Composite Registers*

James H. Anderson†

Department of Computer Science
The University of Maryland at College Park
College Park, Maryland 20742

July 1989
Revised March 1991, February 1992

Abstract

We introduce a shared data object, called a *composite* register, that generalizes the notion of an atomic register. 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. A composite register reduces to an ordinary atomic register when there is only one component. In this paper, we show that multi-reader, single-writer atomic registers can be used to implement a composite register in which there is only one writer per component. In a related paper, we show how to use the composite register construction of this paper to implement a composite register with multiple writers per component. These two constructions show that it is possible to implement a shared memory that can be read in its entirety in a single snapshot operation, without using mutual exclusion.

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

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

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

†Work supported, in part, at the University of Texas at Austin by Office of Naval Research Contract N00014-89-J-1913, and at the University of Maryland 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 shared by a collection of processes and is accessed 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 shared data objects are inherently resilient to halting failures: a process that halts while accessing such a data object cannot block the progress of any other process that also accesses that same data object. Wait-free shared data objects also permit maximum parallelism: such a data object can be accessed concurrently by any number of the processes that share it since one access does not have to wait for another to complete.

The notion of an atomic register is of fundamental importance in the study of wait-free shared data objects [18, 19, 22, 24]. 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 [5, 9, 10, 16, 17, 19, 20, 23, 24, 25, 26, 27, 28]. This work shows that, using only atomic registers of the simplest kind, the classical readers-writers problem [13] can be solved without requiring either readers or writers to wait.

In this paper, we go a step further by defining a new shared data object, called a *composite register*, that generalizes the notion of an atomic register. 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. A composite register reduces to a multi-reader, multi-writer atomic register when there is only one component.

We consider here the important question of whether atomic registers can be used to implement composite registers without waiting. We use a two-step approach in addressing this question. The results of this paper constitute the first step, and those of [3] the second. In this paper, we show that multi-reader, single-writer atomic registers can be used to construct a composite register in which there is only one writer per component (henceforth, called a *single-writer* composite register). In [3], we use the construction of this paper to implement a composite register in which several writers per component are allowed (henceforth, called a *multi-writer* composite register).

One of the surprising consequences of this result is 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. Such a memory can be implemented by a single composite register, with each memory location corresponding to a component of the register. To write a given memory location, a process writes the

corresponding component of the composite register. To read any set of memory locations, a process reads the entire composite register, and then selects the values of the components corresponding to this set.

The problem of constructing a composite register from atomic registers has also been considered independently by Afek et al. [1]. They present two composite register constructions: the first implements a single-writer composite register from multi-reader, single-writer atomic registers, and the second implements a multi-writer composite register from multi-reader, multi-writer atomic registers. By contrast, our two constructions (the one in this paper and the one in [3]) together implement a multi-writer composite register using only single-writer atomic registers. Thus, our constructions also solve the problem of implementing a multi-writer atomic register (the case in which there is only one component) from single-writer ones.

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 [6, 7], 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[1] based upon the original value of that variable. However, unlike an RMW operation, 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 [6, 7] that composite registers can be used to implement any shared data object that can either be read, written, or modified by a commutative PRMW operation in a wait-free manner. These results stand in sharp contrast to those of [4, 14], 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. In Section 3 we present the "Shrinking Lemma," which gives a sufficient condition for proving that a construction is correct. The proof of the Shrinking Lemma is given in an appendix. In Section 4, we present our single-writer 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;

[1] 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.

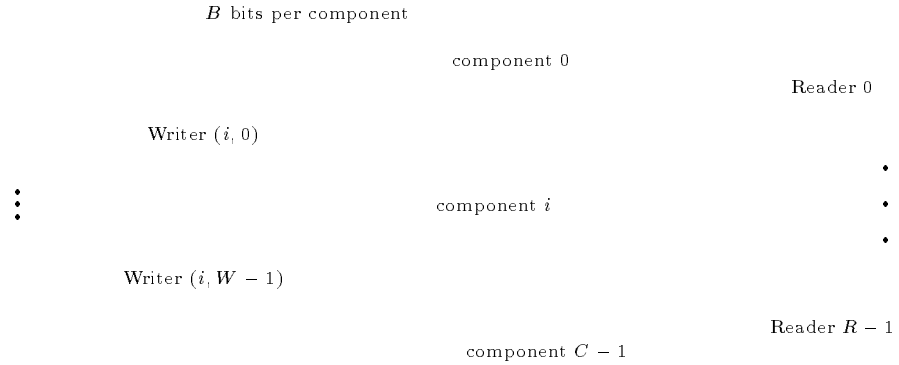

Figure 1: $C/B/W/R$ composite register structure.

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 R-reader, W-writer 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.

- *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 [4].)

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*. An operation of a Reader (Writer) procedure is called *Read operation* (*Write operation*). (Note that, in order to avoid confusion, we 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.) A Write operation of component k of the constructed composite register, where $0 \leq k < C$, is called a *k-Write operation*. An operation p *precedes* another operation q in a history iff each event of p precedes all events of q.

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 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. □

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 [15], 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[2] 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 $\phi_k(v) < \phi_k(w)$.
- *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

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

"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 [26], Proposition 3 in [19], and the definition of an atomic run and the Shrinking Function Theorem in [8].

The correctness proof for the Shrinking Lemma is given in an appendix. 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 $C/B/1/R$ Construction

In this section, we prove that a $C/B/1/R$ composite register can be constructed from multi-reader, single-writer atomic registers. An informal description of the construction is presented in Section 4.1 and the correctness proof is given in Section 4.2.

4.1 Informal Description

The architecture of the construction is depicted in Figure 2, and the construction itself is given in Figure 3. The construction uses $R+2$ shared variables, $Y[0]$, $Y[1..C-1]$, and $Z[0], \ldots, Z[R-1]$. (Note that $Y[1..C-1]$ is considered to be a single variable. We call the variable written by Writer 0 "$Y[0]$" in order to avoid special cases in the proof of correctness. We stress that $Y[0]$ and $Y[1..C-1]$ are two distinct variables.) Notice that the construction is recursive, since variable $Y[1..C-1]$ is a $(C-1)$-component composite register.

As seen in Figures 2 and 3, each $Y[k]$ is partitioned into a number of fields. The fields val and id are common to every $Y[k]$, while $Y[0]$ has a number of additional fields. The val field of $Y[k]$ is used to record the input values of successive k-Write operations, and the id field of $Y[k]$ is an integer variable used to uniquely identify these successive input values. The id fields are auxiliary variables and are used in defining the functions $\phi_0, \ldots, \phi_{C-1}$ of the Shrinking Lemma. (To emphasize that they are auxiliary, we have enclosed them in parentheses in Figure 2.) 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). The term $item$ refers to a (val, id) pair. Fields $seq[0]$ and $seq[1]$ of $Y[0]$ are arrays and are used to store the "sequence numbers" read from $Z[0], \ldots, Z[R-1]$ by a 0-Write operation. Note that each 0-Write operation makes two copies of $Z[0], \ldots, Z[R-1]$, one of which is stored in $Y[0].seq[0]$ and the other in $Y[0].seq[1]$. The ss field of $Y[0]$ is an array used to store the items read from $Y[0], \ldots, Y[C-1]$ by a 0-Write operation; ss stands for "snapshot." The wc field of $Y[0]$ is a modulo-3 integer "write counter" and is incremented by each 0-Write operation.

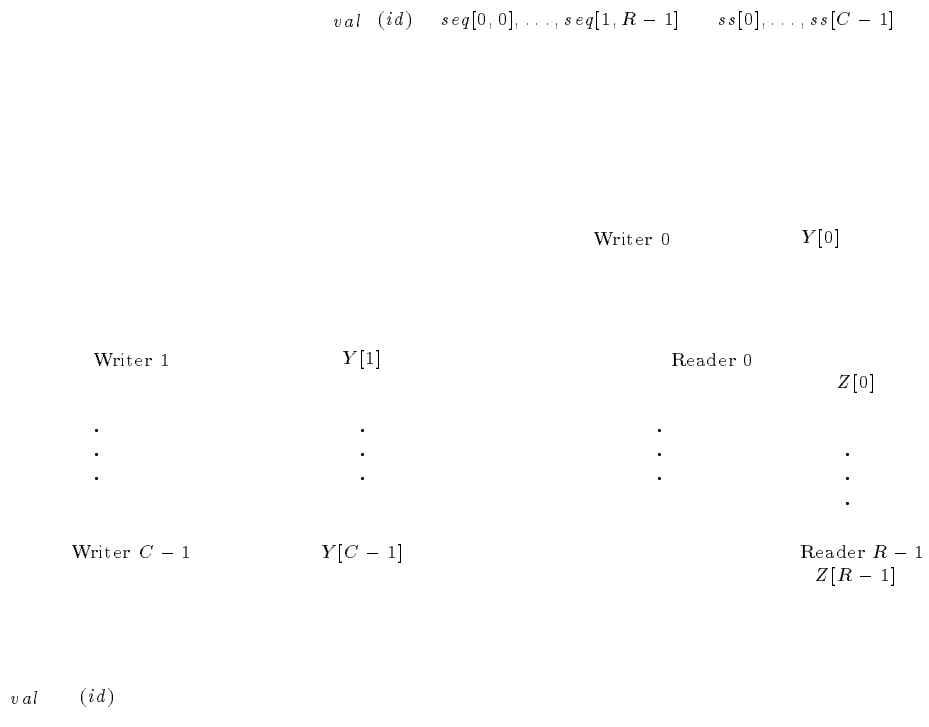

Figure 2: $C/B/1/R$ construction architecture.

Before considering the Reader and Writer procedures in depth, 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 shared variables from reads and writes of private variables. We also assume that each private variable of each procedure retains its value between invocations of that procedure. We use $\oplus$ to denote modulo-3 addition. Each labeled sequence of statements is assumed to be a single atomic statement. In the discussion that follows, we use the following notation for denoting events: if i is a label of a statement of some Reader or Writer procedure, and if p is an operation of that procedure, then $p{:}i$ denotes the event corresponding to the execution of statement i by p. This notation will also be used in the proof of correctness given in Section 4.2.

We now explain the intuition behind the construction. We begin by noting the structure of Writer 0. First, note that $Y[0].val$, $Y[0].wc$, and $Y[0].seq[0]$ are modified only by statement 3 of Writer 0, and each execution of this statement increments the value of $Y[0].wc$, modulo-3. Also, note that $Y[0].ss$ and $Y[0].seq[1]$ are modified only by statement 7 of Writer 0. With this in mind, consider an operation r of Reader j. Let v_k denote the value r returns for component k, where $0 \leq k < C$. We show that there exists a state during the execution of r (i.e., between its first and last events) such that $(\forall k : 0 \leq k < C : Y[k].val = v_k)$. This

```
type valtype   = a B-bit value;
     itemtype  = record  val : valtype;  id : integer /* Auxiliary variable */  end;
     Ytype     = record  val : valtype;  id : integer; /* Auxiliary variable */
                         seq : array[0..1][0..R-1] of 0..2;  ss : array[0..C-1] of itemtype;  wc : 0..2  end

var  Y[0] : Ytype;                                   /* Multi-reader, single-writer atomic register */
     Y[k] : itemtype, for each k, where 1 ≤ k < C;   /* (C-1)-component composite register */
     Z : array[0..R-1] of 0..2                       /* Array of multi-reader, single-writer atomic registers */

initialization
     (∀j : 0 ≤ j < C : Y[j].id = 0)

procedure Reader(j : 0..R-1) returns array[0..C-1] of valtype
private var
     a, c, e, x : Ytype;
     b, d : array[1..C-1] of itemtype;
     item : array[0..C-1] of itemtype;
     newseq : 0..2
begin

     /* Select new sequence number differing from Writer 0's two copies */
 0:  read x := Y[0];
 1:  select newseq such that newseq ≠ x.seq[0, j] ∧ newseq ≠ x.seq[1, j];     /* Can select such a value
 2:  write Z[j] := newseq;                                                       because newseq ranges over 0..2 */

     /* Compute item[0..C-1] */
 3:  read a := Y[0];                          /* Read from Writer 0 */
 4:  read b := Y[1..C-1];                     /* Take snapshot of Writers other than Writer 0 */
 5:  read c := Y[0];                          /* Read from Writer 0 */
 6:  read d := Y[1..C-1];                     /* Take snapshot of Writers other than Writer 0 */
 7:  read e := Y[0];                          /* Read from Writer 0 */
 8:  if e.seq[1, j] = newseq ∨ e.wc = a.wc ⊕ 2 then
         item[0], ..., item[C-1] := e.ss[0], ..., e.ss[C-1]
     else if a.wc = c.wc then
         item[0], item[1], ..., item[C-1] := (a.val, a.id), b[1], ..., b[C-1]
     else /* c.wc = e.wc */
         item[0], item[1], ..., item[C-1] := (c.val, c.id), d[1], ..., d[C-1]
     fi;
 9:  return(item[0].val, ..., item[C-1].val)
end
```

Figure 3: $C/B/1/R$ construction.

```
procedure Writer0(val : valtype)
private var
        seq : array[0..1][0..R − 1] of 0..2;
        wc : 0..2;
        item : itemtype;
        ss : array[0..C − 1] of itemtype;
        y : array[1..C − 1] of itemtype;
        n : 0..R − 1
initialization
        wc = Y[0].wc ∧ item.id = Y[0].id ∧ (∀i : 0 ≤ i < R : seq[1, i] = Y[0].seq[1, i]) ∧
        (∀j : 0 ≤ j < C : ss[j] = Y[0].ss[j])
begin

        /* Compute item, seq[0][0..R − 1], and wc */
   0:   wc, item.val, item.id := wc ⊕ 1, val, item.id + 1;
   1:   for n = 0 to R − 1 do
           2.n: read seq[0, n] := Z[n]                          /* Read from Reader n */
        od;
   3:   write Y[0] := (item.val, item.id, seq[0..1][0..R − 1], ss[0..C − 1], wc);

        /* Compute seq[1][0..R − 1] and ss[0..C − 1] */
   4:   read y := Y[1..C − 1];                                  /* Take snapshot of other Writers */
   5:   ss[0], ss[1], ..., ss[C − 1] := item, y[1], ..., y[C − 1];
   6:   seq[1, 0], ..., seq[1, R − 1] := seq[0, 0], ..., seq[0, R − 1];   /* Note: seq[1] is a copy of seq[0] */
   7:   write Y[0] := (item.val, item.id, seq[0..1][0..R − 1], ss[0..C − 1], wc);
   8:   return
end

procedure Writer(i : 1..C − 1; val : valtype)
private var
        item : itemtype
initialization
        item.id = Y[i].id
begin
   0:   item.val, item.id := val, item.id + 1;
   1:   write Y[i] := item;
   2:   return
end
```

Figure 3: $C/B/1/R$ construction (continued).

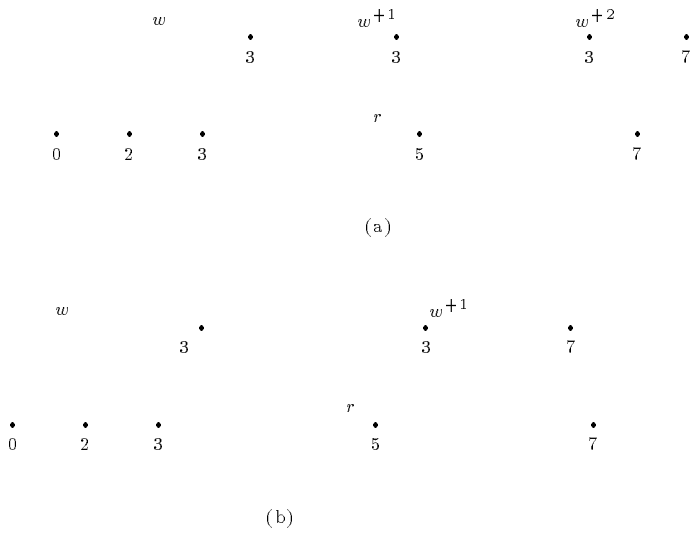

Figure 4: Two example executions.

implies that r can be "shrunk to the point" immediately following this state. We consider three cases, depending on the number of times Writer 0 executes its statement 3 between events $r:3$ and $r:7$. Two of the three cases are illustrated in Figure 4. In this figure, 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.

First, suppose that Writer 0 executes its statement 3 at least three times in the interval between $r:3$ and $r:7$. Then, there exists a 0-Write operation that occurs completely within this interval. Such a scenario is depicted in Figure 4 (a). In this figure, w, w^{+1}, and w^{+2} are three successive operations of Writer 0. By examining statement 4 of the procedure for Writer 0, we see that each 0-Write operation takes a snapshot of the other Writers' values. In the general case under consideration, it is possible to show that r returns the C values from the snapshot of some "overlapping" Write operation. For the specific situation depicted in Figure 4 (a), we can show that r returns the C values from the snapshot of w^{+1}. To see this, first observe that the value written to $Z[j]$ by $r:2$ is copied by w^{+1} to $Y[0].seq[1,j]$ prior to the occurrence of event $r:7$. Because event $w^{+2}:3$ doesn't modify $Y[0].seq[1]$, this implies that $e.seq[1,j] = newseq$ when r executes statement 8 of the Reader procedure. Hence, the C return values of r, v_0, ..., v_{C-1}, equal those read by r from $Y[0].ss[0]$, ..., $Y[0].ss[C-1]$, respectively, when $r:7$ occurs. Because event $w^{+2}:3$ doesn't modify $Y[0].ss$, these C values equal those assigned to $Y[0].ss[0]$, ..., $Y[0].ss[C-1]$ by $w^{+1}:7$. The values so assigned are determined when w^{+1} takes its snapshot, i.e., when the event $w^{+1}:4$ occurs. This implies that $(\forall k :: Y[k].val = v_k)$ at the state immediately prior to the occurrence of event $w^{+1}:4$. By Figure 4 (a), this state occurs during r.

In the second case we consider, Writer 0 executes its statement 3 exactly twice between $r:3$ and $r:7$. In this case, we have a situation such as that depicted in Figure 4 (b). As in the previous case, w and w^{+1} are successive operations of Writer 0. The general case in question is similar to that considered above; in

particular, it can be shown that r returns the C values from the snapshot of an "overlapping" Write operation. For the specific situation depicted in Figure 4 (b), we can show that r returns the C values from the snapshot of w. To see this, first note that, because statement 3 of Writer 0 is executed twice between $r:3$ and $r:7$, $e.wc = a.wc \oplus 2$ holds when r executes its statement 8. Hence, the C return values of r, v_0, ..., v_{C-1}, equal those read by r from $Y[0].ss[0]$, ..., $Y[0].ss[C-1]$, respectively, when $r:7$ occurs. Reasoning as in the previous case, we can show that these values equal those assigned to $Y[0].ss[0]$, ..., $Y[0].ss[C-1]$, respectively, by $w:7$, and that $(\forall k :: Y[0].ss[k] = v_k)$ holds at the state immediately prior to the occurrence of the event $w:4$. By Figure 4 (b), this state occurs during r.

In the third and final case we consider, Writer 0 executes its statement 3 at most once between $r:3$ and $r:7$. In this case, there are two possiblities to consider: statement 3 of Writer 0 is not executed between $r:3$ and $r:5$; or statement 3 of Writer 0 is not executed between $r:5$ and $r:7$. We consider the former possiblity, the latter being similar. Because statement 3 of Writer 0 is not executed between $r:3$ and $r:5$, $Y[0].val$ and $Y[0].wc$ each have same value throughout this interval. This implies that events $r:3$ and $r:5$ both read the same value from $Y[0].wc$, and hence $a.wc = c.wc$ when r executes statement 8 of the Reader procedure. It can also be shown in this case that $e.a[1,j] \neq newseq \ \wedge \ e.wc \neq a.wc \oplus 2$ when r executes statement 8 (i.e., the two cases above do not apply). Hence, the C return values of r, v_0, ..., v_{C-1}, equal those read by r from $Y[0].val$, $Y[1].val$, ..., $Y[C-1].val$, respectively, when events $r:3$ and $r:4$ occur. This implies that $Y[0].val = v_0$ holds prior to the occurrence of event $r:3$ and that $(\forall k : 1 \leq k < C : Y[k].val = v_k)$ holds prior to the occurrence of event $r:4$. Because $Y[0].val$ has the same value in the interval between $r:3$ and $r:5$, this implies that $(\forall k : 0 \leq k < C : Y[k].val = v_k)$ holds at the state prior to the occurrence of $r:4$. This state trivially occurs during r.

We now compute the space complexity of our $C/B/1/R$ construction by determining the number of shared single-reader, singler-writer atomic bits used in the construction. Let $S(C,B,W,R)$ denote the number of such bits required to construct a $C/B/W/R$ composite register. If we remove the auxiliary id fields from our $C/B/1/R$ construction, then the complexity of each of the shared variables is as follows: $Y[0]$ uses $S(1, 4R + CB + B + 2, 1, R)$ bits; $Y[1..C-1]$ uses $S(C-1, B, 1, R+1)$ bits; and $Z[i]$, where $0 \leq i < R$, uses $S(1,2,1,1)$ bits. Using the construction of [26], $S(1,B,1,R) = O(R^2 + BR)$, and using the construction of [27], $S(1,B,1,1) = O(B)$; both of these constructions are asymptotically optimal. This yields $S(C,B,1,R) = O(R^2 + CBR) + S(C-1,B,1,R+1)$. By solving this recurrence, we see that $S(C,B,1,R) = O(CR^2 + C^2BR + C^3B)$.

We compute the time complexity of our $C/B/1/R$ construction by determining the number of reads and writes of shared multi-reader, single-writer atomic registers required for each Read and Write operation of the constructed register (for simplicity, we do not go down to the level of single-reader, single-writer atomic bits when computing the time complexity). Let $TR(C,B,W,R)$ and $TW(C,B,W,R)$ denote the time complexity for Read and Write operations, respectively, of a $C/B/W/R$ composite register. Then, for our construction, $TR(C,B,1,R) = 5 + 2TR(C-1,B,1,R+1)$. By solving this recurrence, we see that $TR(C,B,1,R) = O(2^C)$. For Write operations, we get $TW(C,B,1,R) = R + 2 + TR(C-1,B,1,R+1)$. By solving this recurrence, we get $TW(C,B,1,R) = O(R + 2^C)$.

4.2 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 follows by the architecture of the construction. The Wait-Freedom restriction is satisfied because no procedure contains any unbounded loops or idle-waiting primitives. In this section, we prove that each well-formed history is linearizable. We first define functions $\phi_0, \ldots, \phi_{C-1}$ for a given well-formed history, and then show that the defined ϕ's satisfy the five conditions of the Shrinking Lemma. First, we present a number of definitions and notational conventions.

In the remainder of this section, we assume that k ranges over $\{0, \ldots, C-1\}$. 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
$[\,]$, $(\,)$
.
!
:
$+$, $-$, $\oplus$
$=$, $\neq$, $<$, $>$, $\leq$, $\geq$, $\prec$
$\wedge$, $\vee$
unless
$\equiv$
$\models$

If event e precedes event f, then we write $e \prec f$. If x is a private variable of operation p, then $p!x$ denotes the final value of variable x as assigned by p.

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 [11], we say that the assertion E *unless* F holds iff for every pair of consecutive states in any history, if $E \wedge \neg F$ holds in the first state, then $E \vee F$ holds in the second state.

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.

Let p be an operation of some Reader or Writer procedure P. We use p^{-1} to denote the operation of P that immediately precedes p, and p^{+1} to denote the operation of P that immediately succeeds p. Similarly, p^{-2} denotes the operation of P that immediately precedes p^{-1}, p^{+2} denotes the operation of P that immediately succeeds p^{+1}, etc.

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 X before that state is an event of p.

As mentioned in Section 4.1, each value of type *valtype* is tagged with an integer auxiliary variable, which we call *id*. These auxiliary variables have been introduced in order to facilitate the definition of the functions $\phi_0, \ldots, \phi_{C-1}$.

Definition of ϕ_k: Let r be a Read operation and let w be a k-Write operation. We define the function ϕ_k as follows.

$$\begin{aligned} \phi_k(r) &\equiv r!item[k].id \\ \phi_k(w) &\equiv w!item.id \end{aligned}$$

□

The proof of correctness is based upon Lemma 2. The following lemma is used in the proof of Lemma 2.

Lemma 1: Let r be an operation of Reader j, and let v and w be 0-Write operations such that $last(Y[0]) = v$ at the state prior to $r:3$ and $last(Y[0]) = w$ at the state prior to $r:7$. If $r!e.seq[1,j] \neq r!newseq$, then one of the following is true: $w = v$, $w = v^{+1}$, or $w = v^{+2}$.

Proof: Let r, v, and w be as defined in the lemma. Assume that $r!e.seq[1,j] \neq r!newseq$.

By the text of the Reader procedure, $r:3 \prec r:7$. Therefore, because $last(Y[0]) = v$ at the state prior to $r:3$ and $last(Y[0]) = w$ at the state prior to $r:7$ (and because the 0-Write operations are totally ordered) either $v = w$ or v precedes w. In the former case, our proof obligation is satisfied; so, in the remainder of the proof, assume that v precedes w. We show in this case that either $w = v^{+1}$ or $w = v^{+2}$.

Assume, to the contrary, that $w \neq v^{+1}$ and $w \neq v^{+2}$. Then, because v precedes w, v^{+2} precedes w. Because $last(Y[0]) = v$ at the state prior to $r:3$, $r:3 \prec v^{+1}:3$. Because v^{+1} and v^{+2} are successive 0-Write operations, $v^{+1}:3 \prec v^{+2}:0 \prec v^{+2}:7$. Because v^{+2} precedes w, $v^{+2}:7 \prec w:0$. By the text of the procedure for Writer 0, $w:0 \prec w:3$. Because $last(Y[0]) = w$ at the state prior to $r:7$, $w:3 \prec r:7$. Therefore,

$$r:3 \prec v^{+1}:3 \prec v^{+2}:0 \prec v^{+2}:7 \prec w:0 \prec w:3 \prec r:7 \quad .$$

Observe that v^{+2} precedes or equals w^{-1}. Hence, by the above precedence assertion, $r:3 \prec w^{-1}:2.j \prec w:2.j \prec r:7$. This implies that $last(Z[j]) = r$ at both the state prior to $w^{-1}:2.j$ and the state prior to $w:2.j$. Hence, $w^{-1}!seq[0,j] = r!newseq$ and $w!seq[0,j] = r!newseq$. By statement 6 of the procedure for Writer 0, $w^{-1}!seq[1,j] = w^{-1}!seq[0,j]$ and $w!seq[1,j] = w!seq[0,j]$. Therefore,

$$w^{-1}!seq[1,j] = r!newseq \ \wedge \ w!seq[1,j] = r!newseq \quad . \tag{1}$$

Because $last(Y[0]) = w$ at the state prior to $r:7$, either $w:7 \prec r:7$ or $w:3 \prec r:7 \prec w:7$. In the former case, $r!e.seq[1,j] = w!seq[1,j]$. In the latter case, because statement 3 of Writer 0 does not alter the value of $Y[0].seq[1,j]$, we have $r!e.seq[1,j] = w^{-1}!seq[1,j]$. In either case, by (1), we have $r!e.seq[1,j] = r!newseq$, which is a contradiction. Thus, either $w = v^{+1}$ or $w = v^{+2}$. □

As explained informally in Section 4.1, there exists a state "during" the execution of each Read operation that corresponds to the "snapshot" taken by that operation. This is established formally in the following lemma.

Lemma 2: Let r be a Read operation. Then, there exists a state between the events $r:0$ and $r:9$ such that $(\forall k :: Y[k].val = r!item[k].val \ \wedge \ Y[k].id = \phi_k(r))$.

Proof: Assume that r is an operation of Reader j. We consider four cases, based upon the conditional statement 8 of Reader j.

Case 1: $r!e.seq[1,j] = r!newseq$. Let S be the set of 0-Write operations defined as follows: p is in S iff p is a 0-Write operation and $p:7 \prec r:7$. Note that S is nonempty, since by our assumption concerning the initial Writes, each Read operation is preceded by at least one 0-Write operation. Let w be the Write operation in S such that for each other Write operation p in S, $p:7 \prec w:7$. Then, $last(Y[0])$ equals either w or w^{+1} at the state prior to $r:7$, and in the latter case, $w^{+1}:3 \prec r:7 \prec w^{+1}:7$. Because statement 3 of Writer 0 does not alter the value of $Y[0].seq[1,j]$ or of $Y[0].ss[0..C-1]$, this implies that

$$r!e.seq[1,j] = w!seq[1,j] \ \wedge \ (\forall k :: r!e.ss[k] = w!ss[k]) \quad . \tag{2}$$

By statement 6 of the procedure for Writer 0, $w!seq[0,j] = w!seq[1,j]$. By assumption, $r!e.seq[1,j] = r!newseq$; therefore, by (2),

$$w!seq[0,j] = r!newseq \ \wedge \ w!seq[1,j] = r!newseq \quad . \tag{3}$$

We now show that $r:0 \prec w:3$. Assume, to the contrary, that $w:3 \prec r:0$. By the text of the Reader procedure, $r:0 \prec r:7$. Therefore, $w:3 \prec r:0 \prec r:7$. Because $last(Y[0])$ equals either w or w^{+1} at the state prior to $r:7$, this implies that $last(Y[0])$ equals either w or w^{+1} at the state prior to $r:0$. In the former case, we have $r!x.seq[0,j] = w!seq[0,j]$. By (3), this implies that $r!newseq = r!x.seq[0,j]$. But, by statement 1 of the Reader procedure, $r!newseq \neq r!x.seq[0,j]$; therefore, we have a contradiction.

Now, consider the latter case mentioned above, i.e., $last(Y[0]) = w^{+1}$ at the state prior to $r:0$. By the definition of w, we have $r:7 \prec w^{+1}:7$. Therefore, $w^{+1}:3 \prec r:0 \prec w^{+1}:7$. Because statement 3 of Writer 0 does not alter the value of $Y[0].seq[1,j]$, this implies that $r!x.seq[1,j] = w!seq[1,j]$. Hence, by (3), $r!newseq = r!x.seq[1,j]$, which, by statement 1 of the Reader procedure, is a contradiction. Thus, our assumption that $w:3 \prec r:0$ is false, i.e., $r:0 \prec w:3$.

Because $r!e.seq[1,j] = r!newseq$, by statement 8 of the Reader procedure, $(\forall k :: r!item[k] = r!e.ss[k])$. Therefore, by (2), $(\forall k :: r!item[k] = w!ss[k])$. By the definition of ϕ_k, $(\forall k :: \phi_k(r) = r!item[k].id)$. Hence,

$$(\forall k :: r!item[k].val = w!ss[k].val \ \wedge \ \phi_k(r) = w!ss[k].id) \quad . \tag{4}$$

We now establish the existence of the required state. As shown above, $r:0 \prec w:3$. By the text of the procedure for Writer 0, $w:3 \prec w:4 \prec w:7$. Because w is in set S, $w:7 \prec r:7$. Therefore,

$$r:0 \prec w:3 \prec w:4 \prec w:7 \prec r:7 \quad .$$

Let t denote the state prior to the event $w:4$. By the above precedence assertion, t occurs between $r:0$ and $r:9$. By statement 5 of the procedure for Writer 0, $w!ss[0] = w!item$. Moreover, $t \models Y[0].val = w!item.val \ \wedge \ Y[0].id = w!item.id$. Therefore, $t \models Y[0].val = w!ss[0].val \ \wedge \ Y[0].id = w!ss[0].id$. By (4),

this implies that $t \models Y[0].val = r!item[0].val \wedge Y[0].id = \phi_0(r)$. By statement 5 of the procedure for Writer 0, $(\forall k : k > 0 : w!ss[k] = w!y[k])$. Moreover, $t \models (\forall k : k > 0 : Y[k] = w!y[k])$. Therefore, $t \models (\forall k : k > 0 : Y[k] = w!ss[k])$. By (4), this implies that $t \models (\forall k : k > 0 : Y[k].val = r!item[k].val \wedge Y[k].id = \phi_k(r))$.

Case 2: $r!e.seq[1,j] \neq r!newseq \wedge r!e.wc = r!a.wc \oplus 2$. Assume that $last(Y[0]) = v$ at the state prior to $r:3$ and $last(Y[0]) = w$ at the state prior to $r:7$. (v and w exist because, by our assumption concerning the initial Writes, there exists a 0-Write operation that precedes all Read operations.) Then, by the text of the procedures for Reader j and Writer 0,

$$r!a.wc = v!wc \wedge r!e.wc = w!wc \quad . \tag{5}$$

Because $r!e.seq[1,j] \neq r!newseq$, by Lemma 1, one of the following holds: $w = v$, $w = v^{+1}$, or $w = v^{+2}$. Because $r!e.wc = r!a.wc \oplus 2$, by (5), $w!wc = v!wc \oplus 2$. Therefore, $w = v^{+2}$.

Because $last(Y[0]) = v$ at the state prior to $r:3$, $r:3 \prec v^{+1}:3$. By the text of the procedure for Writer 0, $v^{+1}:3 \prec v^{+1}:4 \prec v^{+1}:7$. Because $w = v^{+2}$, $v^{+1}:7 \prec w:0$. By the text of the procedure for Writer 0, $w:0 \prec w:3$. Because $last(Y[0]) = w$ at the state prior to $r:7$, $w:3 \prec r:7$. Therefore,

$$r:3 \prec v^{+1}:3 \prec v^{+1}:4 \prec v^{+1}:7 \prec w:0 \prec w:3 \prec r:7 \quad . \tag{6}$$

We now show that $r:7 \prec w:7$. By (6), we have $r:2 \prec w:2.j \prec r:7$. This implies that $last(Z[j]) = r$ at the state prior to $w:2.j$, and hence $w!seq[0,j] = r!newseq$. By statement 6 of the procedure for Writer 0, $w!seq[1,j] = w!seq[0,j]$; hence, by transitivity, $w!seq[1,j] = r!newseq$. If $w:7 \prec r:7$, then because $last(Y[0]) = w$ at the state prior to $r:7$, $r!e.seq[1,j] = w!seq[1,j]$. Therefore, by transitivity, $r!e.seq[1,j] = r!newseq$. However, we have assumed in Case 2 that $r!e.seq[1,j] \neq r!newseq$. Hence, we have $r:7 \prec w:7$.

By (6), this implies that $w:3 \prec r:7 \prec w:7$. Since statement 3 of Writer 0 does not alter the value of $Y[0].ss[0..C-1]$, and because v^{+1} and w are successive operations of the same Writer, this implies that $(\forall k :: r!e.ss[k] = v^{+1}!ss[k])$. Because $r!e.seq[1,j] \neq r!newseq \wedge r!e.wc = r!a.wc \oplus 2$, by statement 8 of the Reader procedure, $(\forall k :: r!item[k] = r!e.ss[k])$. Therefore, $(\forall k :: r!item[k] = v^{+1}!ss[k])$. By the definition of ϕ_k, $(\forall k :: \phi_k(r) = r!item[k].id)$. Therefore,

$$(\forall k :: r!item[k].val = v^{+1}!ss[k].val \wedge \phi_k(r) = v^{+1}!ss[k].id) \quad . \tag{7}$$

We now establish the existence of the required state. Let t be the state prior to $v^{+1}:4$. By (6), t occurs between $r:0$ and $r:9$. By statement 5 of the procedure for Writer 0, $v^{+1}!ss[0] = v^{+1}!item$. Moreover, $t \models Y[0].val = v^{+1}!item.val \wedge Y[0].id = v^{+1}!item.id$. Therefore, $t \models Y[0].val = v^{+1}!ss[0].val \wedge Y[0].id = v^{+1}!ss[0].id$. By (7), this implies that $t \models Y[0].val = r!item[0].val \wedge Y[0].id = \phi_0(r)$. By statement 5 of the procedure for Writer 0, $(\forall k : k > 0 : v^{+1}!ss[k] = v^{+1}!y[k])$. Moreover, $t \models (\forall k : k > 0 : Y[k] = v^{+1}!y[k])$. Therefore, $t \models (\forall k : k > 0 : Y[k] = v^{+1}!ss[k])$. By (7), this implies that $t \models (\forall k : k > 0 : Y[k].val = r!item[k].val \wedge Y[k].id = \phi_k(r))$.

Case 3: $r!e.seq[1,j] \neq r!newseq \wedge r!e.wc \neq r!a.wc \oplus 2 \wedge r!a.wc = r!c.wc$. Let v and w be as defined in Case 2, i.e., $last(Y[0]) = v$ at the state prior to $r:3$ and $last(Y[0]) = w$ at the state prior to $r:7$. Let v' be the

0-Write operation such that $last(Y[0]) = v'$ at the state prior to $r{:}5$. Then, by the text of the procedures for Reader j and Writer 0,

$$r!a.wc = v!wc \ \wedge \ r!c.wc = v'!wc \ \wedge \ r!e.wc = w!wc \quad . \tag{8}$$

We first show that $v = v'$. By the definitions of v, v', and w (and the fact that the 0-Write operations are totally ordered), v precedes or equals v' and v' precedes or equals w. As in Case 2, Lemma 1 implies that one of the following holds: $w = v$, $w = v^{+1}$, or $w = v^{+2}$. This implies that one of the following holds as well: $v' = v$, $v' = v^{+1}$, or $v' = v^{+2}$. Because $r!a.wc = r!c.wc$, by (8), we have $v!wc = v'!wc$. Thus, because each 0-Write operation assigns $wc := wc \oplus 1$, and because $\oplus$ is modulo-3 addition, $v' \neq v^{+1}$ and $v' \neq v^{+2}$. Therefore, $v' = v$.

Because $r!e.seq[1,j] \neq r!newseq \ \wedge \ r!e.wc \neq r!a.wc \oplus 2 \ \wedge \ r!a.wc = r!c.wc$, by statement 8 of the Reader procedure, $r!item[0].val = r!a.val$, $r!item[0].id = r!a.id$, and $(\forall k : k > 0 : r!item[k] = r!b[k])$. By the definition of ϕ_k, $(\forall k :: \phi_k(r) = r!item[k].id)$. Therefore,

$$\begin{aligned} &r!item[0].val = r!a.val \ \wedge \ \phi_0(r) = r!a.id \ \wedge \\ &\quad (\forall k : k > 0 : r!item[k].val = r!b[k].val \ \wedge \ \phi_k(r) = r!b[k].id) \quad . \end{aligned} \tag{9}$$

We now establish the existence of the required state. Let t be the state prior to $r{:}4$. Because $last(Y[0]) = v$ both at the state prior to $r{:}3$ and at the state prior to $r{:}5$ (recall $v = v'$), $Y[0].val = r!a.val \ \wedge \ Y[0].id = r!a.id$ holds at each state between $r{:}3$ and $r{:}5$. Thus, because t occurs in this interval, $t \ \models \ Y[0].val = r!a.val \ \wedge \ Y[0].id = r!a.id$. Therefore, by (9), $t \ \models \ Y[0].val = r!item[0].val \ \wedge \ Y[0].id = \phi_0(r)$. By statement 4 of the Reader procedure, $t \ \models \ (\forall k : k > 0 : Y[k] = r!b[k])$. By (9), this implies that $t \ \models \ (\forall k : k > 0 : Y[k].val = r!item[k].val \ \wedge \ Y[k].id = \phi_k(r))$.

Case 4: $r!e.seq[1,j] \neq r!newseq \ \wedge \ r!e.wc \neq r!a.wc \oplus 2 \ \wedge \ r!a.wc \neq r!c.wc$. Let v, v', and w be as defined in Case 3, i.e., $last(Y[0]) = v$ at the state prior to $r{:}3$, $last(Y[0]) = v'$ at the state prior to $r{:}5$, and $last(Y[0]) = w$ at the state prior to $r{:}7$.

We first show that $v' = w$. As in Case 3, v precedes or equals v', v' precedes or equals w, and one of the following holds: $w = v$, $w = v^{+1}$, or $w = v^{+2}$. Because $r!e.wc \neq r!a.wc \oplus 2$, by (8), we have $w!wc \neq v!wc \oplus 2$. Because each 0-Write operation assigns $wc := wc \oplus 1$, this implies that $w \neq v^{+2}$. Because $r!a.wc \neq r!c.wc$, by (8), we have $v!wc \neq v'!wc$. This implies that $v \neq v'$. Therefore, v precedes v', v' precedes or equals w, and either $w = v$ or $w = v^{+1}$. This implies that $v' = w$.

Because $r!e.seq[1,j] \neq r!newseq \ \wedge \ r!e.wc \neq r!a.wc \oplus 2 \ \wedge \ r!a.wc \neq r!c.wc$, by statement 8 of the Reader procedure, $r!item[0].val = r!c.val$, $r!item[0].id = r!c.id$, and $(\forall k : k > 0 : r!item[k] = r!d[k])$. By the definition of ϕ_k, $(\forall k :: \phi_k(r) = r!item[k].id)$. Therefore,

$$\begin{aligned} &r!item[0].val = r!c.val \ \wedge \ \phi_0(r) = r!c.id \ \wedge \\ &\quad (\forall k : k > 0 : r!item[k].val = r!d[k].val \ \wedge \ \phi_k(r) = r!d[k].id) \quad . \end{aligned} \tag{10}$$

We now establish the existence of the required state. Let t be the state prior to $r{:}6$. Because $last(Y[0]) = w$ both at the state prior to $r{:}5$ and at the state prior to $r{:}7$ (recall $v' = w$), $Y[0].val = r!c.val \ \wedge \ Y[0].id =$

$r!c.id$ holds at each state between $r:5$ and $r:7$. Thus, because t occurs in this interval, $t \models Y[0].val = r!c.val \wedge Y[0].id = r!c.id$. Therefore, by (10), $t \models Y[0].val = r!item[0].val \wedge Y[0].id = \phi_0(r)$. By statement 6 of the Reader procedure, $t \models (\forall k : k > 0 : Y[k] = r!d[k])$. By (10), this implies that $t \models (\forall k : k > 0 : Y[k].val = r!item[k].val \wedge Y[k].id = \phi_k(r))$. □

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

Theorem: 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 because each k-Write operation increments the private variable $item.id$ of Writer k, and because the k-Write operations are totally ordered. □

Integrity: Let r be a Read operation. By Lemma 2, there exists a state t that occurs between $r:0$ and $r:9$ such that

$$t \models (\forall k :: Y[k].val = r!item[k].val \wedge Y[k].id = \phi_k(r)) \quad . \tag{11}$$

Let $0 \leq k < C$, and suppose that $last(Y[k]) = v$ at state t. (v exists because, by assumption, there is a k-Write operation that precedes every Read operation.) Then, by the text of the procedure for Writer k, $t \models Y[k].val = v!item.val \wedge Y[k].id = v!item.id$. Therefore, by (11), $v!item.val = r!item[k].val$, and $v!item.id = \phi_k(r)$; by the definition of ϕ_k, this implies that $\phi_k(v) = \phi_k(r)$. □

Proximity: Let r, t, and v be as given in the proof of Integrity. Let w be a k-Write operation. Our proof obligation is to show that if r precedes w, then $\phi_k(r) < \phi_k(w)$, and if w precedes r, then $\phi_k(w) \leq \phi_k(r)$.

First, consider the case in which r precedes w. Because $last(Y[k]) = v$ at state t and because state t occurs between $r:0$ and $r:9$, r does not precede v. Therefore, because r precedes w and because the Write operations on a given component are totally ordered, v precedes w. Therefore, by Uniqueness, $\phi_k(v) < \phi_k(w)$. From the proof of Integrity, $\phi_k(r) = \phi_k(v)$. Therefore, $\phi_k(r) < \phi_k(w)$.

Now, consider the case in which w precedes r. In this case, because $t \models last(Y[k]) = v$, w precedes or equals v. Therefore, by Uniqueness, $\phi_k(w) \leq \phi_k(v)$. Thus, $\phi_k(w) \leq \phi_k(r)$. □

Read Precedence: The proof of Read Precedence is based upon the following property:

$$(\forall D :: Y[k].id = D \quad unless \quad Y[k].id > D) \quad . \tag{12}$$

This property holds because each k-Write operation increments the private variable $item.id$ of Writer k, and because the k-Write operations are totally ordered.

Consider two Read operations r and s. By Lemma 2, there exists a state t that occurs between the first and last events of r such that $t \models (\forall k :: Y[k].id = \phi_k(r))$, and a state u that occurs between the first and last events of s such that $u \models (\forall k :: Y[k].id = \phi_k(s))$. If state t equals state u, then $(\forall k :: \phi_k(r) = \phi_k(s))$. If state t occurs before state u, then by (12), $(\forall k :: \phi_k(r) \leq \phi_k(s))$. If state u occurs before state t, then by

(12), $(\forall k :: \phi_k(s) \leq \phi_k(r))$. This implies that Read Precedence is satisfied. □

Write Precedence: Let r be a Read operation, and let v be an operation of Writer i and w be an operation of Writer j, where $0 \leq i < C$ and $0 \leq j < C$. Assume that v precedes w and $\phi_j(w) \leq \phi_j(r)$. By the definition of ϕ_j, this implies that $w!item.id \leq \phi_j(r)$. Our proof obligation is to show that $\phi_i(v) \leq \phi_i(r)$. By the definition of ϕ_i, it suffices to prove that $v!item.id \leq \phi_i(r)$.

By Lemma 2, there exists a state t between the first and last events of r such that

$$t \models (\forall k :: Y[k].id = \phi_k(r)) \quad . \tag{13}$$

Thus, because $w!item.id \leq \phi_j(r)$, we have $t \models w!item.id \leq Y[j].id$. Let t' be the state prior to $w:0$. Then, by the text of the procedure for Writer j, $t' \models Y[j].id = w!item.id - 1$. Therefore, the value of $Y[j].id$ at state t' is less than the value of $Y[j].id$ at state t. By (12), this implies that state t' occurs before state t.

Define state t'' as follows: if $i = 0$, then let t'' be the state following $v:3$; otherwise, let t'' be the state following $v:1$. Observe that $t'' \models Y[i].id = v!item.id$. Because v precedes w, t'' occurs before t'. This implies that t'' occurs before t. Thus, by (12), the value of $Y[i].id$ at state t is at least the value of $Y[i].id$ at state t''. Hence, $t \models Y[i].id \geq v!item.id$. By (13), we have $t \models Y[i].id = \phi_i(r)$. Therefore, $v!item.id \leq \phi_i(r)$. This establishes our proof obligation. □

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 [4, 12, 14, 21].

The construction of this paper and the single-writer construction of Afek et al. [1] are both based upon the following insight: if a Read operation is overlapped by "too many" Write operations, then it returns the C values as read in a single snapshot by one of these overlapping Writes. In our construction, we have resorted to recursion to enable a Write operation to take a snapshot of all C components. This has the effect of reducing the general multi-writer case to the two-writer case. Afek et al. do not resort to recursion, and as a result, their solution is polynomial in both space and time.

The results of [6, 7] show that composite registers are quite powerful and can be used to implement a variety of other nontrivial shared data objects without waiting. It remains to be seen whether other interesting applications exist. A complete characterization of the class of shared data objects that can be implemented from composite registers without waiting remains an important open question.

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, Jay Misra, Paul Vitanyi, and the referees for their comments on earlier drafts of this paper.

Appendix: Proof of 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 $\phi_k(v) < \phi_k(w)$.
- *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 $\phi_j(v) \leq \phi_j(r)$.

Proof: The proof strategy is as follows. We first augment the precedence relation on operations in history h by adding pairs of operations. We then show that the resulting relation is an irreflexive partial order, i.e., it is irreflexive and transitive. Finally, we show that any extension of this relation to an irreflexive total order satisfies the conditions in the definition of a linearizable history given in Section 2.

In the remainder of the proof, we use r and s to denote Read operations in history h, v and w to denote Write operations in h, and x, y, and z to denote arbitrary operations in h. We also assume that i, j, and k each range over $\{0, \ldots, C-1\}$. If x precedes y in h, then we write $x \lhd y$. We let $(x \trianglelefteq y) \ \equiv \ (x = y \ \vee \ x \lhd y)$. We now define six relations A, B, C, D, E, and F; in these definitions, we assume that v and v' denote j-Write operations, and w and w' denote k-Write operations.

- A includes all pairs (x, y) such that $x \lhd y$.
- B includes all pairs (w, r) such that $\phi_k(w) \leq \phi_k(r)$, and all pairs (r, w) such that $\phi_k(r) < \phi_k(w)$.
- C includes all pairs (r, s) such that $(\exists k :: \phi_k(r) < \phi_k(s))$.
- D includes all pairs (v, w) such that $(\exists r :: vBr \ \wedge \ rBw)$.
- E includes all pairs (v, w), such that $v \neq w$ and for some v' and w',

$$\phi_j(v) \leq \phi_j(v') \ \wedge \ v' \trianglelefteq w' \ \wedge \ \phi_k(w') \leq \phi_k(w) \quad .$$

- $F \ \equiv \ A \ \cup \ B \ \cup \ C \ \cup \ D \ \cup \ E$

Relation A is the precedence relation on operations in history h. Relation F is obviously an extension of A. We now show that F is irreflexive and transitive. To prove that F is irreflexive, we are obliged to show that $xFy \Rightarrow x \neq y$. If xAy, then because A is an irreflexive partial order, $x \neq y$. If xBy, then one of x and y is a Read operation and the other is a Write operation, so $x \neq y$. If xCy, then $\phi_k(x) < \phi_k(y)$ for some k, which implies that $x \neq y$. If xDy, then because relation B (which is used to define D) is anti-symmetric, we have $x \neq y$. If xEy, then by the definition of E, $x \neq y$. Therefore, we conclude that F is irreflexive.

In the proof of transitivity, we use the following three properties.

Property 1: For each pair of Read operations r and s, $rFs \Rightarrow (\forall k :: \phi_k(r) \leq \phi_k(s))$.

Proof of Property 1: Assume that rFs holds. Of the five relations that define F, only A and C can relate two Read operations. Therefore, rAs holds or rCs holds. In the former case (i.e., r precedes s), by Read Precedence, $(\forall k :: \phi_k(r) \leq \phi_k(s))$. In the latter case, by the definition of C, $(\exists k :: \phi_k(r) < \phi_k(s))$; hence, by Read Precedence, $(\forall k :: \phi_k(r) \leq \phi_k(s))$. □

Property 2: For each Read operation r and k-Write operation w, $rFw \Rightarrow \phi_k(r) < \phi_k(w)$ and $wFr \Rightarrow \phi_k(w) \leq \phi_k(r)$.

Proof of Property 2: We prove that $rFw \Rightarrow \phi_k(r) < \phi_k(w)$; the proof that $wFr \Rightarrow \phi_k(w) \leq \phi_k(r)$ is similar. Assume that rFw holds. Of the five relations that define F, only A and B can relate a Read operation and a Write operation. Therefore, rAw holds or rBw holds. In the former case (i.e., r precedes w), by Proximity, $\phi_k(r) < \phi_k(w)$. In the latter case, by the definition of B, $\phi_k(r) < \phi_k(w)$. □

Property 3: For each pair of Write operation v and w, $vAw \Rightarrow vEw$.

Proof of Property 3: Let v be a j-Write operation and let w be a k-Write operation such that vAw holds. Because vAw holds (i.e., v precedes w), v and w are distinct. Therefore, letting $v' = v$ and $w' = w$, we have

$$v \neq w \ \wedge \ \phi_j(v) \leq \phi_j(v') \ \wedge \ v' \trianglelefteq w' \ \wedge \ \phi_k(w') \leq \phi_k(w) \ .$$

This implies that vEw. □

To prove that F is transitive, we are obliged to show that $xFy \ \wedge \ yFz \Rightarrow xFz$. We have to consider eight cases since each of x, y, and z can be either a Read operation or a Write operation.

Case 1: x, y, and z are all Read operations. Of the five relations that define F, only A and C can relate two Read operations. If xAy and yAz, then because A is a partial order, xAz. Now, suppose that xCy holds. By the definition of C, we have $\phi_j(x) < \phi_j(y)$ for some j. By Property 1, $(\forall k :: \phi_k(y) \leq \phi_k(z))$. Therefore, by transitivity, $\phi_j(x) < \phi_j(z)$, i.e., xCz. Similar reasoning applies if yCz holds.

Case 2: x and y are Read operations and z is a j-Write operation. By Property 1, $(\forall k :: \phi_k(x) \leq \phi_k(y))$.

By Property 2, $\phi_j(y) < \phi_j(z)$. Therefore, by transitivity, $\phi_j(x) < \phi_j(z)$, i.e., xBz.

Case 3: x and z are Read operations and y is a j-Write operation. By Property 2, $\phi_j(x) < \phi_j(y)$ and $\phi_j(y) \leq \phi_j(z)$. Therefore, by transitivity, $\phi_j(x) < \phi_j(z)$, i.e., xCz.

Case 4: x is a Read operation, y is a j-Write operation, and z is a k-Write operation. Of the five relations that define F, only A, D, and E can relate two Write operations. Thus, by Property 3, yDz holds or yEz holds. If yDz holds, then by the definition of D, there exists a Read operation r such that yBr and rBz. Because $xFy \ \wedge \ yBr$ holds, by Case 3, we have xFr. Because $xFr \ \wedge \ rBz$ holds, by Case 2, we have xFz.

Now, consider the case yEz. By the definition of E, there exists a j-Write operation v and a k-Write operation w such that

$$\phi_j(y) \leq \phi_j(v) \ \wedge \ v \trianglelefteq w \ \wedge \ \phi_k(w) \leq \phi_k(z) \quad .$$

By Property 2, $\phi_j(x) < \phi_j(y)$; thus, by transitivity, $\phi_j(x) < \phi_j(v)$. If $v = w$ (which implies that $j = k$), then $\phi_k(x) < \phi_k(w)$. If, on the other hand, $v \triangleleft w$, then by the contrapositive of Write Precedence, $\phi_k(x) < \phi_k(w)$. Therefore, by transitivity, $\phi_k(x) < \phi_k(z)$, i.e., xBz.

Case 5: x is a j-Write operation and both y and z are Read operations. By Property 2, $\phi_j(x) \leq \phi_j(y)$. By Property 1, $(\forall k :: \phi_k(y) \leq \phi_k(z))$. Therefore, by transitivity, $\phi_j(x) \leq \phi_j(z)$, i.e., xBz.

Case 6: x is a j-Write operation, y is a Read operation, and z is a k-Write operation. By Property 2, $\phi_j(x) \leq \phi_j(y)$ and $\phi_k(y) < \phi_k(z)$. Hence, by the definition of B, xBy and yBz. Therefore, by the definition of D, xDz.

Case 7: x is a j-Write operation, y is a k-Write operation, and z is a Read operation. Of the five relations that define F, only A, D, and E can relate two Write operations. Thus, by Property 3, xDy holds or xEy holds. If xDy holds, then by the definition of D, there exists a Read operation r such that xBr and rBy. Because $rBy \ \wedge \ yFz$ holds, by Case 3, we have rFz. Because $xBr \ \wedge \ rFz$ holds, by Case 5, we have xFz.

Now, consider the case xEy. By the definition of E, there exists a j-Write operation v and a k-Write operation w such that

$$\phi_j(x) \leq \phi_j(v) \ \wedge \ v \trianglelefteq w \ \wedge \ \phi_k(w) \leq \phi_k(y) \quad .$$

By Property 2, $\phi_k(y) \leq \phi_k(z)$. Thus, by transitivity, $\phi_k(w) \leq \phi_k(z)$. If $v = w$ (which implies that $j = k$), then $\phi_j(v) \leq \phi_j(z)$. If, on the other hand, $v \triangleleft w$, then by Write Precedence, $\phi_j(v) \leq \phi_j(z)$. Hence, by transitivity, $\phi_j(x) \leq \phi_j(z)$, i.e., xBz.

Case 8: x, y, and z are all Write operations. Of the five relations that define F, only A, D, and E can relate two Write operations. Thus, by Property 3, xDy holds or xEy holds, and yDz holds or yEz holds. If xDy holds, then there exists a Read operation r such that xBr and rBy. Because $rBy \ \wedge \ yFz$ holds, by Case 4, we have rFz. Because $xBr \ \wedge \ rFz$ holds, by Case 6, we have xFz. The case in which yDz holds is similar.

The remaining possibility is xEy and yEz. Assume that x is an i-Write operation, y is a j-Write operation, and z is a k-Write operation. By the definition of E, there exists an i-Write operation v, j-Write

operations w and v', and a k-Write operation w' such that

$$x \neq y \ \wedge \ \phi_i(x) \leq \phi_i(v) \ \wedge \ v \trianglelefteq w \ \wedge \ \phi_j(w) \leq \phi_j(y) \tag{14}$$

and

$$y \neq z \ \wedge \ \phi_j(y) \leq \phi_j(v') \ \wedge \ v' \trianglelefteq w' \ \wedge \ \phi_k(w') \leq \phi_k(z) \ . \tag{15}$$

There are three possibilities to consider: $i = j$, $j = k$, and $i \neq j \ \wedge \ j \neq k$. First, suppose that $i = j$. We show that xEz holds by first proving that $\phi_j(x) < \phi_j(v')$. By (14), $v \trianglelefteq w$; hence, by Uniqueness, $\phi_j(v) \leq \phi_j(w)$. Therefore (because $i = j$), by (14), $\phi_j(x) \leq \phi_j(y)$. Because $x \neq y$, Uniqueness implies that $\phi_j(x) \neq \phi_j(y)$. Thus, $\phi_j(x) < \phi_j(y)$. Hence, by (15), $\phi_j(x) < \phi_j(v')$.

We now show that xEz. If $i = j \ \wedge \ j \neq k$, then because x is an i-Write operation and z is a k-Write operation, $x \neq z$. If $i = j \ \wedge \ j = k$, then by (15) and Uniqueness, $\phi_j(v') \leq \phi_j(w')$. Thus, because $\phi_j(x) < \phi_j(v')$, by (15), $\phi_j(x) < \phi_j(z)$. This implies that $x \neq z$. Therefore, we conclude for the case $i = j$ that

$$x \neq z \ \wedge \ \phi_j(x) < \phi_j(v') \ \wedge \ v' \trianglelefteq w' \ \wedge \ \phi_k(w') \leq \phi_k(z) \ .$$

Thus, xEz.

The case in which $j = k$ is similar to the case $i = j$.

Now suppose that $i \neq j$ and $j \neq k$. In this case, we prove that xEz by first showing that $v \triangleleft w'$. Because $i \neq j$ and because v is an i-Write operation and w a j-Write operation, we have $v \neq w$. Similarly, because $j \neq k$, we have $v' \neq w'$. Hence, by (14) and (15), we have $v \triangleleft w$ and $v' \triangleleft w'$. Note that (14) and (15) also imply that $\phi_j(w) \leq \phi_j(v')$. Therefore, by Uniqueness, $\neg(v' \triangleleft w)$. Also, observe that $v \triangleleft w \ \wedge \ v' \triangleleft w' \ \Rightarrow \ v' \triangleleft w \ \vee \ v \triangleleft w'$. Thus, $v \triangleleft w'$. Hence, the following expression holds.

$$\phi_i(x) \leq \phi_i(v) \ \wedge \ v \triangleleft w' \ \wedge \ \phi_k(w') \leq \phi_k(z)$$

If $i \neq k$, then because x is an i-Write operation and z a k-Write operation, $x \neq z$. If, on the other hand, $i = k$, then by Uniqueness, $\phi_i(v) < \phi_i(w')$; hence, $\phi_i(x) < \phi_i(z)$, which implies that $x \neq z$. Therefore, we conclude that xEz holds.

Thus, we have established that F is an irreflexive partial order. We now show that any extension of F to an irreflexive total order satisfies the conditions given in the definition of a linearizable history. The following property is used in the proof.

Property 4: For each pair of k-Write operations v and w, $vFw \ \Rightarrow \ \phi_k(v) < \phi_k(w)$.

Proof of Property 4: Assume that vFw holds. Then, by Property 3, vDw holds or vEw holds. If vDw holds, then there exists a Read operation r such that $vBr \ \wedge \ rBw$. By the definition of B, $\phi_k(v) \leq \phi_k(r)$ and $\phi_k(r) < \phi_k(w)$. This implies that $\phi_k(v) < \phi_k(w)$. If, on the other hand, vEw holds, then there exists k-Write operations v' and w' such that

$$\phi_k(v) \leq \phi_k(v') \ \wedge \ v' \trianglelefteq w' \ \wedge \ \phi_k(w') \leq \phi_k(w) \ .$$

By Uniqueness, $\phi_k(v') \leq \phi_k(w')$. This implies that $\phi_k(v) \leq \phi_k(w)$. By the definition of E, $v \neq w$. Therefore, by Uniqueness, $\phi_k(v) < \phi_k(w)$. □

Let r be a Read operation. By Integrity, there exists a k-Write operation v such that $\phi_k(v) = \phi_k(r)$ and the input value of v is the same as the output value of r for component k. By the definition of B, vFr. Moreover, by Properties 2 and 4, $\neg(\exists w : w \text{ is a } k\text{-Write} : vFw \ \wedge \ wFr)$. Observe that, by the definition of B, F totally orders each Read with respect to all Writes. Also, by the definition of E and Uniqueness, the Writes on a given component are totally ordered. Thus, any extension of relation F to an irreflexive total order satisfies the conditions given in the definition of a linearizable history. □

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," *Proceedings of the Ninth Annual Symposium on Principles of Distributed Computing*, 1990, pp. 15-30.

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

[4] 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.

[5] 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.

[6] 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. Expanded version to appear in *Science of Computer Programming*.

[7] 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.

[8] 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 entitled "A Proof Technique for Register Atomicity" appeared in *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.

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

[10] 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.

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

[12] 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.

[13] 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.

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

[15] 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.

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

[17] 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.

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

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

[20] 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.

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

[22] 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.

[23] 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.

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

[25] 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.

[26] 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.

[27] 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.

[28] 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.