time complexity of our construction depend on the space and time complexity of Q. If we remove the auxiliary *phi* fields from Q, then the size of each field of each element is as follows: *val* uses B bits; *tag* uses $\log(8W-1)$ bits; *done* and $flag$ each use 1 bit; $seq[i]$, $0 \leq i < 2W$, uses $\log(2W+1)$ bits; and $count[i]$, $0 \leq i < 2W$, uses 2 bits. Thus, variable Q is a $CW/B'/1/CW+R$ composite register, where $B' = \Theta(B + W\log W)$. Let $S(C,B,W,R)$ denote the number of single-reader, single-writer atomic bits required to construct a $C/B/W/R$ composite register. Then, for our construction, $S(C,B,W,R) = S(CW,B',1,CW+R)$. Similarly, let $TR(C,B,W,R)$ and $TW(C,B,W,R)$ denote the number of reads and writes of multi-reader, single-writer atomic registers required to Read and Write, respectively, a $C/B/W/R$ composite register. (For simplicity, we do not go down to the level of single-reader, single-writer atomic bits when computing time complexity.) Then, for our construction, $TR(C,B,W,R) = TR(CW,B',1,CW+R)$ and $TW(C,B,W,R) = 2TR(CW,B',1,CW+R) + 2TW(CW,B',1,CW+R)$. The actual values of $S(CW,B',1,CW+R)$, $TR(CW,B',1,CW+R)$, and $TW(CW,B',1,CW+R)$ depend on the implementation of Q.

Let us now compare the complexity of our construction with the multi-writer construction of Afek et al. [1]. In [1], the assumption is made that each of the processes that share the constructed register can both read the register and write each component. Under this assumption, it is possible to reduce the complexity of our construction. In particular, suppose that there are N processes, and that each process may write each component. Then, we have $R = N$ and $W = N$. Furthermore, each process i, where $0 \leq i < N$, writes $Q[k, i]$ and $Q[k, i + N]$ for each component k. This implies that we can store $Q[0, i], Q[0, i + N], \ldots, Q[C - 1, i], Q[C - 1, i + N]$ as a single component (of Q), as all of these elements are written only by process i. This reduces the number of components of Q from CN to N and increases the number of bits per component from B' to CB'. Thus, the space complexity becomes $S(C, B, N, N) = S(N, B'', 1, N)$, where $B'' = CB' = \Theta(CB + CN \log N)$. The time complexity for Reading and Writing, respectively, becomes $TR(C, B, N, N) = TR(N, B'', 1, N)$ and $TW(N, B, N, N) = 2TR(N, B'', 1, N) + 2TW(N, B'', 1, N)$. If the single-writer composite register construction of [1] is used to implement Q, then we have $S(N, B'', 1, N) = \Theta(B''N^3)$ and $TR(N, B'', 1, N) = TW(N, B'', 1, N) = \Theta(N^2)$. Thus, for our construction, $S(C, B, N, N) = \Theta(BCN^3 + CN^4 \log N)$ and $TR(C, B, N, N) = TR(C, B, N, N) = \Theta(N^2)$.[5] The multi-writer construction given in [1] has space complexity that is $\Theta(BCN^2 + CN^3 + N^4)$ and time complexity that is $\Theta(N^3)$. (Note that Afek et al. incorrectly state that the time complexity of our construction is $\Theta(N^4)$).

4.3 Correctness Proof

To prove that the construction is correct, we must show that it satisfies the Atomicity and Wait-Freedom restrictions and each of its well-formed histories is linearizable. The Atomicity restriction is satisfied because Q is the only (nonauxiliary) shared variable, and Q is a single-writer composite register. The Wait-Freedom restriction is satisfied because no procedure contains any unbounded loops or synchronization primitives. In this section, we prove that each well-formed history is linearizable. We first prove this for the construction as is, i.e., with unbounded tags. In the next section, we show how to transform the construction into one with bounded tags.

The correctness proof is based on the Shrinking Lemma. We first define functions $\phi_0, \ldots, \phi_{C-1}$ for a given history, and then show that the defined ϕ's satisfy the five conditions of Uniqueness, Integrity, Proximity, Read Precedence, and Write Precedence. We now present a number of definitions and notational conventions that will be used in the rest of the paper.

Unless stated otherwise, we henceforth assume that k ranges over $\{0, \ldots, C - 1\}$, that i, j, and n each range over $\{0, \ldots, 2W - 1\}$, and that v and w are k-Write operations. 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.

all subscripts and superscripts

$[\,], (\,), |\,|$

$.$

$!$

$\neg, :$

$+, -, \oplus$

$=, \neq, <, >, \leq, \geq, \prec, \preceq$

$\wedge, \vee$

[5] See footnote 1.

unless

$\Rightarrow$, $\equiv$

$\models$

If event e precedes event f, then we write $e \prec f$. We let $(e \preceq f) \equiv (e = f \lor e \prec f)$. If x is a private variable of operation p, then $p!x$ denotes the final value of variable x as assigned by p. (Note that, in the proof of correctness, $p!x$ can be thought of as a "constant" value: once we have fixed on a particular history, and an operation p in that history, the value $p!x$ is also fixed. In other words, $p!x$ is not to be thought of as a variable whose value varies from state to state.) Let i be a label of a statement of some Reader or Writer procedure, and let p denote an operation of that procedure. Then, $p\!:\!i$ denotes the event corresponding to the execution of statement i by p.

If E is an expression that holds at state t, then we write $t \models E$. Whenever we say that a given assertion holds without referring to a particular state, we mean that the assertion is an *invariant*; i.e., it is true at each state of every history. Let E and F be two expressions over the variables of a construction. Following [15], we say that the assertion E *unless* F holds iff for every pair of consecutive states in any history, if $E \land \neg F$ holds in the first state, then $E \lor F$ holds in the second state. An assertion E is *stable* iff E *unless false* holds.

We assume that each state in every history is distinct. This assumption is easy to ensure by introducing an integer auxiliary variable that is incremented with each event. In the history $t_0 \xrightarrow{e_0} \cdots t_i \xrightarrow{e_i} t_{i+1} \cdots$, 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 the event *following* state t_i. Note that the events prior to and following a given state are uniquely defined since, by assumption, each state appears at most once in a history.

Let p be an operation of some Reader or Writer procedure P. As in Section 4.1, we use p^{-1} to denote the operation of P that immediately precedes p, p^{-2} to denote the operation of P that immediately precedes p^{-1}, etc. Similarly, we use p^{+1} to denote the operation of P that immediately succeeds p, p^{+2} to denote the operation of P that immediately succeeds p^{+1}, etc. Observe that if w is a (k, j)-Write operation, then w^{-1} and w^{+1} (if they exist in the given history) are $(k, j \oplus W)$-Write operations. This follows from the fact that each Writer (k, m) alternates between writing to $Q[k, m]$ and $Q[k, m + W]$.

Let X be a shared variable of the construction, and let p be an operation. The assertion $last(X) = p$ holds at a state iff the last event to write to X before that state is an event of p. (Note that, because the construction uses only single-writer shared variables, p is an operation of the Reader or Writer with write-access to X.) If e is an event in some history, then $after(e)$ is true at a state of the history iff that state occurs after event e.

Based on the definition of *ALIVE* given in Section 4.1, we define two predicates: *alive* indicates whether a Write operation is "alive," and *pref* indicates whether an alive operation is "preferable," i.e., has the "best" tag value for its component.

Definition of *alive* and *pref*: Let w be a (k, j)-Write operation. Then,

$$alive(w, k) \equiv last(Q[k, j]) = w \land ALIVE(Q, k, j)$$
$$pref(w, k) \equiv alive(w, k) \land (\forall v : alive(v, k) : (v!tag, v!i) \leq (w!tag, j)) \qquad \Box$$

As mentioned in Section 4.1, each procedure has one or more private auxiliary variables. These variables have been introduced in order to facilitate the definition of $\phi_0, \ldots, \phi_{C-1}$.

Definition of ϕ_k: Let r be a Read operation and let w be a k-Write operation. Then, ϕ_k is defined as follows.

$$\phi_k(r) \equiv r!phi[k]$$
$$\phi_k(w) \equiv \begin{cases} w!phi1 & \text{if } pref(w,k) \text{ holds at the state following } w\!:\!10 \\ w!phi0 & \text{otherwise} \end{cases}$$ □

Before establishing the conditions of Uniqueness, Integrity, Proximity, Read Precedence, and Write Precedence, we first prove a number of lemmas. The following lemma gives us a means for determining the value of $Q[k,j]$ at a given state. According to the first part of the lemma, if $Q[k,j]$ was last written by operation v, then $Q[k,j].val$ equals the value of v's input parameter val, and the seq and $count$ fields in $Q[k,j]$ equal the corresponding values computed by v. Note that these fields of Q are all updated during the first phase of v. According to the second part of the lemma, if $Q[k,j]$ was last written by operation v and v has completed execution, then $Q[k,j].tag$ equals the tag value computed by v, $Q[k,j].done$ is true, and $Q[k,j].flag$ and $Q[k,j].phi$ equal the values assigned by v to its private variables $flag$ and $phi1$, respectively. Note that these fields of Q are all updated during the second phase of v.

Lemma 1: Let v be a (k,j)-Write operation. Then,

- $last(Q[k,j]) = v \;\Rightarrow\; Q[k,j].val = v!val \;\wedge\; (\forall n :: Q[k,j].seq[n] = v!seq[n] \;\wedge\; Q[k,j].count[n] = v!count[n])$
- $last(Q[k,j]) = v \;\wedge\; after(v\!:\!10) \;\Rightarrow\; Q[k,j].tag = v!tag \;\wedge\; Q[k,j].done \;\wedge\; Q[k,j].flag = v!flag \;\wedge\; Q[k,j].phi = v!phi1$

Proof: The lemma holds because $v\!:\!5$ assigns the values $v!val$, $false$, $v!seq[0..2W-1]$, and $v!count[0..2W-1]$ to the fields val, $done$, $seq[0..2W-1]$, and $count[0..2W-1]$, respectively, of $Q[k,j]$, while leaving the value of each other field unchanged; and $v\!:\!10$ assigns the values $v!tag$, $true$, $v!flag$, and $v!phi1$ to the fields tag, $done$, $flag$, and phi, respectively, of $Q[k,j]$, while leaving the value of each other field unchanged. □

The following simple lemma follows from the fact that each Write operation selects a unique value for its primary sequence number. It states that, for any (k,j)-Write operation w, at the state prior to w's first read from Q, $Q[k,n].seq[j]$ differs from the value $w!seq[j]$, for each n. Note that $Q[k,n].seq[j]$ is a copy of $Q[k,j].seq[j]$, as made by a (k,n)-Write operation, and $w!seq[j]$ is the *value* w assigns to $Q[k,j].seq[j]$. The lemma follows from the fact that w selects the value $w!seq[j]$ to be different from all copies of $Q[k,j].seq[j]$. (Again, we stress that once we have fixed upon a specific history, and a specific operation w in that history, $w!seq[j]$ is a fixed value, i.e., it is not a state-dependent quantity.)

Lemma 2: If w is a (k,j)-Write operation, then $(\forall n :: Q[k,n].seq[j] \neq w!seq[j])$ at the state prior to $w\!:\!1$.

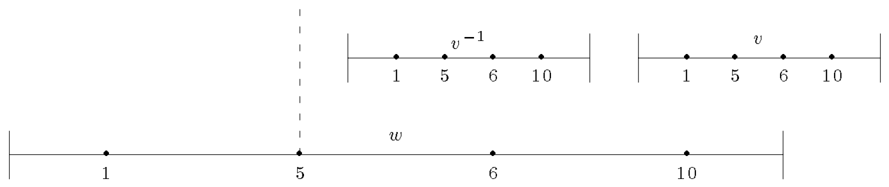

Figure 5: Proof of Lemma 3.

Proof: By statement 1 of the Writer procedure, $Q = w!y$ at the state prior to $w\!:\!1$. By statement 2 of the Writer procedure, $(\forall n :: w!seq[j] \neq w!y[k,n].seq[j])$. Hence, at the state prior to $w\!:\!1$, we have $(\forall n :: w!seq[j] \neq Q[k,n].seq[j])$. □

In the next lemma, we consider the case in which at least three successive operations of one Writer have copied the current sequence number of another Writer. This lemma is illustrated in Figure 5. The lemma states that if there exists a state at which $Q[k,n]$ and $Q[k,j]$ were last written by Write operations v and w, respectively, and if at that state $Q[k,n].count[j] \geq 2 \ \wedge\ Q[k,n].seq[j] = Q[k,j].seq[j]$ holds (indicating that v is the third of three successive operations of the same Writer to copy the current value of $Q[k,j].seq[j]$), then v^{-1}, the immediate predecessor of v, exists in the given history. Moreover, the first read by v^{-1} from Q happened after w wrote to Q, and the value read by v^{-1} from the field $Q[k,j].seq[j]$ during that read equals w's primary sequence number.

Lemma 3: $(\ n \neq j \ \wedge\ last(Q[k,n]) = v \ \wedge\ last(Q[k,j]) = w \ \wedge\ Q[k,n].count[j] \geq 2 \ \wedge\ Q[k,n].seq[j] = Q[k,j].seq[j]\ ) \Rightarrow (\ v^{-1} \text{ exists} \ \wedge\ w\!:\!5 \prec v^{-1}\!:\!1 \ \wedge\ v^{-1}!seq[j] = w!seq[j]\ )$.

Proof: Let $n \neq j$, let $n' = n \oplus W$, and assume that the following expression holds for some state t.

$$t \ \models\ last(Q[k,n]) = v \ \wedge\ last(Q[k,j]) = w \ \wedge\ Q[k,n].count[j] \geq 2 \ \wedge\ Q[k,n].seq[j] = Q[k,j].seq[j]$$

By Lemma 1 and the above assertion, $v!count[j] \geq 2 \ \wedge\ v!seq[j] = w!seq[j]$. We show that this implies v^{-1} exists and $v^{-1}!seq[j] = w!seq[j]$. (Note that the above assertion implies that v is a (k,n)-Write operation, w is a (k,j)-Write operation, and v^{-1}, if it exists in the given history, is a (k,n')-Write operation.)

Let u be the state prior to $v\!:\!1$. Because $v!count[j] \geq 2$, by statement 3 of the Writer procedure, $v!y[k,n'].seq[j] = v!seq[j] \ \wedge\ v!y[k,n'].count[j] \geq 1$. This implies that $u \ \models\ Q[k,n'].seq[j] = v!seq[j] \ \wedge\ Q[k,n'].count[j] \geq 1$. Because $Q[k,n'].count[j]$ is initially 0, and because $Q[k,n]$ and $Q[k,n']$ are written by the same Writer, this implies that $u \ \models\ last(Q[k,n']) = v^{-1}$. Thus, by Lemma 1, $u \ \models\ Q[k,n'].seq[j] = v^{-1}!seq[j] \ \wedge\ Q[k,n'].count[j] = v^{-1}!count[j]$. Therefore,

$$v^{-1}!seq[j] = v!seq[j] = w!seq[j] \ \wedge\ v^{-1}!count[j] \geq 1 \quad . \tag{1}$$

Our remaining proof obligation is to show that $w\!:\!5 \prec v^{-1}\!:\!1$. To this end, we first prove that $w\!:\!1 \prec v^{-1}\!:\!5$. Assume, to the contrary that $v^{-1}\!:\!5 \prec w\!:\!1$. Let e be the event prior to state t, and let t' be the state prior to $w\!:\!1$. Because $t \ \models\ last(Q[k,j]) = w$, we have $w\!:\!5 \preceq e$. Therefore, $v^{-1}\!:\!5 \prec w\!:\!1 \prec e$.

Consider the two events $w\!:\!1$ and $v\!:\!5$. Either $w\!:\!1 \prec v\!:\!5$ or $v\!:\!5 \prec w\!:\!1$. In the former case, we have $v^{-1}\!:\!5 \prec w\!:\!1 \prec v\!:\!5$; because v^{-1} and v are successive operations of the same Writer, this implies that $t' \models last(Q[k,n']) = v^{-1}$. In the latter case, we have $v\!:\!5 \prec w\!:\!1 \prec e$; because $last(Q[k,n]) = v$ at state t (i.e., the state following e), this implies that $t' \models last(Q[k,n]) = v$. Combining these two cases, by Lemma 1, we have $t' \models Q[k,n'].seq[j] = v^{-1}!seq[j] \vee Q[k,n].seq[j] = v!seq[j]$. By (1), this implies that $t' \models Q[k,n'].seq[j] = w!seq[j] \vee Q[k,n].seq[j] = w!seq[j]$, which contradicts Lemma 2. Thus, our assumption that $v^{-1}\!:\!5 \prec w\!:\!1$ is false, i.e., $w\!:\!1 \prec v^{-1}\!:\!5$.

We now prove that $w\!:\!5 \prec v^{-1}\!:\!1$. Assume, to the contrary, that $v^{-1}\!:\!1 \prec w\!:\!5$. We consider two cases, depending on the relative ordering of $w\!:\!1$ and $v^{-1}\!:\!1$. We show that both cases lead to a contradiction.

First, suppose that $w\!:\!1 \prec v^{-1}\!:\!1 \prec w\!:\!5$. Because v^{-1} is a (k,n')-Write and w is a (k,j)-Write operation, this precedence assertion implies that $j \neq n'$. Because $v^{-1}\!:\!1$ occurs between $w\!:\!1$ and $w\!:\!5$, the value of $Q[k,j]$ is the same at both the state prior to $w\!:\!1$ and the state prior to $v^{-1}\!:\!1$. By statement 1 of the Writer procedure, this implies that $w!y[k,j].seq[j] = v^{-1}!y[k,j].seq[j]$. Because w is a (k,j)-Write operation, by statement 2 of the Writer procedure, $w!seq[j] \neq w!y[k,j].seq[j]$. Because v^{-1} is a (k,n')-Write operation and $n' \neq j$, by statement 3 of the Writer procedure, $v^{-1}!seq[j] = v^{-1}!y[k,j].seq[j]$. Therefore, $w!seq[j] \neq v^{-1}!seq[j]$. However, this contradicts (1).

Now, consider the other case mentioned above, i.e., $v^{-1}\!:\!1 \prec w\!:\!1$. In this case, because $w\!:\!1 \prec v^{-1}\!:\!5$, we have $v^{-1}\!:\!1 \prec w\!:\!1 \prec v^{-1}\!:\!5$. By (1), $v^{-1}!count[j] \geq 1$. Because $v^{-1}!count[j]$ is nonzero, by statement 3 of the Writer procedure, $v^{-1}!seq[j] = v^{-1}!y[k,n].seq[j]$. This implies that $Q[k,n].seq[j] = v^{-1}!seq[j]$ at the state prior to $v^{-1}\!:\!1$. Because $v^{-1}\!:\!1 \prec w\!:\!1 \prec v^{-1}\!:\!5$, and because the (k,n')- and (k,n)-Write operations are totally ordered (being operations of the same Writer), $Q[k,n]$ has the same value both at the state prior to $v^{-1}\!:\!1$ and at the state prior to $w\!:\!1$. Therefore, $Q[k,n].seq[j] = v^{-1}!seq[j]$ at the state prior to $w\!:\!1$. By (1), this implies that $Q[k,n].seq[j] = w!seq[j]$ at that state, which contradicts Lemma 2. $\Box$

According to the following lemma, if a completed k-Write operation w is not "alive" then there exists another completed k-Write operation v such that $w\!:\!5 \prec v\!:\!1$.

Lemma 4: $(after(w\!:\!10) \wedge \neg alive(w,k)) \Rightarrow (\exists v :: w\!:\!5 \prec v\!:\!1 \;\wedge\; after(v\!:\!10))$.

Proof: Suppose that $after(w\!:\!10) \wedge \neg alive(w,k)$ holds at some state t, where w is a (k,j)-Write operation. Our proof obligation is to show that there exists a k-Write operation v such that $w\!:\!5 \prec v\!:\!1$ and $t \models after(v\!:\!10)$.

We first dispose of the case in which $t \models last(Q[k,j]) \neq w$. In this case, because $after(w\!:\!10)$ holds at t, there exists a (k,j)-Write operation w', where w precedes w', such that $t \models last(Q[k,j]) = w'$. Because successive operations of the same Writer write to different elements of Q, this implies that Write operation w^{+1} exists and $after(w^{+1}\!:\!10)$ holds at t. Because w and w^{+1} are successive operations of the same Writer, $w\!:\!5 \prec w^{+1}\!:\!1$. This establishes our proof obligation.

In the remainder of the proof, we assume that $t \models last(Q[k,j]) = w$. In this case, because $t \models \neg alive(w,k)$, by the definition of $alive$, $t \models \neg ALIVE(Q,k,j)$. We now show that there exists a state u, where u either equals or occurs before t, such that for some $n \neq j$ the following expression holds.

$$u \models \mathit{last}(Q[k,j]) = w \ \wedge\ Q[k,n].\mathit{count}[j] \geq 2 \ \wedge\ Q[k,n].\mathit{seq}[j] = Q[k,j].\mathit{seq}[j] \qquad (2)$$

Because $t \models \mathit{last}(Q[k,j]) = w \ \wedge\ \mathit{after}(w\!:\!10)$, by Lemma 1, $t \models Q[k,j].\mathit{done}$. Therefore, because $t \models \neg \mathit{ALIVE}(Q,k,j)$, by the definition of ALIVE, there are two possibilities to consider: (i) there exists $n \neq j$ such that $t \models Q[k,n].\mathit{count}[j] = 3 \ \wedge\ Q[k,n].\mathit{seq}[j] = Q[k,j].\mathit{seq}[j]$; or (ii) $t \models Q[k,j].\mathit{flag}$. If (i) holds, then take $u = t$. Because $t \models \mathit{last}(Q[k,j]) = w$, this establishes (2).

Now, suppose that (ii) holds, i.e., $Q[k,j].\mathit{flag}$ holds at t. Because $t \models \mathit{last}(Q[k,j]) = w \ \wedge\ \mathit{after}(w\!:\!10)$, by Lemma 1, $t \models Q[k,j].\mathit{flag} = w!\mathit{flag}$. Therefore, $w!\mathit{flag}$ is true. Let u be the state prior to $w\!:\!6$. Because $\mathit{after}(w\!:\!10)$ holds at t, u occurs before t. Because u occurs in the interval of states between $w\!:\!5$ and $w\!:\!10$, $u \models \mathit{last}(Q[k,j]) = w$. Because $w!\mathit{flag}$ holds, there exists $n \neq j$ such that $w!z[k,n].\mathit{count}[j] \geq 2 \ \wedge\ w!z[k,n].\mathit{seq}[j] = w!\mathit{seq}[j]$. By the definition of state u, $u \models Q = w!z$. Also, because event $w\!:\!5$ assigns $Q[k,j].\mathit{seq}[j] := w!\mathit{seq}[j]$, we have $u \models Q[k,j].\mathit{seq}[j] = w!\mathit{seq}[j]$. Therefore, $u \models Q[k,n].\mathit{count}[j] \geq 2 \ \wedge\ Q[k,n].\mathit{seq}[j] = Q[k,j].\mathit{seq}[j]$. This establishes (2).

We now use (2) to establish our proof obligation. Let v be the Write operation such that $u \models \mathit{last}(Q[k,n]) = v$. ($v$ exists because $Q[k,n].\mathit{count}[j]$ is initially 0.) Then, by (2) and Lemma 3, the Write operation v^{-1} exists and $w\!:\!5 \prec v^{-1}\!:\!1$. Because v^{-1} and v are successive operations of the same Writer and $\mathit{last}(Q[k,n]) = v$ at state u, $\mathit{after}(v^{-1}\!:\!10)$ holds at u. Because u either equals or occurs before t, this implies that $\mathit{after}(v^{-1}\!:\!10)$ holds at t. This establishes our proof obligation. $\Box$

The next lemma shows that $\mathit{alive}(w,k)$ holds for some w at every state that occurs after the initial k-Write operation.

Lemma 5: $(\exists v :: \mathit{after}(v\!:\!10)) \Rightarrow (\exists w :: \mathit{alive}(w,k) \ \wedge\ (\forall w' : w\!:\!5 \prec w'\!:\!5 : \neg \mathit{after}(w'\!:\!10))$.

Proof: Let t be a state such that for some k-Write operation v, $t \models \mathit{after}(v\!:\!10)$. Let S denote the set of k-Write operations defined as follows: p is in S iff $t \models \mathit{after}(p\!:\!10)$. Note that v is in S, i.e., S is nonempty. Let w denote the k-Write operation in S such that for each other k-Write operation w' in S, $w'\!:\!5 \prec w\!:\!5$. Then, by Lemma 4, $t \models \mathit{alive}(w,k)$. $\Box$

Corollary: $(\exists v :: \mathit{after}(v\!:\!10)) \Rightarrow (\exists w :: \mathit{pref}(w,k))$. $\Box$

According to the next lemma, $(\exists j :: \mathit{ALIVE}(Q,k,j))$ is an invariant. As a result, the computation of $\mathit{max}[k]$ and max in the Reader and Writer procedures, respectively, is well-defined.

Lemma 6: $(\exists j :: \mathit{ALIVE}(Q,k,j))$.

Proof: The given assertion is initially true, by the definition of the initial state. It could potentially be falsified only by an event of the form $w\!:\!5$ or $w\!:\!10$, where w is a k-Write operation. Consider an event e of this form, and let t be the state following e. Also, let v denote the initial k-Write operation, and assume

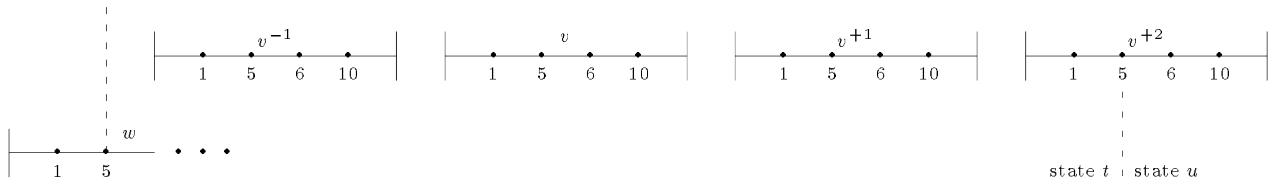

Figure 6: Proof of Lemma 7.

that v is a (k,j)-Write operation. We consider two cases, depending on whether *after*$(v\!:\!10)$ holds at t. If *after*$(v\!:\!10)$ does hold at t, then by Lemma 5, $t \models (\exists w :: alive(w,k))$. By the definition of *alive*, this implies that $t \models (\exists n :: ALIVE(Q,k,n))$.

Now, suppose that *after*$(v\!:\!10)$ does not hold at t. By our assumption concerning the initial Writes, v precedes all other k-Write operations. This implies that e is the event $v\!:\!5$. Because t is the state following $v\!:\!5$, $t \models \neg Q[k,j].done$. Because v precedes all other k-Write operations, by the definition of the initial state, $t \models (\forall n : n \neq j : Q[k,n].done \ \wedge\ \neg Q[k,n].flag \ \wedge (\forall i : 0 \leq i < 2W : Q[k,n].count[i] = 0))$. By the definition of $ALIVE$, this implies that $t \models (\forall n : n \neq j : ALIVE(Q,k,n))$. Because, by assumption, n ranges over $0 \leq n < 2W$, the range in this expression is not empty. Therefore, $t \models (\exists n :: ALIVE(Q,k,n))$. □

The next lemma applies to the interval of states for which $last(Q[k,j]) = w$ holds: if the expression $(\exists n : n \neq j : Q[k,n].done \ \wedge\ Q[k,n].count[j] = 3 \ \wedge\ Q[k,n].seq[j] = Q[k,j].seq[j])$ holds at some state in this interval, then it holds at all subsequent states in this interval. Note that this expression is true iff at least four successive operations of some Writer have read the same value from $Q[k,j].seq[j]$. When reading the proof of this lemma, the reader may wish to refer to Figure 6.

Lemma 7: $(\ last(Q[k,j]) = w \ \wedge\ (\exists n : n \neq j : Q[k,n].done \ \wedge\ Q[k,n].count[j] = 3 \ \wedge\ Q[k,n].seq[j] = Q[k,j].seq[j])\ )$ *unless* $(\ last(Q[k,j]) \neq w\ )$.

Proof: Let t and u be consecutive states such that $t \models last(Q[k,j]) = w$ and $u \models last(Q[k,j]) = w$, and assume that the following expression holds for some $n \neq j$.

$$t \models Q[k,n].done \ \wedge\ Q[k,n].count[j] = 3 \ \wedge\ Q[k,n].seq[j] = Q[k,j].seq[j] \qquad (3)$$

Our proof obligation is to show that there exists some $n' \neq j$ such that $u \models Q[k,n'].done \wedge Q[k,n'].count[j] = 3 \ \wedge\ Q[k,n'].seq[j] = Q[k,j].seq[j]$. Because $last(Q[k,j]) = w$ at both states t and u, by Lemma 1, $Q[k,j].seq[j]$ has the same value at both t and u. Therefore, if $Q[k,n]$ has the same value at both t and u, then by (3), our proof obligation is satisfied with $n' = n$. In the remainder of the proof, assume that the value of $Q[k,n]$ at u differs from its value at t.

Let v be the (k,n)-Write operation such that $t \models last(Q[k,n]) = v$. ($v$ exists because $Q[k,n].count[j]$ is initially 0.) By (3), $t \models Q[k,n].done$. Therefore, $t \models after(v\!:\!10)$. Because $t \models last(Q[k,n]) = v$, and because the value of $Q[k,n]$ at u differs from its value at t, this implies that u is reached from t via the occurrence of the event $v^{+2}\!:\!5$. Consider the Write operation v^{+1}. Note that v^{+1} is a (k,n')-Write operation, where $n' = n \oplus W$. In the remainder of the proof, we establish our proof obligation by showing that the following assertion holds.

$$(n' \neq j) \ \wedge \ (u \ \models \ Q[k,n'].done \ \wedge \ Q[k,n'].count[j] = 3 \ \wedge \ Q[k,n'].seq[j] = Q[k,j].seq[j])$$

Because u is reached from t via the occurrence of $v^{+2}\!:\!5$, and because v^{+1} and v^{+2} are successive operations of the same Writer, we have $u \ \models \ last(Q[k,n']) = v^{+1} \ \wedge \ after(v^{+1}\!:\!10)$. Therefore, because $u \ \models \ last(Q[k,j]) = w$, by Lemma 1, it suffices to prove the following.

$$n' \neq j \ \wedge \ v^{+1}!count[j] = 3 \ \wedge \ v^{+1}!seq[j] = w!seq[j] \tag{4}$$

Because $t \ \models \ last(Q[k,j]) = w \ \wedge \ last(Q[k,n]) = v$, by Lemma 1, $t \ \models \ Q[k,n].seq[j] = v!seq[j] \ \wedge \ Q[k,n].count[j] = v!count[j] \ \wedge \ Q[k,j].seq[j] = w!seq[j]$. Therefore, by (3),

$$v!count[j] = 3 \ \wedge \ v!seq[j] = w!seq[j] \quad . \tag{5}$$

Because $n \neq j$, and because $t \ \models \ last(Q[k,j]) = w \ \wedge \ last(Q[k,n]) = v$, by (3) and Lemma 3, Write operation v^{-1} exists and $w\!:\!5 \prec v^{-1}\!:\!1$. Because v^{-1}, v, v^{+1}, and v^{+2} are successive operations of the same Writer, $v^{-1}\!:\!1 \prec v^{+1}\!:\!1 \prec v^{+1}\!:\!10 \prec v^{+2}\!:\!5$. This establishes the following precedence assertion.

$$w\!:\!5 \prec v^{+1}\!:\!1 \prec v^{+1}\!:\!10 \prec v^{+2}\!:\!5 \tag{6}$$

By (6), $v^{+1} \neq w$. Therefore, because $u \ \models \ last(Q[k,n']) = v^{+1} \ \wedge \ last(Q[k,j]) = w$, we have $n' \neq j$. Hence, to establish (4), it suffices to show that $v^{+1}!count[j] = 3 \ \wedge \ v^{+1}!seq[j] = w!seq[j]$.

Let t' be the state prior to $v^{+1}\!:\!1$. Because $last(Q[k,j]) = w$ at state u (the state following $v^{+2}\!:\!5$), by (6), $t' \ \models \ last(Q[k,j]) = w$. Also, because v and v^{+1} are successive operations of the same Writer, $t' \ \models \ last(Q[k,n]) = v$. Therefore, by Lemma 1, $t' \ \models \ Q[k,j].seq[j] = w!seq[j] \ \wedge \ Q[k,n].seq[j] = v!seq[j] \ \wedge \ Q[k,n].count[j] = v!count[j]$. By (5), this implies that $t' \ \models \ Q[k,j].seq[j] = Q[k,n].seq[j] = w!seq[j] \ \wedge \ Q[k,n].count[j] = 3$. By statement 1 of the Writer procedure, $t' \ \models \ v^{+1}!y = Q$. Therefore,

$$v^{+1}!y[k,j].seq[j] = v^{+1}!y[k,n].seq[j] = w!seq[j] \ \wedge \ v^{+1}!y[k,n].count[j] = 3 \quad .$$

By statement 3 of the Writer procedure, $v^{+1}!seq[j] = v^{+1}!y[k,j].seq[j]$ and $(v^{+1}!seq[j] = v^{+1}!y[k,n].seq[j]) \Rightarrow v^{+1}!count[j] = min(3, v^{+1}!y[k,n].count[j]+1)$. It follows, then, that $v^{+1}!seq[j] = w!seq[j] \wedge v^{+1}!count[j] = 3$. $\Box$

According to the next lemma, if a completed Write operation is not "alive" at some state, then it is forever after not "alive."

Lemma 8: $\neg alive(w,k) \ \wedge \ after(w\!:\!10)$ is stable.

Proof: Let t and u be consecutive states such that $t \ \models \ \neg alive(w,k) \ \wedge \ after(w\!:\!10)$. By the definition of *after*, $u \ \models \ after(w\!:\!10)$. Thus, our proof obligation is to show that $u \ \models \ \neg alive(w,k)$.

Assume that w is a (k,j)-Write operation. By the definition of *alive*, if $u \models last(Q[k,j]) \neq w$, then $u \models \neg alive(w,k)$. So, assume that $u \models last(Q[k,j]) = w$. Because $after(w\!:\!10)$ holds at state u, we have

$$u \models last(Q[k,j]) = w \ \wedge \ after(w\!:\!10) \quad . \tag{7}$$

Because $u \models last(Q[k,j]) = w$, by the definition of *alive*, our proof obligation is to show that $ALIVE(Q,k,j)$ is false at state u.

Because t and u are consecutive states and $t \models after(w\!:\!10)$, by (7) and the text of the Writer procedure,

$$t \models last(Q[k,j]) = w \ \wedge \ after(w\!:\!10) \quad . \tag{8}$$

Because $t \models last(Q[k,j]) = w \wedge \neg alive(w,k)$, by the definition of *alive*, $t \models \neg ALIVE(Q,k,j)$. By (8) and Lemma 1, $t \models Q[k,j].done$. Therefore, by the definition of $ALIVE$, $t \models Q[k,j].flag$ or there exists n, where $n \neq j$, such that $t \models Q[k,n].done \wedge Q[k,n].count[j] = 3 \wedge Q[k,n].seq[j] = Q[k,j].seq[j]$. In the former case, by (7) and (8) and Lemma 1, we have $u \models Q[k,j].flag$. In the latter case, by (7), (8), and Lemma 7, there exists n', where $n' \neq j$, such that $u \models Q[k,n'].done \wedge Q[k,n'].count[j] = 3 \wedge Q[k,n'].seq[j] = Q[k,j].seq[j]$. Therefore, in either case, $u \models \neg ALIVE(Q,k,j)$. □

The following lemma considers a (k,j)-Write operation v and a (k,n)-Write operation w. According to this lemma, if v assigns the value false to its private variable $flag$ (indicating that its sequence number has not yet been copied three times by any other Writer), and if w is the third of three successive operations of the same Writer to read the value $v!seq[j]$ from $Q[k,j].seq[j]$, then v does not read the value assigned to $Q[k,n]$ by w when computing $flag$.

Lemma 9: Let v be a (k,j)-Write operation, and let w be a (k,n)-Write operation. If $\neg v!flag \ \wedge \ w!seq[j] = v!seq[j] \ \wedge \ w!count[j] \geq 2$, then $last(Q[k,n]) \neq w$ at the state prior to $v\!:\!6$.

Proof: Let v and w be as defined in the lemma, and let t be the state prior to $v\!:\!6$. Assume that $\neg v!flag \wedge w!seq[j] = v!seq[j] \ \wedge w!count[j] \geq 2$. Our proof obligation is to show that $t \models last(Q[k,n]) \neq w$. Assume, to the contrary, that $t \models last(Q[k,n]) = w$. Then, by Lemma 1, $t \models Q[k,n].count[j] = w!count[j] \ \wedge \ Q[k,n].seq[j] = w!seq[j]$. Because $w!seq[j] = v!seq[j] \ \wedge \ w!count[j] \geq 2$, this implies that $t \models Q[k,n].count[j] \geq 2 \ \wedge \ Q[k,n].seq[j] = v!seq[j]$. Because t is the state prior to $v\!:\!6$, $t \models Q = v!z$. Therefore, $v!z[k,n].count[j] \geq 2 \ \wedge \ v!z[k,n].seq[j] = v!seq[j]$. Because $w!count[j]$ is nonzero, by statement 4 of the Writer procedure, w is not a (k,j)-Write operation, i.e., $j \neq n$. This implies that $v!flag$ is true, which is a contradiction. Thus, our assumption that $t \models last(Q[k,n]) = w$ is false. □

In the following two lemmas, we consolidate a number of simple properties that will be used repeatedly in the lemmas that follow.

Lemma 10: Let w be a (k,j)-Write operation. Then, $pref(w,k) \ \Rightarrow \ alive(w,k)$ and $alive(w,k) \ \Rightarrow \ last(Q[k,j]) = w \ \wedge \ ALIVE(Q,k,j) \ \wedge \ after(w\!:\!10)$.

Proof: Let w be a (k,j)-Write operation. By the definition of *pref*, $pref(w,k) \;\Rightarrow\; alive(w,k)$. By the definition of *alive*, $alive(w,k) \;\Rightarrow\; last(Q[k,j]) = w \;\wedge\; ALIVE(Q,k,j)$. By the definition of *ALIVE*, $ALIVE(Q,k,j) \;\Rightarrow\; Q[k,j].done$. By the text of the Writer procedure, $last(Q[k,j]) = w \;\wedge\; Q[k,j].done \;\Rightarrow$ $after(w\!:\!10)$. □

Lemma 11: For each k-Write operation w, $w!tag > 0$. Also, if w is the initial k-Write operation, then $w!tag = 1$ and $pref(w,k)$ holds at the state following $w\!:\!10$.

Proof: Let w be a k-Write operation, and let $m = w!max$. (Note that Lemma 6 implies that $w!max$ is well-defined.) It can be shown that each *tag* field in the construction is always nonnegative. Therefore, $w!tag = w!z[k,m].tag + 1 > 0$. Now, suppose that w is the initial k-Write operation, and assume that w is a (k,j)-Write operation. By our assumption concerning the initial Writes, w precedes each other k-Write operation. By the definition of the initial state, each *tag* field in Q is initially 0. It follows, then, that $w!tag = 1$. Moreover, the following expression holds at the state following $w\!:\!10$.

$$\begin{aligned} &last(Q[k,j]) = w \;\wedge\; Q[k,j].done \;\wedge\; \neg Q[k,j].flag \;\wedge \\ &\quad (\forall n : n \neq j : Q[k,n].count[j] = 0 \;\wedge\; Q[k,j].tag > Q[k,n].tag) \end{aligned}$$

This implies that $pref(w,k)$ holds. □

The next lemma gives the conditions under which $alive(v,k)$ may be falsified. The two cases of the lemma are illustrated in Figures 7(a) and 7(b).

Lemma 12: Suppose that t and u are consecutive states such that $t \models alive(v,k)$ and $u \models \neg alive(v,k)$. Then, there exists a k-Write operation w such that $u \models after(w\!:\!10)$, and $v\!:\!10 \prec w\!:\!0$ or $v\!:\!6 \prec w^{-1}\!:\!5$. (Note that, in either case, $v\!:\!6 \prec w\!:\!5$.)

Proof: Let t, u, and v be as defined in the statement of the lemma. Assume that v is a (k,j)-Write operation. We first dispose of the case in which $u \models last(Q[k,j]) \neq v$. Because $t \models alive(v,k)$, by the definition of *alive*, $t \models last(Q[k,j]) = v$. Because t and u are consecutive states and $last(Q[k,j]) = v$ holds at t but not u, u is reached from t via the occurrence of the event $v^{+2}\!:\!5$. This is illustrated in Figure 7(a). Consider the Write operation v^{+1}. Because $v^{+2}\!:\!5$ is the event prior to u, $u \models after(v^{+1}\!:\!10)$. Because v and v^{+1} are successive operations of the same Writer, $v\!:\!10 \prec v^{+1}\!:\!0$. Therefore, letting $w = v^{+1}$, our proof obligation is satisfied.

In the remainder of the proof, we assume that $u \models last(Q[k,j]) = v$. This case is illustrated in Figure 7(b). Because $t \models alive(v,k)$, by Lemma 10 and the definition of *ALIVE*, $t \models last(Q[k,j]) = v \wedge after(v\!:\!10) \wedge Q[k,j].done \wedge \neg Q[k,j].flag$. Because t and u are consecutive states and $t \models after(v\!:\!10)$, we have $u \models after(v\!:\!10)$. Because $last(Q[k,j]) = v \wedge after(v\!:\!10)$ holds at both states t and u, by Lemma 1, $Q[k,j]$ has the same value at both t and u. Therefore, $u \models last(Q[k,j]) = v \wedge Q[k,j].done \wedge \neg Q[k,j].flag$. Because $alive(v,k)$ does not hold at u, this implies that there exists n, where $n \neq j$, such that the following

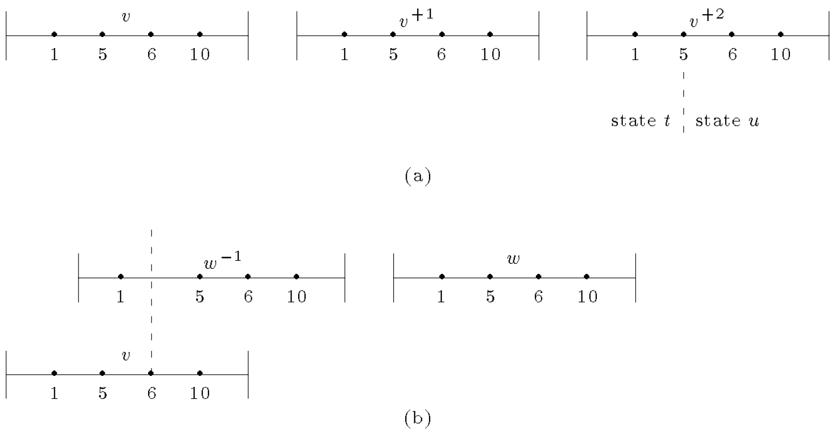

Figure 7: Proof of Lemma 12.

expression holds.

$$u \models Q[k,n].done \ \wedge \ Q[k,n].count[j] = 3 \ \wedge \ Q[k,n].seq[j] = Q[k,j].seq[j] \tag{9}$$

Let w be the Write operation such that $u \models last(Q[k,n]) = w$. ($w$ exists because $Q[k,n].count[j]$ is initially 0.) Because $u \models last(Q[k,n]) = w \ \wedge \ last(Q[k,j]) = v$, by (9) and Lemma 3, the Write operation w^{-1} exists and $w^{-1}!seq[j] = v!seq[j]$. Moreover, because $u \models after(v{:}10) \ \wedge \ \neg Q[k,j].flag$, by (9) and Lemma 1, $\neg v!flag \ \wedge \ w!seq[j] = v!seq[j] \ \wedge \ w!count[j] = 3$. Because w^{-1} and w are successive operations of the same Writer and $w!count[j] = 3$, we have $w^{-1}!count[j] \geq 2$. This establishes the following assertion.

$$w^{-1}!count[j] \geq 2 \ \wedge \ w!count[j] = 3 \ \wedge \ w^{-1}!seq[j] = w!seq[j] = v!seq[j] \ \wedge \ \neg v!flag \ . \tag{10}$$

Because $u \models last(Q[k,n]) = w \ \wedge \ Q[k,n].done$, we have $u \models after(w{:}10)$. Therefore, we can meet our proof obligation by showing that $v{:}6 \prec w^{-1}{:}5$. Assume, to the contrary, that $w^{-1}{:}5 \prec v{:}6$. Note that w^{-1} is a (k,n')-Write operation, where $n' = n \oplus W$. Let t' be the state prior to $v{:}6$, and let e be the event prior to state u. Because $u \models after(v{:}10)$, we have $v{:}6 \prec e$. Consider the event $w{:}5$. Either $w{:}5 \prec v{:}6$ or $v{:}6 \prec w{:}5$. In the former case, we have $w{:}5 \prec v{:}6 \prec e$. Because $last(Q[k,n]) = w$ at state u, this implies that $t' \models last(Q[k,n]) = w$. In the latter case, we have $w^{-1}{:}5 \prec v{:}6 \prec w{:}5$. Because w^{-1} and w are successive operations of the same Writer, this implies that $t' \models last(Q[k,n']) = w^{-1}$. Because $t' \models last(Q[k,n]) = w \ \vee \ last(Q[k,n']) = w^{-1}$, by (10) and Lemma 9, we have a contradiction. Thus, our assumption that $w^{-1}{:}5 \prec v{:}6$ is false, i.e., $v{:}6 \prec w^{-1}{:}5$. □

The next lemma considers a (k,j)-Write operation w; this lemma specifies an assertion over the fields of Q that holds whenever $pref(w,k)$ does.

Lemma 13: Let w be a (k,j)-Write operation. Then, $pref(w,k) \Rightarrow Q[k,j].val = w!val \ \wedge \ Q[k,j].tag = w!tag \wedge Q[k,j].phi = w!phi1 \wedge ALIVE(Q,k,j) \wedge (\forall n : ALIVE(Q,k,n) : (Q[k,n].tag, n) \leq (Q[k,j].tag, j))$.

Proof: Let w be a (k,j)-Write operation, and suppose that $\mathit{pref}(w,k)$ holds at some state t. Then, by Lemma 10, $t \models \mathit{last}(Q[k,j]) = w \ \wedge\ \mathit{after}(w\!:\!10) \ \wedge\ \mathit{ALIVE}(Q,k,j)$. Therefore, by Lemma 1,

$$t \models Q[k,j].\mathit{val} = w!\mathit{val} \ \wedge\ Q[k,j].\mathit{tag} = w!\mathit{tag} \ \wedge\ Q[k,j].\mathit{phi} = w!\mathit{phi1} \quad . \tag{11}$$

Let $n \neq j$ and suppose that $t \models \mathit{ALIVE}(Q,k,n)$. Our remaining proof obligation is to show that $t \models (Q[k,n].\mathit{tag},n) \leq (Q[k,j].\mathit{tag},j)$. We first dispose of the case $t \models Q[k,n].\mathit{tag} = 0$. By Lemma 11, $w!\mathit{tag} > 0$. Thus, in this case, by (11), $t \models (Q[k,n].\mathit{tag},n) \leq (Q[k,j].\mathit{tag},j)$.

Now, consider the case $t \models Q[k,n].\mathit{tag} \neq 0$. In this case, the value of $Q[k,n].\mathit{tag}$ at state t differs from its initial value. Therefore, there exists a k-Write operation v such that $t \models \mathit{last}(Q[k,n]) = v$. Because $t \models \mathit{last}(Q[k,n]) = v \ \wedge\ \mathit{ALIVE}(Q,k,n)$, we have $t \models \mathit{alive}(v,k)$. Therefore, because $t \models \mathit{pref}(w,k)$, $(v!\mathit{tag},n) \leq (w!\mathit{tag},j)$. Also, by Lemma 10 and Lemma 1, $t \models Q[k,n].\mathit{tag} = v!\mathit{tag}$. Hence, by (11), $t \models (Q[k,n].\mathit{tag},n) \leq (Q[k,j].\mathit{tag},j)$. □

The next lemma shows that the "best" tag value for a component does not decrease from state to state. It also gives a necessary condition for an increase to occur. This condition will be used later when we show that the *tag* fields can be bounded.

Lemma 14: Let t and u be consecutive states such that $t \models \mathit{pref}(v,k)$ and $u \models \mathit{pref}(w,k)$. Then, either $w!\mathit{tag} = v!\mathit{tag}$ or $w!\mathit{tag} = v!\mathit{tag} + 1$, and in the latter case, u is reached from t via the occurrence of event $w\!:\!10$.

Proof: Let t and u be consecutive states such that $t \models \mathit{pref}(v,k)$ and $u \models \mathit{pref}(w,k)$. It suffices to prove that the lemma holds for states t and u, given the assumption that the lemma holds for the prefix of the given history ending with state t. Let e be the event prior to state u. Our proof obligation is to show that either $w!\mathit{tag} = v!\mathit{tag}$ or $w!\mathit{tag} = v!\mathit{tag} + 1$, and in the latter case, $e = w\!:\!10$.

We first show that if $w!\mathit{tag} = v!\mathit{tag} + 1$, then $e = w\!:\!10$. In this case, because $\mathit{pref}(v,k)$ holds at t, $\mathit{alive}(w,k)$ does not hold at t. Because $u \models \mathit{pref}(w,k)$, by Lemma 10, $u \models \mathit{alive}(w,k) \ \wedge\ \mathit{after}(w\!:\!10)$. Because t and u are consecutive states such that $t \models \neg\mathit{alive}(w,k)$ and $u \models \mathit{alive}(w,k) \ \wedge\ \mathit{after}(w\!:\!10)$, by Lemma 8, $t \models \neg\mathit{after}(w\!:\!10)$. Therefore, because $\mathit{after}(w\!:\!10)$ is false at t but true at u, u is reached from t via the occurrence of the event $w\!:\!10$.

Our remaining proof obligation is to show that $v!\mathit{tag} \leq w!\mathit{tag} \leq v!\mathit{tag} + 1$. We first prove that $w!\mathit{tag} \leq v!\mathit{tag} + 1$. Observe that if w is the initial k-Write operation, then by Lemma 11, $w!\mathit{tag} = 1 < v!\mathit{tag} + 1$. So, assume that w is not the initial k-Write operation. Let u' be the state prior to the event $w\!:\!6$. Because w is not the initial k-Write operation, the initial k-Write operation precedes w. Therefore, by the corollary to Lemma 5, there exists a k-Write operation v' such that $u' \models \mathit{pref}(v',k)$. By Lemma 13 and the text of the Writer procedure, this implies that $w!\mathit{tag} = v'!\mathit{tag} + 1$. Because $u \models \mathit{pref}(w,k)$, by Lemma 10, $u \models \mathit{after}(w\!:\!10)$. Because t and u are consecutive states, this implies that u' occurs before t. Because $t \models \mathit{pref}(v,k)$, and because the lemma holds for the prefix of the given history ending with state t, this implies that $v'!\mathit{tag} \leq v!\mathit{tag}$. (Note that the corollary to Lemma 5 implies that *pref* is well-defined for all states between u' and t.) Therefore, $w!\mathit{tag} \leq v!\mathit{tag} + 1$.

Our final proof obligation is to show that $v!tag \leq w!tag$. If $u \models alive(v,k)$, then, because $u \models pref(w,k)$, we have $v!tag \leq w!tag$. Also, if v is the initial k-Write operation, then, by Lemma 11, $v!tag = 1 \leq w!tag$. In the remainder of the proof, assume that v is not the initial k-Write operation and that $u \models \neg alive(v,k)$.

Because $t \models pref(v,k)$, by Lemma 10, $t \models alive(v,k)$. Therefore, by Lemma 12, there exists a k-Write operation w' such that $v\!:\!6 \prec w'\!:\!5$ and $u \models after(w'\!:\!10)$. Because $u \models after(w'\!:\!10)$, by Lemma 5, there exists a k-Write operation w'' such that

$$u \models alive(w'',k) \;\wedge\; (\forall q : q \text{ is a } k\text{-Write} \;\wedge\; w''\!:\!5 \prec q\!:\!5 : \neg after(q\!:\!10)) \quad . \tag{12}$$

Because $u \models alive(w'',k)$, by Lemma 10, $u \models after(w''\!:\!10)$. This implies that $w''\!:\!10 \preceq e$. Also, because $u \models after(w'\!:\!10)$, by (12), we have $w'\!:\!5 \preceq w''\!:\!5$. Therefore,

$$v\!:\!6 \prec w'\!:\!5 \preceq w''\!:\!5 \prec w''\!:\!10 \preceq e \quad .$$

Let t' be the state prior to $v\!:\!6$ and let t'' be the state prior to $w''\!:\!6$. Notice that the above precedence assertion implies that t'' occurs between t' and t. Because v is not the initial k-Write operation, by our assumption concerning the initial Writes, the initial k-Write operation precedes both v and w''. Hence, by the corollary to Lemma 5, there exist Write operations v' and v'' such that $t' \models pref(v',k)$ and $t'' \models pref(v'',k)$. By Lemma 13 and the text of the Writer procedure, $v!tag = v'!tag + 1$, and $w''!tag = v''!tag + 1$. Moreover, because the lemma holds for the prefix of the given history ending with state t, $v'!tag \leq v''!tag$. Therefore, $v!tag \leq w''!tag$. By (12), $u \models alive(w'',k)$. Therefore, because $u \models pref(w,k)$, we have $w''!tag \leq w!tag$. Consequently, by transitivity, $v!tag \leq w!tag$. □

The next lemma directly follows from the previous one.

Lemma 15: If $v\!:\!6 \prec w\!:\!6$, then $v!tag \leq w!tag$.

Proof: If v is the initial k-Write operation, then by Lemma 11, $v!tag = 1 \leq w!tag$. In the remainder of the proof, assume that v is not the initial k-Write operation. Then, by our assumption concerning the initial Writes, the initial k-Write operation precedes both v and w. Hence, by the corollary to Lemma 5, there exists k-Write operations p and q such that $pref(p,k)$ holds at the state prior to $v\!:\!6$, and $pref(q,k)$ holds at the state prior to $w\!:\!6$. By Lemma 13 and the text of the Writer procedure, $v!tag = p!tag + 1$ and $w!tag = q!tag + 1$. Because $v\!:\!6 \prec w\!:\!6$, by Lemma 14, $p!tag \leq q!tag$. Therefore, $v!tag \leq w!tag$. □

The following lemma shows that the value of the "best" tag/process identifier pair does not decrease from state to state.

Lemma 16: Let t and u be consecutive states such that $t \models pref(v,k)$ and $u \models pref(v',k)$. Then, $(v!tag, v!i) \leq (v'!tag, v'!i)$.

Proof: Let t, u, v, and v' be as defined in the lemma. If $u \models alive(v,k)$, then because $u \models pref(v',k)$, we have $(v!tag, v!i) \leq (v'!tag, v'!i)$. Thus, in this case, our proof obligation is satisfied. In the remainder of the proof, assume that $u \models \neg alive(v,k)$.

Because $u \models pref(v',k)$, by Lemma 10, $u \models after(v'\!:\!10)$. Hence, by Lemma 5, there exists a k-Write operation w such that

$$u \models alive(w,k) \ \wedge\ (\forall p : p \text{ is a } k\text{-Write} \ \wedge\ w\!:\!5 \prec p\!:\!5 : \neg after(p\!:\!10)) \ . \tag{13}$$

Because $u \models alive(w,k) \wedge pref(v',k)$, by the definition of $pref$, $w!tag \leq v'!tag$. Therefore, we can establish our proof obligation by showing that $v!tag < w!tag$.

Because $t \models pref(v,k)$, by Lemma 10, $t \models alive(v,k)$. Because t and u are consecutive states and $alive(v,k)$ holds at t but not u, by Lemma 12, there exists a k-Write operation q such that $u \models after(q\!:\!10)$, and either $v\!:\!10 \prec q\!:\!0$ or $v\!:\!6 \prec q^{-1}\!:\!5$. Because $u \models after(q\!:\!10)$, by (13), $q\!:\!5 \preceq w\!:\!5$. Therefore, either $v\!:\!10 \prec w\!:\!5$ or $v\!:\!6 \prec q^{-1}\!:\!5 \prec q^{-1}\!:\!10 \prec q\!:\!5 \preceq w\!:\!5$. We consider these two cases separately.

First, suppose that $v\!:\!10 \prec w\!:\!5$. Let e be the event prior to state u, i.e., $t \xrightarrow{e} u$. By (13) and Lemma 10, $u \models after(w\!:\!10)$. Hence, $w\!:\!10 \preceq e$. This implies that $v\!:\!10 \prec w\!:\!5 \prec w\!:\!6 \prec w\!:\!10 \preceq e$. Hence, because $alive(v,k)$ holds at state t, i.e., the state prior to e, by Lemma 8, $alive(v,k)$ also holds at the state prior to $w\!:\!6$. Therefore, letting $j = v!i$, by Lemma 10 and Lemma 1, $ALIVE(Q,k,j) \ \wedge\ Q[k,j].tag = v!tag$ holds at that state. This implies that w chooses a larger tag value than v, i.e., $w!tag > v!tag$.

Now, consider the other case mentioned above, i.e., $v\!:\!6 \prec q^{-1}\!:\!5 \prec q^{-1}\!:\!10 \prec q\!:\!5 \preceq w\!:\!5$. Let t' be the state prior to $w\!:\!6$. Observe that $t' \models after(q^{-1}\!:\!10)$. Therefore, by Lemma 5, there exists a k-Write operation w' such that

$$t' \models alive(w',k) \ \wedge\ (\forall p : p \text{ is a } k\text{-Write} \ \wedge\ w'\!:\!5 \prec p\!:\!5 : \neg after(p\!:\!10)) \ . \tag{14}$$

Let $n = w'!i$. Because $t' \models alive(w',k)$, by Lemma 10 and Lemma 1, $ALIVE(Q,k,n) \wedge Q[k,n].tag = w'!tag$. This implies that w chooses a larger tag value than w', i.e., $w!tag > w'!tag$. Because $t' \models after(q^{-1}\!:\!10)$, by (14), $q^{-1}\!:\!5 \preceq w'\!:\!5$. This implies that $v\!:\!6 \prec q^{-1}\!:\!5 \preceq w'\!:\!5 \prec w'\!:\!6$. Because $v\!:\!6 \prec w'\!:\!6$, by Lemma 15, $v!tag \leq w'!tag$. Therefore, by transitivity, $v!tag < w!tag$. □

According to the next lemma, if a completed Write operation is not "preferable" at some state, then it is forever after not "preferable."

Lemma 17: $\neg pref(w,k) \wedge after(w\!:\!10)$ is stable.

Proof: Let t and u be consecutive states such that $t \models \neg pref(w,k) \wedge after(w\!:\!10)$. By the definition of $after$, $u \models after(w\!:\!10)$. Thus, our proof obligation is to show that $u \models \neg pref(w,k)$. If $alive(w,k)$ is false at u, then by Lemma 10, $u \models \neg pref(w,k)$. So, assume that $alive(w,k)$ holds at u. By Lemma 8, this implies that $alive(w,k)$ holds at t as well.

Because $after(w\!:\!10)$ holds at both states t and u, by the corollary to Lemma 5, there exist k-Write operations v and v' such that $t \models pref(v,k)$ and $u \models pref(v',k)$. Because $t \models alive(w,k) \ \wedge\ \neg pref(w,k)$, by the definition of $pref$, $(w!tag, w!i) < (v!tag, v!i)$. Because t and u are consecutive states, by Lemma 16, $(v!tag, v!i) \leq (v'!tag, v'!i)$. Therefore, by transitivity, $(w!tag, w!i) < (v'!tag, v'!i)$. Hence, because

$u \models pref(v', k)$, we have, $u \models \neg pref(w, k)$. □

The next lemma is used in the proof of Proximity to show that if Read operation r returns the input value of k-Write operation w, then no other k-Write operation succeeds w and precedes r.

Lemma 18: Suppose that $pref(w, k)$ holds at the state prior to $r{:}0$. Then, for each k-Write operation w' that differs from w, $w'{:}0 \prec w{:}10$ or $r{:}0 \prec w'{:}10$.

Proof: Let r and w be as defined in the lemma. Assume that w is a (k, j)-Write operation. Let t be the state prior to $r{:}0$. Because $t \models pref(w, k)$, by Lemma 10, $t \models alive(w, k) \wedge last(Q[k, j]) = w \wedge after(w{:}10)$. This implies that $w{:}10 \prec r{:}0$. We establish our proof obligation by assuming, to the contrary, that there exists a (k, n)-Write operation w' (that differs from w) such that

$$w{:}10 \prec w'{:}0 \prec w'{:}10 \prec r{:}0 \quad . \tag{15}$$

Because $last(Q[k, j]) = w$ at state t, this precedence assertion implies that $j \neq n$.

By (15), $t \models after(w'{:}10)$. Hence, by Lemma 5, there exists a k-Write operation v such that

$$t \models alive(v, k) \wedge (\forall p : p \text{ is a } k\text{-Write} \wedge v{:}5 \prec p{:}5 : \neg after(p{:}10)) \quad . \tag{16}$$

We now show that $v!tag > w!tag$. Because $t \models after(w'{:}10)$, (16) implies that $w'{:}5 \preceq v{:}5$. Therefore, by (15), $w{:}10 \prec v{:}5$. Because $t \models alive(v, k)$, by Lemma 10, $t \models after(v{:}10)$. Hence, because t is the state prior to $r{:}0$, $v{:}10 \prec r{:}0$. Therefore, $w{:}10 \prec v{:}5 \prec v{:}10 \prec r{:}0$. Because $alive(w, k)$ holds at state t, by Lemma 8, this implies that $alive(w, k)$ holds at the state prior to $v{:}6$. Therefore, by Lemma 10 and Lemma 1, $ALIVE(Q, k, j) \wedge Q[k, j].tag = w!tag$ holds at that state. This implies that v chooses a larger tag value than w, i.e., $v!tag > w!tag$.

Because $t \models alive(v, k)$ and $v!tag > w!tag$, by the definition of $pref$, $t \models \neg pref(w, k)$. But, by the statement of the lemma, $t \models pref(w, k)$. Therefore, we have a contradiction. Hence, our assumption that there exists w' such that $w{:}10 \prec w'{:}0 \prec w'{:}10 \prec r{:}0$ is false. □

According to the next lemma, for each Read operation r there exists a preceding k-Write operation w that is "preferable" when r reads from Q. As shown in the proof of Integrity, the output value of r for component k equals the input value of w.

Lemma 19: Let r be a Read operation. Then, there exists a k-Write operation w such that $w{:}10 \prec r{:}0$ and $pref(w, k)$ holds at each state between $w{:}10$ and $r{:}0$.

Proof: Let r be a Read operation and let t be the state prior to the event $r{:}0$. Let $j = r!max[k]$. (Lemma 6 implies that $r!max[k]$ is well-defined.) Then, by the text of the Reader procedure,

$$t \models ALIVE(Q, k, j) \wedge (\forall n : ALIVE(Q, k, n) : (Q[k, n].tag, n) \leq (Q[k, j].tag, j)) \quad . \tag{17}$$

We now show that the value of $Q[k, j]$ at state t differs from its initial value. By our assumption concerning the initial Writes, there exists a k-Write operation that precedes r. By Lemma 5, this implies that there exists a (k,l)-Write operation p such that $t \models alive(p,k)$. By Lemma 10 and Lemma 1, $t \models ALIVE(Q,k,l) \wedge Q[k,l].tag = p!tag$. By Lemma 11, $p!tag > 0$. Therefore, by (17), $t \models Q[k,j].tag \geq Q[k,l].tag > 0$. Thus, the value of $Q[k,j].tag$ at state t differs from its initial value.

This implies that $t \models last(Q[k,j]) = w$ for some k-Write operation w. By (17), $t \models alive(w,k)$. We now show that $t \models pref(w,k)$. Consider a (k,n)-Write operation v such that $t \models alive(v,k)$. Our proof obligation is to show that $(v!tag, n) \leq (w!tag, j)$. Because $t \models alive(v,k)$, by Lemma 10 and Lemma 1, $t \models ALIVE(Q,k,n) \wedge Q[k,n].tag = v!tag$. Similarly, because $t \models alive(w,k)$, we have $t \models ALIVE(Q,k,j) \wedge Q[k,j].tag = w!tag$. Consequently, by (17), $(v!tag, n) \leq (w!tag, j)$.

Because $t \models alive(w,k)$, by Lemma 10, $after(w{:}10)$ holds at state t (i.e., the state prior to $r{:}0$). Therefore, $w{:}10 \prec r{:}0$. Because $pref(w,k)$ holds at state t, by Lemma 17, $pref(w,k)$ holds at each state between $w{:}10$ and $r{:}0$. This establishes our proof obligation. $\Box$

We now use the preceding lemmas to establish the correctness of the construction.

Theorem 1: Each well-formed history of the construction is linearizable.

Proof: We establish the theorem by proving that the five conditions of the Shrinking Lemma are satisfied.

Uniqueness: Uniqueness is satisfied since the shared auxiliary variable $P[k]$ is atomically incremented whenever a k-Write operation assigns its value to either private variable *phi0* or *phi1*. $\Box$

Integrity: Consider a Read operation r. By Lemma 19, there exists a k-Write operation w such that $w{:}10 \prec r{:}0$ and $pref(w,k)$ holds at each state between $w{:}10$ and $r{:}0$. Assume that w is a (k,j)-Write operation, and let t be the state prior to $r{:}0$. Because $t \models pref(w,k)$, by Lemma 13 and the text of the Reader procedure, $r!val[k] = w!val$ and $r!phi[k] = w!phi1$. By the definition of ϕ_k, we have $\phi_k(r) = r!phi[k]$. Also, because $pref(w,k)$ holds at the state following $w{:}10$, $\phi_k(w) = w!phi1$. Hence, $\phi_k(r) = \phi_k(w)$. $\Box$

Proximity: Let r be a Read operation and let v be a k-Write operation. We prove that Proximity is satisfied by proving the stronger result $r{:}0 \prec v{:}0 \Rightarrow \phi_k(r) < \phi_k(v)$ and $v{:}10 \prec r{:}0 \Rightarrow \phi_k(v) \leq \phi_k(r)$.

Let t denote the state prior to the event $r{:}0$, let u denote the state prior to the event $v{:}0$, and let u' denote the state prior to the event $v{:}10$.

Case 1: $r{:}0 \prec v{:}0$. By the definition of ϕ_k, either $u \models \phi_k(v) = P[k]$ or $u' \models \phi_k(v) = P[k]$. Because $r{:}0 \prec v{:}0$, state t occurs before both states u and u'. Notice that a Write operation only changes the value of $P[k]$ by atomically incrementing it; thus, the value of $P[k]$ at either state u or u' is at least the value of $P[k]$ at state t. Therefore, $t \models P[k] \leq \phi_k(v)$.

By Lemma 6, the text of the Reader procedure, and the definition of ϕ_k, there exists j such that $t \models Q[k,j].phi = \phi_k(r)$. Because $P[k]$ is incremented atomically when its value is assigned by a Write operation to either of its private variables *phi0* or *phi1*, $t \models Q[k,j].phi < P[k]$. Therefore, by transitivity, $\phi_k(r) < \phi_k(v)$.

Case 2: $v\!:\!10 \prec r\!:\!0$. By Lemma 19, there exists a k-Write operation w such that $w\!:\!10 \prec r\!:\!0$ and $pref(w,k)$ holds at each state between $w\!:\!10$ and $r\!:\!0$. Moreover, by the proof of Integrity, $\phi_k(r) = \phi_k(w) = w!phi1$.

We establish our proof obligation by showing that $\phi_k(v) \leq \phi_k(w)$. If $v = w$, then the result trivially holds, so assume that $v \neq w$. Then, by Lemma 18, $v\!:\!0 \prec w\!:\!10$. If $v\!:\!10 \prec w\!:\!10$, then $v!phi0 < w!phi1$ and $v!phi1 < w!phi1$; thus, by the definition of ϕ_k, $\phi_k(v) < \phi_k(w)$. Now consider the other case, i.e., $v\!:\!0 \prec w\!:\!10 \prec v\!:\!10 \prec r\!:\!0$. Because $pref(w,k)$ holds for all states between $w\!:\!10$ and $r\!:\!0$, this precedence assertion implies that $pref(v,k)$ is false at the state following $v\!:\!10$. Hence, by the definition of ϕ_k, $\phi_k(v) = v!phi0$. Because $v\!:\!0 \prec w\!:\!10$, $v!phi0 < w!phi1$. Therefore, $\phi_k(v) < \phi_k(w)$. $\Box$

Read Precedence: Let r and s be two Read operations. We prove that Read Precedence holds by proving $r\!:\!0 \prec s\!:\!0 \;\Rightarrow\; (\forall k :: \phi_k(r) \leq \phi_k(s))$. Assume that $r\!:\!0 \prec s\!:\!0$. By Lemma 19, there exists a k-Write operation w such that $w\!:\!10 \prec r\!:\!0$ and $pref(w,k)$ holds at each state between $w\!:\!10$ and $r\!:\!0$. By transitivity, $w\!:\!10 \prec s\!:\!0$. By the proof of Proximity, this implies that $\phi_k(w) \leq \phi_k(s)$. By the proof of Integrity, $\phi_k(w) = \phi_k(r)$. Therefore, $\phi_k(r) \leq \phi_k(s)$. $\Box$

Write Precedence: Let r be a Read operation, let v be a j-Write operation, and let w be a k-Write operation. Assume that v precedes w and $\phi_k(w) \leq \phi_k(r)$. In the proof of Proximity, we showed that $r\!:\!0 \prec w\!:\!0 \;\Rightarrow\; \phi_k(r) < \phi_k(w)$. By the contrapositive of this expression and by our assumption that $\phi_k(w) \leq \phi_k(r)$, we conclude that $w\!:\!0 \prec r\!:\!0$. Because v precedes w, $v\!:\!10 \prec w\!:\!0$. Thus, by transitivity, $v\!:\!10 \prec r\!:\!0$. By the proof of Proximity, this implies that $\phi_j(v) \leq \phi_j(r)$. $\Box$

4.4 Bounding the Tags

In this section, we show that is possible to bound the size of the *tag* fields. As seen in Figure 3, a Read or Write operation compares the *tag* fields of two different elements of Q only if both elements are alive. In what follows, we show that the *tag* fields of the alive elements of a particular component are within some bounded range, particularly a range of size $4W$. Based on this, we then explain how to obtain a construction that uses only bounded variables. We establish the former by proving that the following expression holds.

$$(ALIVE(Q,k,i) \;\wedge\; ALIVE(Q,k,j)) \;\Rightarrow\; (|Q[k,i].tag - Q[k,j].tag| \leq 4W-1)$$

(It is actually possible to prove a slightly tighter bound, at the expense of a somewhat longer proof.) Therefore, if the smallest *tag* field among the alive elements for some component is b, then the *tag* fields for these elements lie within the range $b, \ldots, b+4W-1$. As explained below, this implies that we can restrict the size of each *tag* field to range over $0..8W-2$.

The following lemma is used in the proof. This lemma gives us means for determining how much the "best" tag value for a given component can increase over an interval of states.

Lemma 20: Let v and v' be k-Write operations such that $t \models pref(v,k)$ and $u \models pref(v',k)$, where state t either equals or occurs before state u. Furthermore, suppose that $Q[k,j].seq[j]$ has the same value at each state in the closed interval $[t,u]$. If $u \models ALIVE(Q,k,j)$, then $v'!tag \leq v!tag + 4W - 2$.

Proof: Let t, u, v, and v' be as defined in the statement of the lemma. Assume that $Q[k,j].seq[j]$ has the same value at every state in the closed interval $[t,u]$ and that $u \models ALIVE(Q,k,j)$. Let D denote the number of events between t and u of the form $p\!:\!10$, where p is a k-Write operation. Then, by Lemma 14, $v'!tag \leq v!tag + D$. Therefore, it suffices to show that $D \leq 4W - 2$.

Let $j' = j \oplus W$. By the text of the Writer procedure, if p is a (k,j)-Write operation, then the value of $Q[k,j].seq[j]$ at the state prior to $p\!:\!5$ differs from its value at the state following $p\!:\!5$. Hence, because $Q[k,j].seq[j]$ has the same value at every state in $[t,u]$, there are no events between t and u of the form $p\!:\!5$, where p is a (k,j)-Write operation. Because successive operations of the same Writer write to different elements of Q, this implies that between t and u there is at most one event $p\!:\!10$, where p is a (k,j)-Write operation, and at most one such event, where p is a (k,j')-Write operation.

Let $n \neq j \ \wedge \ n \neq j'$, and let $n' = n \oplus W$. In the remainder of the proof, we use q to denote an arbitrary (k,n)- or (k,n')-Write operation. We show that there are at most four events of the form $q\!:\!10$ between t and u. Assume, to the contrary, that there are at least five such events between t and u, and let $w\!:\!10$ be the last such event. Let e be the event following state t, and let f be the event prior to state u. Then, because there are at least five events of the form $q\!:\!10$ between t and u, the following precedence assertion holds.

$$e \prec w^{-3}\!:\!0 \prec w^{-3}\!:\!10 \prec w^{-2}\!:\!0 \prec w^{-2}\!:\!10 \prec w^{-1}\!:\!0 \prec w^{-1}\!:\!10 \prec w\!:\!0 \prec w\!:\!10 \preceq f \tag{18}$$

Without loss of generality, assume that w is a (k,n)-Write operation. Because $w\!:\!10$ is the last event between t and u of the form $q\!:\!10$ (and because successive operations of the same Writer write to different elements of Q), (18) implies that $w\!:\!10$ is the last event to write to $Q[k,n]$ before state u.

By assumption, there exists a value c such that $Q[k,j].seq[j] = c$ at every state in the interval $[t,u]$. By (18), this implies that $w^{-3}!seq[j] = w^{-2}!seq[j] = w^{-1}!seq[j] = w!seq[j] = c$. Hence, $w!count[j] = 3$. Because $w\!:\!10$ is the last event to write to $Q[k,n]$ before state u, we have $u \models Q[k,n].done \ \wedge \ Q[k,n].count[j] = 3 \ \wedge \ Q[k,n].seq[j] = c$. Because $u \models Q[k,j].seq[j] = c$, this implies that $u \models \neg ALIVE(Q,k,j)$, which is a contradiction.

So, to summarize, there is at most one event between t and u of the form $p\!:\!10$, if p is a (k,j)-Write operation; at most one such event, if p is a (k,j')-Write operation; and at most four such events, if p is either a (k,n)- or $(k,n \oplus W)$-Write operation, $0 \leq n < W$ and $n \neq j$ *modulo* W. Therefore, there are at most $2 + 4(W-1)$ such events in total between t and u. Hence, $D \leq 4W - 2$. This establishes our proof obligation. □

Lemma 21: $(ALIVE(Q,k,i) \ \wedge \ ALIVE(Q,k,j)) \ \Rightarrow \ (|Q[k,i].tag - Q[k,j].tag| \leq 4W - 1)$.

Proof: Let u be a state, and suppose that $u \models ALIVE(Q,k,i) \ \wedge \ ALIVE(Q,k,j)$, where $i \neq j$. Our proof obligation is to show that $u \models |Q[k,i].tag - Q[k,j].tag| \leq 4W - 1$.

If $u \models Q[k,i].tag = 0 \ \wedge \ Q[k,j].tag = 0$, then our proof obligation is satisfied. So, without loss of generality, assume that $u \models Q[k,j].tag \neq 0$. We have two cases to consider, depending on whether $u \models Q[k,i].tag = 0$.

Case 1: $u \models Q[k,i].tag = 0$. In this case, it suffices to prove that $u \models Q[k,j].tag \leq 4W - 1$. Let v be the initial k-Write operation, and let t be the state following $v\!:\!10$. By the definition of the initial state and our assumption concerning the initial Writes, $Q[k,j].tag = 0$ at every state that occurs before t. Because $u \models Q[k,j].tag \neq 0$, this implies that t either equals or occurs before u.

We now show that $Q[k,i].seq[i]$ has the same value at every state in the closed interval $[t,u]$. If no (k,i)-Write operation exists in the given history, then clearly $Q[k,i].seq[i]$ has the same value at every state in $[t,u]$. Otherwise, it suffices to prove that the event $p\!:\!5$ does not occur between t and u, where p is the initial (k,i)-Write operation. (Recall that all (k,i)-Write operations are of the same process and hence are totally ordered. Hence, assuming there exists a (k,i)-Write operation in the given history, there exists an initial (k,i)-Write operation that precedes all others.) By Lemma 11, each Write operation assigns a nonzero value to its *tag* field. Because $u \models Q[k,i].tag = 0$, this implies that the event $p\!:\!10$ occurs after u. By assumption, $u \models ALIVE(Q,k,i)$. By the definition of $ALIVE$, this implies that $u \models Q[k,i].done$. Therefore, because p is the initial (k,i)-Write operation and $p\!:\!10$ occurs after u, $p\!:\!5$ does not occur between t and u.

Because v is the initial k-Write operation, by Lemma 11, $t \models pref(v,k)$ and $v!tag = 1$. Because u occurs after the initial k-Write operation, by the corollary to Lemma 5, there exists a k-Write operation w such that $u \models pref(w,k)$. Because $Q[k,i].seq[i]$ has the same value at every state in the closed interval $[t,u]$, by Lemma 20, $w!tag \leq v!tag + 4W - 2$. Hence, because $v!tag = 1$, $w!tag \leq 4W - 1$. Because $u \models pref(w,k) \wedge ALIVE(Q,k,j)$, by Lemma 13, $Q[k,j].tag \leq 4W - 1$. This establishes our proof obligation.

Case 2: $u \models Q[k,i].tag \neq 0$. In this case, the value of $Q[k,i].tag$ at state u differs from its initial value. Therefore, there exists a (k,i)-Write operation v such that $u \models last(Q[k,i]) = v$. Similarly, because $u \models Q[k,j].tag \neq 0$, there exists a (k,j)-Write operation w such that $u \models last(Q[k,j]) = w$.

Because $u \models last(Q[k,i]) = v \wedge ALIVE(Q,k,i)$, we have $u \models alive(v,k)$. Similarly, because $u \models last(Q[k,j]) = w \wedge ALIVE(Q,k,j)$, we have $u \models alive(w,k)$. Therefore,

$$u \models alive(v,k) \wedge alive(w,k) \quad . \tag{19}$$

By (19), Lemma 10, and Lemma 1, $u \models Q[k,i].tag = v!tag \wedge Q[k,j].tag = w!tag$. This implies that we can establish our proof obligation by showing that $|v!tag - w!tag| \leq 4W - 1$.

Without loss of generality, assume that $v\!:\!6 \prec w\!:\!6$. Let e be the event prior to state u. By (19) and Lemma 10, $u \models after(w\!:\!10)$. This implies that $w\!:\!10 \preceq e$. Therefore,

$$v\!:\!6 \prec w\!:\!6 \prec w\!:\!10 \preceq e \quad . \tag{20}$$

By (20) and Lemma 15, $v!tag \leq w!tag$. Also, by (20) and the corollary to Lemma 5, there exists a k-Write operation w' such that $pref(w',k)$ holds at u (the state following e). By (19) and the definition of $pref$, $w!tag \leq w'!tag$. Because $v!tag \leq w!tag \leq w'!tag$, to establish our proof obligation, it suffices to prove that $w'!tag \leq v!tag + 4W - 1$.

We consider two cases, depending on whether v is the initial k-Write operation. First, suppose that v is the initial k-Write operation. Let t be the state following $v\!:\!10$. Because v is the initial k-Write operation, by our assumption concerning the initial Writes, v precedes w. Furthermore, by Lemma 11, $t \models pref(v,k)$. Because v precedes w, by (20), we have $v\!:\!6 \prec v\!:\!10 \prec w\!:\!0 \prec w\!:\!6 \prec w\!:\!10 \preceq e$. This implies that t occurs before u. Because $last(Q[k,i]) = v$ at state u, this precedence assertion also implies that $last(Q[k,i]) = v$

at each state in the closed interval $[t, u]$. By Lemma 1, this implies that $Q[k, i].seq[i]$ has the same value at every state in $[t, u]$. Therefore, by Lemma 20, $w'!tag \leq v!tag + 4W - 2$.

Now, suppose that v is not the initial k-Write operation. Let t' be the state prior to $v{:}6$. By our assumption concerning the initial Writes, the initial k-Write operation precedes v. By the corollary to Lemma 5, this implies that there exists a k-Write operation v' such that $t' \models pref(v', k)$. By Lemma 13 and the text of the Writer procedure, $v!tag = v'!tag + 1$. By (20), t' occurs before u. Moreover, because $last(Q[k, i]) = v$ at state u, (20) implies that $last(Q[k, i]) = v$ at each state in the closed interval $[t', u]$. By Lemma 1, this implies that $Q[k, i].seq[i]$ has the same value at every state in $[t', u]$. Hence, by Lemma 20, $w'!tag \leq v'!tag + 4W - 2$. Therefore, $w'!tag \leq v!tag + 4W - 3$. □

We now explain how the original construction given in Figure 3 can be modified to use only bounded variables. We obtain the bounded construction via a series of correctness-preserving transformations. The first transformation involves the introduction of new fields in *Qtype* for holding bounded tags, and corresponding modifications to the Reader and Writer procedures for reading and updating these fields. Specifically, we change the definition of *Qtype* by adding a new field *btag*, ranging over $0..8W - 2$. We require that all *btag* fields, both in Q and in the private variables y and z of each Writer, are initially 0. We modify the Reader procedure by changing statement 1 to the following.

```
1:  for $k = 0$ to $C - 1$ do
        select $max[k]$ such that $ALIVE(x, k, max[k]) \wedge$
                $(\forall n : ALIVE(x, k, n) : (x[k, n].tag,\ n) \leq (x[k, max[k]].tag,\ max[k]))$;
        select $bmax[k]$ such that $ALIVE(x, k, bmax[k]) \wedge$
                $(\forall n : ALIVE(x, k, n) : (x[k, n].btag,\ n) \leq_{mod\ 8W-1} (x[k, bmax[k]].btag,\ bmax[k]))$;
        $val[k],\ phi[k] := x[k, max[k]].val,\ x[k, max[k]].phi$
    od;
```

In the above code fragment, $bmax[k]$ is a new private variable that ranges over $0..2W - 1$. The relation "$\leq_{mod\ 8W-1}$" is defined as follows.

$$\begin{aligned}(z[k,n].btag,\ n) \leq_{mod\ 8W-1} (z[k,m].btag,\ m) \ \equiv\ & (z[k,m].btag = z[k,n].btag\ \wedge\ n \leq m) \vee \\ & (z[k,m].btag > z[k,n].btag\ \wedge\ (z[k,m].btag - z[k,n].btag \leq 4W - 1)) \vee \\ & (z[k,m].btag < z[k,n].btag\ \wedge\ (z[k,m].btag + 8W - 1 - z[k,n].btag \leq 4W - 1))\end{aligned}$$

We modify the Writer procedure by introducing the following two new statements.

```
8':  select $bmax$ such that $ALIVE(z, k, bmax) \wedge$
             $(\forall n : ALIVE(z, k, n) : (z[k, n].btag,\ n) \leq_{mod\ 8W-1} (z[k, bmax].btag,\ bmax))$;
9':  $btag := z[k, bmax].btag + 1\ \ modulo\ \ 8W - 1$;
```

In the above code fragment, *bmax* is a new private variable that ranges over $0..2W - 1$, and *btag* is a new private variable that ranges over $0..8W - 2$, initially 0. Statement $8'$ is inserted after statement 8, and statement $9'$ is inserted after statement 9. Finally, we modify statements 5 and 10 of the Writer procedure so that the value of the private variable *btag* is assigned to the corresponding *btag* field of Q.

Observe that, with the above transformation, all *bmax* and *btag* variables can be viewed as being auxiliary. Thus, the construction's correctness is preserved. We now prove two lemmas that show that it is possible to interchange the roles of the *bmax* and *btag* variables with that of the *max* and *tag* variables, respectively, making the former nonauxiliary and the latter auxiliary. Our final transformation will then involve removing the *max* and *tag* variables. The following lemma shows that the unbounded and bounded tags within each element of Q remain congruent.

Lemma 22: $Q[k,j].btag = Q[k,j].tag \;\; modulo \;\; 8W-1$.

Proof: The lemma is proved by induction. The base case follows from the initial conditions specified above and in Figure 3. Now, assume that the lemma holds for the given history at all states prior to some state u. We show that the lemma also holds at state u. Note that the lemma could possibly be falsified only if state u is reached via the occurrence of the event $w\!:\!10$ for some (k,j)-Write operation w. However, by the induction hypothesis, the lemma holds at the state prior to $w\!:\!6$. Thus, at that state, the *tag* and *btag* fields of each element of Q are congruent. From statements 8, 8′, 9, and 9′ of the Writer procedure, the definition of $\leq_{mod\ 8W-1}$, and Lemma 21, this implies that $w!btag = w!tag \;\; modulo \;\; 8W-1$. Thus, we have $Q[k,j].btag = Q[k,j].tag \;\; modulo \;\; 8W-1$ at state u. □

Our final lemma shows that the *bmax* and *max* variables of each procedure can be interchanged.

Lemma 23: For each Read operation r, $r!max[k] = r!bmax[k]$, and for each Write operation w, $w!max = w!bmax$.

Proof: This lemma follows from Lemmas 21 and 22 and the text of the Writer procedure as modified above. □

Let us now transform the construction by using $bmax[k]$ instead of $max[k]$ when computing $val[k]$ and $phi[k]$ in the Reader procedure. By Lemma 23, this transformation preserves the construction's correctness. Note that, with this change, all *max* and *tag* variables can now be viewed as being auxiliary variables. Thus, all such variables can be removed without affecting the construction's correctness. The resulting construction uses only bounded variables. Furthermore, because this construction was obtained from the original one via transformations that were shown to preserve the latter's correctness, by Theorem 1, we have the following.

Theorem 2: Each well-formed history of the bounded construction is linearizable. □

5 Concluding Remarks

According to our results, if each operation of a concurrent program either writes a single shared variable or reads several shared variables (but not both), then the operations of that program can be implemented from atomic registers without waiting. By contrast, operations that either write several shared variables, or that both read and write shared variables cannot, in general, be implemented from atomic registers in a wait-free manner [5, 16, 18, 27].

Our construction shows that multi-writer composite registers can be implemented with space and time complexity that is very close to that required for implementing single-writer composite registers. (The time complexity is asymptotically the same, given the assumption of [1] that each process that shares the constructed register can both read the register and write each component.) Thus, in the quest for optimal composite register constructions, it probably suffices to focus on the single-writer case: using our multi-writer construction, any improvement in complexity in the single-writer case yields a corresponding improvement for the multi-writer case.

The results of [7, 8, 10] show that composite registers are quite powerful and can be used to implement a variety of other nontrivial shared data objects without waiting. A complete characterization of the class of shared data objects that can be implemented in a wait-free manner from composite registers (and hence atomic registers) remains an important open question. An initial step towards answering this question is given in [9], where a necessary and sufficient condition for wait-free implementation is established for a class of objects called "snapshot objects." A snapshot object can be modified by a set of operations that do not return values, or can be read in its entirety by any process by means of a "snapshot" operation. The condition for wait-free implementation requires that for any pair of operation invocations, either the two invocations commute or one overwrites the other. Assuming unbounded space, the sufficiency of this condition follows from previous results given in [10], where composite registers are used to obtain a wait-free construction with unbounded space complexity that implements any snapshot object satisfying the commutes/overwrites condition. A bounded-space construction for a large subclass of snapshot objects is given in [9]. It is further shown in [9] that no snapshot object that fails to satisfy the commutes/overwrites condition has a wait-free implementation from atomic registers.

Acknowledgements: Special thanks go to Mohamed Gouda for his help and encouragement. I am grateful to Fred Schneider for his comments on this work as it appeared in my Ph.D. dissertation. I would also like to thank Anish Arora, Ken Calvert, Jacob Kornerup, and the referees for their comments on earlier drafts of this paper.

References

[1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit, "Atomic Snapshots of Shared Memory," *Proceedings of the Ninth Annual Symposium on Principles of Distributed Computing*, 1990, pp. 1-14.

[2] J. Anderson, "Composite Registers (Extended Abstract)," *Proceedings of the Ninth Annual Symposium on Principles of Distributed Computing*, 1990, pp. 15-30.

[3] J. Anderson, "Composite Registers," *Distributed Computing*, Vol. 6, pp. 141-154, 1993. First appeared as Technical Report TR.89.25, Department of Computer Sciences, University of Texas at Austin, 1989.

[4] J. Anderson, "Multi-Writer Composite Registers," Technical Report, Department of Computer Science, The University of Maryland at College Park, 1991. First appeared as Technical Report TR.89.26, Department of Computer Sciences, University of Texas at Austin, 1989.

[5] J. Anderson and M. Gouda, "The Virtue of Patience: Concurrent Programming With and Without Waiting," Technical Report TR.90.23, Department of Computer Sciences, The University of Texas at Austin, 1990.

[6] J. Anderson and M. Gouda, "A Criterion for Atomicity," *Formal Aspects of Computing: The International Journal of Formal Methods*, Vol. 4, No. 3, May 1992, pp. 273-298.

[7] J. Anderson and B. Grošelj, "Pseudo Read-Modify-Write Operations: Bounded Wait-Free Implementations," *Proceedings of the Fifth International Workshop on Distributed Algorithms*, Lecture Notes in Computer Science 579, Springer-Verlag, 1991, pp. 52-70.

[8] J. Anderson and B. Grošelj, "Beyond Atomic Registers: Bounded Wait-Free Implementations of Nontrivial Objects," *Science of Computer Programming*, Vol. 19, No. 3, December 1992, pp. 197-237.

[9] J. Anderson and M. Moir, "Towards a Necessary and Sufficient Condition for Wait-Free Synchronization," *Proceedings of the Seventh International Workshop on Distributed Algorithms*, Lecture Notes in Computer Science 725, Springer-Verlag, 1993, pp. 39-53.

[10] J. Aspnes and M. Herlihy, "Wait-Free Data Structures in the Asynchronous PRAM Model," *Proceedings of the Second Annual ACM Symposium on Parallel Architectures and Algorithms*, July, 1990, pp. 340-349.

[11] H. Attiya and O. Rachman, "Atomic Snapshots in $O(n \log n)$ Operations," *Proceedings of the 12th Annual Symposium on Principles of Distributed Computing*, 1993, pp. 29-40.

[12] B. Awerbuch, L. Kirousis, E. Kranakis, P. Vitanyi, "On Proving Register Atomicity," Report CS-R8707, Centre for Mathematics and Computer Science, Amsterdam, 1987. A shorter version appeared as: "A Proof Technique for Register Atomicity," *Proceedings of the Eighth Conference on Foundations of Software Techniques and Theoretical Computer Science*, Lecture Notes in Computer Science 338, Springer-Verlag, 1988, pp. 286-303.

[13] B. Bloom, "Constructing Two-Writer Atomic Registers," *IEEE Transactions on Computers*, Vol. 37, No. 12, December 1988, pp. 1506-1514.

[14] J. Burns and G. Peterson, "Constructing Multi-Reader Atomic Values from Non-Atomic Values," *Proceedings of the Sixth Annual Symposium on Principles of Distributed Computing*, 1987, pp. 222-231.

[15] K. Chandy and J. Misra, *Parallel Program Design: A Foundation*, Addison-Wesley, 1988.

[16] B. Chor, A. Israeli, and M. Li, "On Processor Coordination Using Asynchronous Hardware," *Principles of the Sixth Annual Symposium on Principles of Distributed Computing*, 1987, pp. 86-97.

[17] P. Courtois, F. Heymans, and D. Parnas, "Concurrent Control with Readers and Writers," *Communications of the ACM*, Vol. 14, No. 10, October 1971, pp. 667-668.

[18] M. Herlihy, "Wait-Free Synchronization," *ACM Transactions on Programming Languages and Systems*, Vol. 13, No. 1, January 1991, pp. 124-149.

[19] M. Herlihy and J. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," *ACM Transactions on Programming Languages and Systems*, Vol. 12, No. 3, July 1990, pp. 463-492.

[20] A. Israeli and M. Li, "Bounded time-stamps," *Proceedings of the 28th IEEE Symposium on Foundations of Computer Science*, 1987, pp. 371-382.

[21] L. Kirousis, E. Kranakis, and P. Vitanyi, "Atomic Multireader Register," *Proceedings of the Second International Workshop on Distributed Computing*, Lecture Notes in Computer Science 312, Springer-Verlag, 1987, pp. 278-296.

[22] L. Kirousis, P. Spirakis, and P. Tsigas, "Simple Atomic Snapshots: A Solution With Unbounded Time Stamps," *Proceedings of the International Conference on Computing and Information*, Lecture Notes in Computer Science 497, Springer-Verlag, 1991, pp. 582-587.

[23] L. Kirousis, P. Spirakis, and P. Tsigas, "Reading Many Variables in One Atomic Operation: Solutions with Linear or Sublinear Complexity," *Proceedings of the Fifth International Workshop on Distributed Algorithms*, Lecture Notes in Computer Science 579, Springer-Verlag, 1991, pp. 229-241.

[24] L. Lamport, "Concurrent Reading and Writing," *Communications of the ACM*, Vol. 20, No. 11, November 1977, pp. 806-811.

[25] L. Lamport, "On Interprocess Communication, Parts I and II," *Distributed Computing*, Vol. 1, 1986, pp. 77-101.

[26] M. Li, J. Tromp, and P. Vitanyi, "How to Construct Wait-Free Variables," *Proceedings of International Colloquium on Automata, Languages, and Programming*, Lecture Notes in Computer Science 372, Springer-Verlag, 1989, pp. 488-505.

[27] M. Loui and H. Abu-Amara, "Memory Requirements for Agreement Among Unreliable Asynchronous Processes," *Advances in Computing Research*, Vol. 4, JAI Press, 1987, pp. 163-183.

[28] J. Misra, "Axioms for Memory Access in Asynchronous Hardware Systems," *ACM Transactions on Programming Languages and Systems*, Vol. 8, No. 1, January 1986, pp. 142-153.

[29] R. Newman-Wolfe, "A Protocol for Wait-Free, Atomic, Multi-Reader Shared Variables," *Proceedings of the Sixth Annual Symposium on Principles of Distributed Computing*, 1987, pp. 232-248.

[30] G. Peterson, "Concurrent Reading While Writing," *ACM Transactions on Programming Languages and Systems*, Vol. 5, 1983, pp. 46-55.

[31] G. Peterson and J. Burns, "Concurrent Reading While Writing II: The Multi-Writer Case," *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, 1987, pp. 383-392.

[32] A. Singh, J. Anderson, and M. Gouda, "The Elusive Atomic Register, Revisited," *Proceedings of the Sixth Annual Symposium on Principles of Distributed Computing*, 1987, pp. 206-221. Expanded version to appear in *Journal of the ACM*.

[33] J. Tromp, "How to Construct an Atomic Variable," *Proceedings of the Third International Workshop on Distributed Algorithms*, Lecture Notes in Computer Science 392, Springer-Verlag, 1989, pp. 292-302.

[34] P. Vitanyi and B. Awerbuch, "Atomic Shared Register Access by Asynchronous Hardware," *Proceedings of the 27th IEEE Symposium on the Foundations of Computer Science*, 1986, pp. 233-243.