COMPOSITE REGISTERS[*]

James H. Anderson[†]

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-89-25 September 1989
July 1990 (Revision)

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 variable that is partitioned into a number of components. An operation of such a register either writes a value to one of the components or reads the values of all of the components. A composite register reduces to an ordinary atomic register when there is only one component. In this paper, we show that 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 for a process of a concurrent program to take an atomic snapshot of an entire shared memory 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.

[*] A preliminary version of this paper will be presented at the *Ninth Annual ACM Symposium on Principles of Distributed Computing*.
[†] Work supported in part by Office of Naval Research Contract N00014-89-J-1913.

1 Introduction

The concept of an atomic register is of fundamental importance in the theory of concurrent programming; see, for example, [4, 5, 8, 9, 11, 12, 13, 14, 15]. An *atomic register* is a shared data object that can either be read or written (but not both) in one indivisible operation. Such a data object is characterized by the number of processes that can write it, the number of processes that can read it, and the number of bits that it stores. The simplest atomic register can be written by one process, read by one process, and store a one-bit value; the most complex can be written and read by several processes and store any number of bits. The previously cited papers show that the most complex atomic register can be implemented in terms of the simplest.

In this paper, we go one step further by defining a new shared data object, which we call a *composite register*, that generalizes the notion of an atomic register. A composite register is an array-like variable that is partitioned into a number of components. An operation of such a register either writes a value to one of the components or reads the values of all of the components. A composite register reduces to an ordinary atomic register when there is only one component.

A composite register differs significantly from an atomic register. A write operation of an atomic register always "overwrites" the current value of the register. By contrast, a write operation of a composite register only "overwrites" the value of a particular component; the values of all other components are left unchanged.

We consider here the important question of whether composite registers can be constructed from atomic registers. Such a construction consists of a set of writer and reader programs that communicate via a set of "internal" atomic registers. A process writes a value to one of the components of the constructed composite register by invoking one of the writer programs for that component. A process reads the values of all of the components by invoking one of the reader programs. Different programs can be invoked by different processes concurrently; the net effect, however, is required to resemble that of a serial invocation. The programs are restricted to be wait-free, i.e., synchronization primitives and unbounded busy-waiting loops are not allowed. This restriction guarantees that a process reads or writes the constructed composite register in a finite amount of time, regardless of the activities of other processes. It also ensures that the read or write of a process is immune to the failure of other processes that also access the constructed composite register.

We use a two step approach to show that atomic registers can be used to construct composite registers. In this paper, we show that atomic registers can be used to construct a composite register in which there is only one writer per component. In [2], we show that a composite register with multiple writers per component can be constructed from a composite register with one writer per component.

One of the surprising consequences of this result is that, using only atomic registers, a process of a concurrent program can take an atomic "snapshot" of an entire shared memory without using mutual exclusion. Such a shared memory can be implemented by a single composite register, with each shared variable corresponding to a component of the register. To write a given variable, a process writes the corresponding component of the composite register. To read some set of variables, a process reads the entire composite register, and then selects the values of the components corresponding to the set. A global snapshot operation is performed by simply reading the set of all variables.

The problem of constructing a composite register from atomic registers has also been considered independently by Afek et al. [1, 10]. In particular, Afek et al. show that an "atomic snapshot" primitive can be constructed from multiple-writer atomic registers. It is interesting to note that the construction of this paper uses only single-writer atomic registers. Thus, the construction given in [2] can be used as a means for constructing a multiple-writer atomic register (the case in which there is only one component). A more detailed comparison of our solution with that of Afek et al. appears in Section 4.

Our results shed new light on where the boundary lies between those operations that can be implemented from atomic registers without waiting and those that cannot. It has been shown that it is impossible to use atomic registers to implement (without waiting) an operation that atomically reads and modifies a shared variable [3, 6]. For example, it is impossible to implement an operation that *both* reads and increments the value of a shared variable in one atomic step. Although this "read-and-modify" barrier cannot be crossed, by using composite registers we can, in some cases, come surprisingly close. For example, we show in Section 4 that it is possible to implement a shared variable that can *either* be read or incremented in one atomic step. In this implementation, an increment operation is performed without reading any shared variable.

The rest of the paper is organized as follows. In Section 2, we formally define the problem of constructing a composite register from atomic registers. In Section 3, we present a construction of a composite register with only one writer per component. Concluding remarks appear in Section 4.

2 Composite Register Construction

In this section, we give the conditions that a composite register construction must satisfy to be correct. For brevity, we will be rather informal about describing what we mean by a "construction." However, our treatment of the correctness condition will be formal.

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

A construction consists of a set of Writer programs and a set of Reader programs that communicate via a set of "internal" variables. A Writer program is invoked in order to

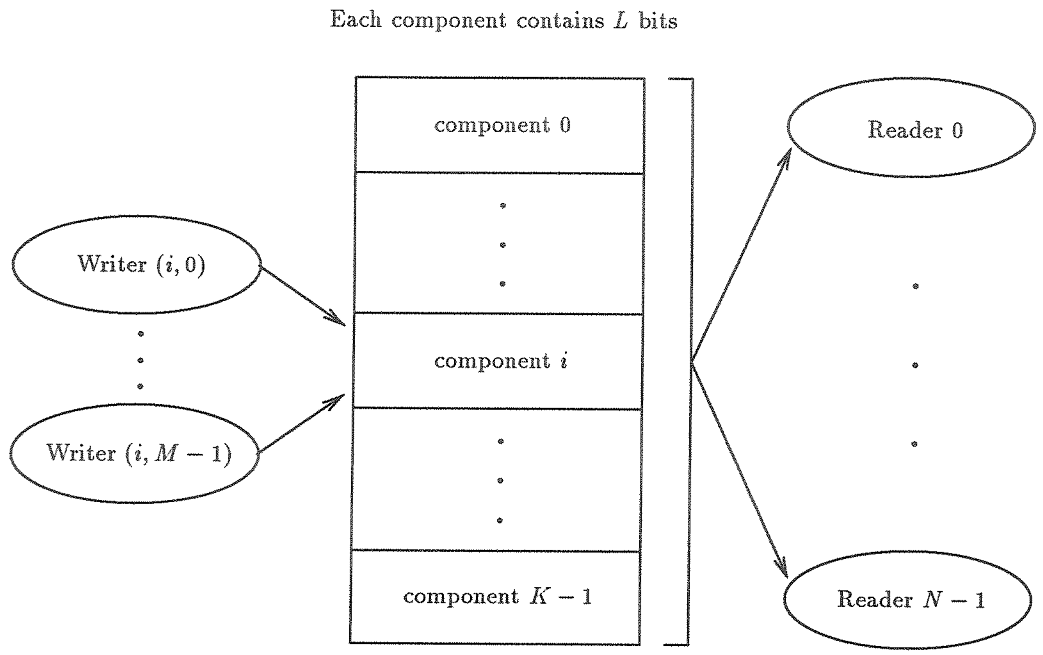

Figure 1: $K/L/M/N$ composite register structure.

Write a value to a component of the constructed composite register. A Reader program is invoked in order to Read the values of all of the components of the constructed composite register. Each Writer program has one input parameter indicating the value to be Written; each Reader program has one output parameter for each component of the constructed register.

We designate a composite register construction by a 4-tuple $K/L/M/N$, where K is the number of components, L is the number of bits per component, M is the number of Writers per component, and N is the number of Readers. (Thus, a $1/L/M/N$ composite register is an ordinary atomic register.) The structure of a $K/L/M/N$ composite register construction is depicted in Figure 1. In this figure, ovals denote programs, boxes denote variables, and arrows denote direction of communication; an outgoing arrow from a program to a variable indicates that the program Writes the variable, while an arrow in the reverse direction indicates that the program Reads the variable. Note that this figure only depicts the Writer programs for component i. For an example of a Reader or Writer program, see the programs in Figures 4 and 5.

Each internal shared variable of a construction corresponds to an atomic register — thus, a statement of a program can either read a single shared variable, or write a single shared variable, but not both; i.e., in each statement, there is at most one occurrence of a shared variable. As mentioned in the introduction, each program of a construction is "wait-free," i.e., synchronization primitives and busy-waiting loops are not allowed. (A more formal definition of wait-freedom is given in [3].)

We now define several concepts that are needed to state the correctness condition for a construction. These definitions apply to a given construction.

Definition: A *state* is an assignment of values to the variables of the construction. One or more states are designated as *initial states.* □

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

Definition: Let t and u be any two states of a construction such that state u is the result of executing some statement at state t. If e is the event corresponding to this statement execution, then we write $t \xrightarrow{e} u$. A *history* of a construction is a sequence $t_0 \xrightarrow{e_0} t_1 \xrightarrow{e_1} \cdots$ where t_0 is an initial state. □

A Reader (or Writer) program can be repeatedly invoked to Read (or Write) the constructed composite register. Therefore, a given statement may be executed many times in a given history. Each such execution corresponds to a distinct event.

Definition: Event e *precedes* another event f in a history iff e occurs before f in the history. The set of events in a history corresponding to some program invocation is called an *operation*. An operation p *precedes* another operation q in a history iff each event of p precedes all events of q. □

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

For the proof of correctness of a construction, it is sufficient to consider only histories that do not contain any incomplete program executions (i.e., incomplete operations). From now on, we deal only with such well-formed histories.

Definition: A Write operation of component k of the constructed composite register, where $0 \leq k < K$, is called a *k-Write operation.* □

In order to avoid special cases when proving the correctness of a construction, we make the following assumption concerning the initial Write operations.

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

According to the following definition, if several operations are executed concurrently, then the net effect should be equivalent to some serial order. This definition is similar to the definition of linearizability as given in [7].

Definition: Let h be a history of a construction. History h is *atomic* iff the precedence relation on operations (which is a partial order) can be extended[1] to a total order $\sqsubset$ where for each Read operation r in h and each k in the range $0 \leq k < K$, the value Read by r for component k is the same as the value Written by the k-Write operation v that is defined as follows: $v \sqsubset r \ \wedge \ \neg(\exists w : w \text{ is a } k\text{-Write} : v \sqsubset w \sqsubset r)$. □

Note that the Write operation v in the definition above exists by our assumption concerning the initial Writes.

Definition: A construction of a composite register is *correct* iff all of its histories are atomic. □

This correctness condition, 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 atomic. Intuitively, a history is atomic 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 the operation appears to take effect. For this reason, the following lemma is referred to as the "Shrinking Lemma." The proof of this lemma is given in an appendix.

Shrinking Lemma: A history h is atomic if for each k, where $0 \leq k < K$, 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 $0 \leq k < K$, there exists a k-Write operation w in h such that $\phi_k(r) = \phi_k(w)$. Furthermore, the value Read by r for component k is the same as the value Written by 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 $0 \leq j < K$ and $0 \leq k < K$, if v precedes w and $\phi_k(w) \leq \phi_k(r)$, then $\phi_j(v) \leq \phi_j(r)$. □

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

3 $K/L/1/N$ Construction

In this section, we prove that a $K/L/1/N$ composite register can be constructed from atomic registers. An informal description of the construction is presented in Section 3.1 and the correctness proof is given in Section 3.2.

3.1 Informal Description

The architecture of the construction is depicted in Figure 2. The construction uses $N + 2$ shared variables, $Y[0]$, $Y[1..K-1]$, and $Z[0], \ldots, Z[N-1]$. (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 $Y[1..K-1]$ are two distinct variables.) Notice that the construction is recursive, since variable $Y[1..K-1]$ is a $(K-1)$-component composite register.

The shared variable declarations are given in Figure 3. The field names appearing in the type definitions are as follows.

val: The value of a particular component.

id: An auxiliary variable that each Writer appends to the *val* that it Writes. The *id* fields are introduced solely to facilitate the proof of correctness.

item: A (val, id) pair.

z: The values read from $Z[0], \ldots, Z[N-1]$ by an operation of Writer 0. Note that Writer 0 makes two "copies" of these values, one of which is stored in $z[0]$ and the other in $z[1]$.

ss: The set of all items (one for each Writer) as read by an operation of Writer 0; *ss* stands for "snapshot."

seq: A modulo-3 integer "sequence number" that is incremented by each operation of Writer 0. (We use $\oplus$ to denote modulo-3 addition.)

The Reader and Writer programs are shown in Figures 4 and 5, respectively. We use a special syntax in order to distinguish reads and writes of shared variables from reads and writes of private variables. A program reads a given shared variable V by executing a statement of the form "**read** $x := V$," where x is a private variable of the same type as V. A program writes a shared variable V by executing a statement of the form "**write** $V := x$." We assume that the private variables of each program retain their values between invocations.

Each Writer i, where $1 \leq i < K$, simply writes its *val* and *id* to $Y[i]$. Writer 0, on the other hand, is more interesting. The execution of Writer 0 consists of two phases. In its first phase, new values are computed for each of $Y[0].item$, $Y[0].z[0]$, and $Y[0].seq$. In its second phase, Writer 0 takes a "snapshot" of the other Writers' values. Then, new values are computed for each of $Y[0].z[1]$ and $Y[0].ss$. Note that $Y[0].z[1]$ is a "copy" of $Y[0].z[0]$.

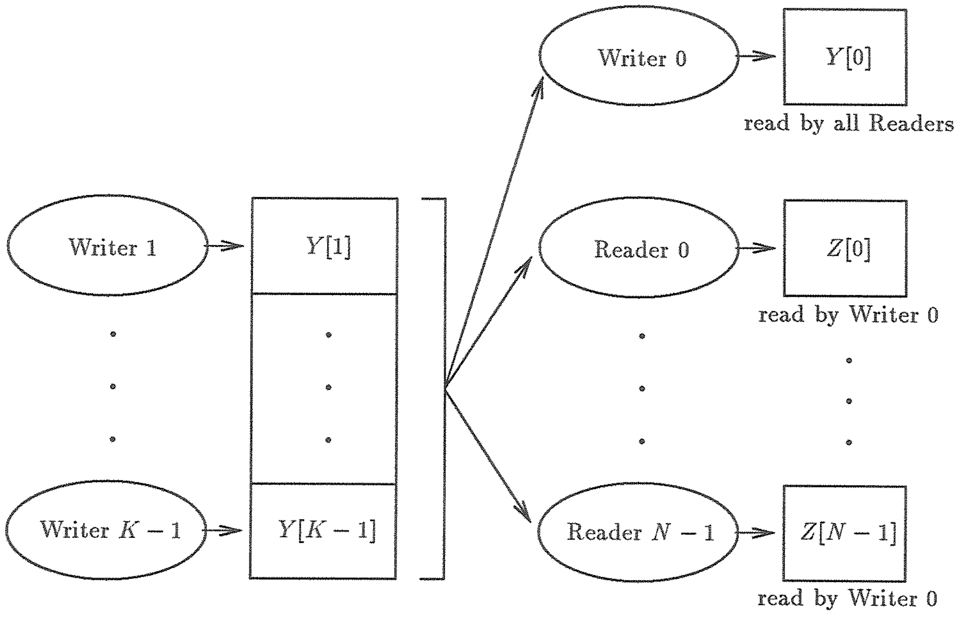

Figure 2: $K/L/1/N$ construction architecture.

```
type itemtype = record   val : valtype;   id : integer /* auxiliary variable */   end;
     Ytype =    record   val : valtype;   id : integer; /* auxiliary variable */
                         z : array[0..1][0..N − 1] of 0..2;
                         ss : array[0..K − 1] of itemtype;   seq : 0..2   end
shared var
     Y[0] : Ytype;
     Y[k] : itemtype, for each k, where 1 ≤ k < K;
     Z : array[0..N − 1] of 0..2
```

Figure 3: Shared variable declarations for $K/L/1/N$ construction.

The values written to $Y[0].z[0]$, $Y[0].seq$, and $Y[0].z[1]$ are used by a Read operation to detect whether it is "overlapped" by an operation of Writer 0.

Reader j also consists of two phases. In its first phase, a new value is computed for $Z[j]$. This value is later used in the second phase of Reader j to detect an "overlapping" operation of Writer 0. In its second phase, Reader j alternates between reading from $Y[0]$ and $Y[1..K-1]$. $Y[0]$ is read three times, and $Y[1..K-1]$ twice. Then, the K return values are computed based upon three cases: (i) there exists an "overlapping" 0-Write operation ($e.z[1,j] = z \ \vee \ e.seq = a.seq \oplus 2$); (ii) the values read from $Y[0]$ at statements 3 and 5, respectively, are written by the same 0-Write operation ($a.seq = c.seq$); and (iii) the values

```
program $Reader(j : 0..N-1)$ returns array$[0..K-1]$ of $valtype$
private var
        $a, c, e, x : Ytype$;
        $b, d :$ array$[1..K-1]$ of $itemtype$;
        $item :$ array$[0..K-1]$ of $itemtype$;
        $z : 0..2$
begin

        /* select new value for $Z[j]$ */
    0:  read $x := Y[0]$;
    1:  select $z$ such that $z \neq x.z[0, j] \wedge z \neq x.z[1, j]$;
    2:  write $Z[j] := z$;

        /* compute $item[0..K-1]$ */
    3:  read $a := Y[0]$;    4:  read $b := Y[1..K-1]$;
    5:  read $c := Y[0]$;    6:  read $d := Y[1..K-1]$;
    7:  read $e := Y[0]$;
    8:  if $e.z[1, j] = z \vee e.seq = a.seq \oplus 2$ then
            $item[0], \ldots, item[K-1] := e.ss[0], \ldots, e.ss[K-1]$
        else if $a.seq = c.seq$ then
            $item[0].val, item[0].id, item[1], \ldots, item[K-1] := a.val, a.id, b[1], \ldots, b[K-1]$
        else /* $c.seq = e.seq$ */
            $item[0].val, item[0].id, item[1], \ldots, item[K-1] := c.val, c.id, d[1], \ldots, d[K-1]$
        fi;
    9:  return$(item[0].val, \ldots, item[K-1].val)$
end
```

Figure 4: Reader program for $K/L/1/N$ construction.

read from $Y[0]$ at statements 5 and 7, respectively, are written by the same 0-Write operation ($c.seq = e.seq$). In case (i), Reader j returns the K values from the "snapshot" of the overlapping 0-Write operation. In case (ii), Reader j returns the values read from $Y[0]$ and $Y[1..K-1]$ at statements 3 and 4, respectively. In case (iii), Reader j returns the values read from $Y[0]$ and $Y[1..K-1]$ at statements 5 and 6, respectively.

The proof of correctness for this construction is based upon the following fact (which is proved as a lemma in Section 3.2): for each Read operation r, if r returns the K values $v_0, \ldots, v_{K-1}$, then there exists a state that occurs "during" r such that $(\forall k : 0 \leq k < K : Y[k].val = v_k)$. In case (i) above, this state exists by virtue of the fact that the overlapping 0-Write operation takes its snapshot "during" r. In case (ii), the existence of this state follows because the value read by r from $Y[0].val$ at statement 3 persists until r reads $Y[0]$ again at statement 5. Thus, at the state prior to the execution of statement 4 by r,

```
program $Writer0(val : valtype)$
private var
        $z$ : array[0..1][0..$N-1$] of 0..2;
        $seq$ : 0..2;
        $item$ : $itemtype$;
        $ss$ : array[0..$K-1$] of $itemtype$;
        $y$ : array[1..$K-1$] of $itemtype$;
        $n$ : 0..$N-1$
initialization
        $seq = Y[0].seq \ \wedge \ item.id = Y[0].id \ \wedge \ (\forall i : 0 \leq i < N : z[1,i] = Y[0].z[1,i]) \ \wedge$
        $(\forall j : 0 \leq j < K : ss[j] = Y[0].ss[j])$
begin

        /* compute $item$, $z[0][0..N-1]$, and $seq$ */
   0:   $seq,\ item.val,\ item.id := seq \oplus 1,\ val,\ item.id + 1$;
   1:   for $n = 0$ to $N-1$ do  2.$n$: read $z[0,n] := Z[n]$ od;
   3:   write $Y[0] := (item.val,\ item.id,\ z[0..1][0..N-1],\ ss[0..K-1],\ seq)$;

        /* compute $z[1][0..N-1]$ and $ss[0..K-1]$ */
   4:   read $y := Y[1..K-1]$;
   5:   $z[1,0],\ \ldots,\ z[1,N-1] := z[0,0],\ \ldots,\ z[0,N-1]$;
   6:   $ss[0],\ ss[1],\ \ldots,\ ss[K-1] := item,\ y[1],\ \ldots,\ y[K-1]$;
   7:   write $Y[0] := (item.val,\ item.id,\ z[0..1][0..N-1],\ ss[0..K-1],\ seq)$
end
```

```
program $Writer(i : 1..K-1; val : valtype)$
private var
        $item : itemtype$
initialization
        $item.id = Y[i].id$
begin
   0:   $item.val,\ item.id := val,\ item.id + 1$;
   1:   write $item$ to $Y[i]$
end
```

Figure 5: Writer programs for $K/L/1/N$ construction.

$Y[0].val = v_0$. By the program for the Reader, $(\forall k : 1 \leq k < K : Y[k].val = v_k)$ also holds at this state. Case (iii) is similar to case (ii).

We now compute the space complexity of our $K/L/1/N$ construction by determining the number of shared $1/1/1/1$ composite registers used in the construction. Let $B(K, L, M, N)$ denote the number of shared $1/1/1/1$ composite registers required to construct a $K/L/M/N$ composite register. If we remove the auxiliary *id* fields from our $K/L/1/N$ construction, then the complexity of each of the shared variables is as follows.

- $Y[0]$ uses $B(1, 4N + KL + L + 2, 1, N)$ bits.
- $Y[1..K-1]$ uses $B(K-1, L, 1, N+1)$ bits.
- $Z[i]$, where $0 \leq i < N$, uses 2 bits.

If we use the construction of [14] to implement a $1/L/1/N$ composite register, then $B(1, L, 1, N) = 6N^2 + 2LN$. Therefore,

$$B(K, L, 1, N) = 14N^2 + 2KLN + 2LN + 6N + B(K-1, L, 1, N+1) \quad .$$

By solving this recurrence, we see that $B(K, L, 1, N) = O(KN^2 + K^2LN + K^3L)$.

We compute the time complexity of our $K/L/1/N$ construction by determining the number of reads and writes of shared $1/L/1/N$ composite registers (i.e., single-writer atomic registers) required to Read and Write the constructed register (for simplicity, we do not go down to the level of $1/1/1/1$ registers when computing the time complexity). Let $TR(K, L, M, N)$ and $TW(K, L, M, N)$ denote the time complexity for Reading and Writing, respectively, a $K/L/M/N$ composite register. Then, the time complexity of a Read in our construction is $TR(K, L, 1, N) = 5 + 2TR(K-1, L, 1, N+1)$. By solving this recurrence, we see that $TR(K, L, 1, N) = O(2^K)$. The time complexity of a Write in our construction is $TW(K, L, 1, N) = N + 2 + TR(K-1, L, 1, N+1)$. By solving this recurrence, we get $TW(K, L, 1, N) = O(N + 2^K)$.

3.2 Correctness Proof

We prove that the $K/L/1/N$ construction is correct by defining functions $\phi_0, \ldots, \phi_{K-1}$ for a given history, and by showing that the defined ϕ's satisfy the five conditions of Uniqueness, Integrity, Proximity, Read Precedence, and Write Precedence given in the Shrinking Lemma.

Notation: In the remainder of this section, we assume that k ranges over $\{0, \ldots, K-1\}$. We use p and q to denote arbitrary operations, r and s to denote Read operations, v and w to denote Write operations, and t and u to denote states. $\square$

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

[ ], ( )

.

!

:

$+, -, \oplus$

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

$\wedge, \vee$

$\equiv$

$\models$ $\Box$

Definition: If event e precedes event f, then we write $e \prec f$. We let $(e \preceq f) \equiv (e = f \vee e \prec f)$. $\Box$

Definition: Let p be an operation, and let x be any private variable of p. Then, $p!x$ denotes the final value of variable x as assigned by operation p. $\Box$

Definition: Let p be an operation of some Reader or Writer program and let i be a label of a statement in that program. We denote the event corresponding to the execution of statement i in operation p by $p\!:\!i$. $\Box$

Definition: If E is an expression that holds at state t, then we write $t \models E$. $\Box$

Assumption: 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. $\Box$

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

Note that the event prior to a given state is uniquely defined since, by assumption, each state appears at most once in a given history.

Definition: Let e be the event corresponding to the execution of the statement **read** $x := Y$ in an operation p, where x is a private variable and Y is a shared variable. If f is the last event to write Y before e, then we say that f *determines* $p!x$. $\Box$

Definition: If p and q are successive operations of the same Reader or Writer program, then we write $p = \mathit{pred}(q)$ and $q = \mathit{succ}(p)$. $\Box$

As mentioned in Section 3.1, 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 $\phi_0, \ldots, \phi_{K-1}$.

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

$$\phi_k(r) \equiv r!item[k].id$$
$$\phi_k(w) \equiv w!item.id \qquad \Box$$

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 $r!a$ is determined by an event of v, and $r!e$ is determined by an event of w. If $r!e.z[1,j] \neq r!z$, then one of the following is true: $w = v$, $w = succ(v)$, or $w = succ(succ(v))$.

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

By the program for the Reader, $r\!:\!3 \prec r\!:\!7$. Therefore, because $r!a$ is determined by an event of v and $r!e$ is determined by an event of w (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 $w = succ(v)$ or $w = succ(succ(v))$.

Assume, to the contrary, that $w \neq succ(v)$ and $w \neq succ(succ(v))$. Then, because v precedes w, there exist 0-Write operations v' and v'' such that $v' = succ(v)$, $v'' = succ(v')$, and v'' precedes w.

Because $r!a$ is determined by an event of v, $r\!:\!3 \prec v'\!:\!3$. Because v' and v'' are successive 0-Write operations, $v'\!:\!3 \prec v''\!:\!0 \prec v''\!:\!7$. Because v'' precedes w, $v''\!:\!7 \prec w\!:\!0$. By the program for Writer 0, $w\!:\!0 \prec w\!:\!3$. Because $r!e$ is determined by an event of w, $w\!:\!3 \prec r\!:\!7$. Therefore,

$$r\!:\!3 \prec v'\!:\!3 \prec v''\!:\!0 \prec v''\!:\!7 \prec w\!:\!0 \prec w\!:\!3 \prec r\!:\!7 \quad .$$

Let $w' = pred(w)$. Then, v'' precedes or equals w'. Hence, by the above precedence assertion, $r\!:\!3 \prec w'\!:\!2.j \prec w\!:\!2.j \prec r\!:\!7$. This implies that $w'!z[0,j]$ and $w!z[0,j]$ are both determined by $r\!:\!2$; thus, $w'!z[0,j] = w!z[0,j] = r!z$. By the program for Writer 0, $w'!z[1,j] = w'!z[0,j]$ and $w!z[1,j] = w!z[0,j]$. Therefore,

$$w'!z[1,j] = w!z[1,j] = r!z \quad . \tag{1}$$

Notice that statement 3 of Writer 0 does not alter the value of $Y[0].z[1,j]$. Hence, if $r!e$ is determined by $w\!:\!3$, then $r!e.z[1,j] = w'!z[1,j]$. On the other hand, if $r!e$ is determined by $w\!:\!7$, then $r!e.z[1,j] = w!z[1,j]$. In either case, by (1), we have $r!e.z[1,j] = r!z$, which is a contradiction. Thus, either $w = succ(v)$ or $w = succ(succ(v))$. $\square$

As mentioned in Section 3.1, the following lemma establishes the existence of a state "during" a Read operation r that corresponds to the "snapshot" taken by r.

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: $r!e.z[1,j] = r!z$. 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, $r!e$ is determined by either $w{:}7$ or $w'{:}3$, where $w' = succ(w)$. Because statement 3 of Writer 0 does not alter the value of $Y[0].z[1,j]$ or $Y[0].ss[0..K-1]$, this implies that

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

By the program for Writer 0, $w!z[0,j] = w!z[1,j]$. By assumption, $r!e.z[1,j] = r!z$; therefore, by (2),

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

We now show that $r{:}0 \prec w{:}3$. Assume, to the contrary, that $w{:}3 \prec r{:}0$. By the program for the Reader, $r{:}0 \prec r{:}7$. Therefore, $w{:}3 \prec r{:}0 \prec r{:}7$. Because $r!e$ is determined by either $w{:}7$ or $w'{:}3$, this precedence assertion implies that $r!x$ is determined by $w{:}3$, $w{:}7$, or $w'{:}3$. If $r!x$ is determined by $w{:}3$, then $r!x.z[0,j] = w!z[0,j]$. By (3), this implies that $r!z = r!x.z[0,j]$. But, by the program for the Reader, $r!z \neq r!x.z[0,j]$; therefore, we have a contradiction.

If, on the other hand, $r!x$ is determined by either $w{:}7$ or $w'{:}3$, then because statement 3 of Writer 0 does not alter the value of $Y[0].z[1,j]$, $r!x.z[1,j] = w!z[1,j]$. By (3), this implies that $r!z = r!x.z[1,j]$. But, by the program for the Reader, $r!z \neq r!x.z[1,j]$; therefore, we have a contradiction. Thus, our assumption that $w{:}3 \prec r{:}0$ is false, i.e., $r{:}0 \prec w{:}3$.

Because $r!e.z[1,j] = r!z$, by the program for the Reader, $(\forall k :: r!item[k] = r!e.ss[k])$. 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 program 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 the program 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 the program 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: $r!e.z[1,j] \neq r!z \wedge r!e.seq = r!a.seq \oplus 2$. Assume that $r!a$ is determined by an event of 0-Write operation v and $r!e$ is determined by an event of 0-Write operation w. (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 programs for the Reader and Writer 0,

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

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

Let $w' = succ(v)$. Because $r!a$ is determined by an event of v, $r\!:\!3 \prec w'\!:\!3$. By the program for Writer 0, $w'\!:\!3 \prec w'\!:\!4 \prec w'\!:\!7$. Because $w = succ(w')$, $w'\!:\!7 \prec w\!:\!0$. By the program for Writer 0, $w\!:\!0 \prec w\!:\!3$. Because $r!e$ is determined by an event of w, $w\!:\!3 \prec r\!:\!7$. Therefore,

$$r\!:\!3 \prec w'\!:\!3 \prec w'\!:\!4 \prec w'\!:\!7 \prec w\!:\!0 \prec w\!:\!3 \prec r\!:\!7 \quad . \tag{6}$$

We now show that $r!e$ is determined by $w\!:\!3$. By (6), $w!z[0,j]$ is determined by $r\!:\!2$. Therefore, $w!z[0,j] = r!z$. By the program for Writer 0, $w!z[1,j] = w!z[0,j]$; hence, by transitivity, $w!z[1,j] = r!z$. By assumption, $r!e$ is determined by either $w\!:\!3$ or $w\!:\!7$. In the latter case, $r!e.z[1,j] = w!z[1,j]$. Therefore, by transitivity, $r!e.z[1,j] = r!z$. However, we have assumed in Case 2 that $r!e.z[1,j] \neq r!z$. Hence, $r!e$ is determined by $w\!:\!3$.

Since statement 3 of Writer 0 does not alter the value of $Y[0].ss[0..K-1]$, this implies that $(\forall k :: r!e.ss[k] = w'!ss[k])$. Because $r!e.z[1,j] \neq r!z \wedge r!e.seq = r!a.seq \oplus 2$, by the program for the Reader, $(\forall k :: r!item[k] = r!e.ss[k])$. Therefore, $(\forall k :: r!item[k] = w'!ss[k])$. By the definition of ϕ_k, $(\forall k :: \phi_k(r) = r!item[k].id)$. Therefore,

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

We now establish the existence of the required state. Let t be the state prior to $w'\!:\!4$. By (6), t occurs between $r\!:\!0$ and $r\!:\!9$. By the program 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 (7), this implies that $t \models Y[0].val = r!item[0].val \wedge Y[0].id = \phi_0(r)$. By the program 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 (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: $r!e.z[1,j] \neq r!z \ \wedge\ r!e.seq \neq r!a.seq \oplus 2 \ \wedge\ r!a.seq = r!c.seq$. Let v and w be as defined in Case 2, i.e., $r!a$ is determined by an event of v and $r!e$ is determined by an event of w. Assume that $r!c$ is determined by an event of 0-Write operation v'. Then, by the programs for the Reader and Writer 0,

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

We first show that $v = v'$. Because $r!a$ is determined by an event of v, $r!c$ by an event of v', and $r!e$ by an event of w, by the program for the Reader (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: $w = v$, $w = succ(v)$, or $w = succ(succ(v))$. This implies that one of the following holds as well: $v' = v$, $v' = succ(v)$, or $v' = succ(succ(v))$. Because $r!a.seq = r!c.seq$, by (8), we have $v!seq = v'!seq$. Thus, because each 0-Write operation assigns $seq := seq \oplus 1$, and because $\oplus$ is modulo-3 addition, $v' \neq succ(v)$ and $v' \neq succ(succ(v))$. Therefore, $v' = v$.

Because $r!e.z[1,j] \neq r!z \ \wedge\ r!e.seq \neq r!a.seq \oplus 2 \ \wedge\ r!a.seq = r!c.seq$, by the program for the Reader, $r!item[0].val = r!a.val$, $r!item[0].id = r!a.id$, 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 $r!a$ and $r!c$ are both determined by an event of v (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 the program for the Reader, $t \models (\forall k : k > 0 : Y[k] = r!b[k])$. 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: $r!e.z[1,j] \neq r!z \ \wedge\ r!e.seq \neq r!a.seq \oplus 2 \ \wedge\ r!a.seq \neq r!c.seq$. Let v, v', and w be as defined in Case 3, i.e., $r!a$ is determined by an event of v, $r!c$ is determined by an event of v', and $r!e$ is determined by an event of w.

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: $w = v$, $w = succ(v)$, or $w = succ(succ(v))$. Because $r!e.seq \neq r!a.seq \oplus 2$, by (8), we have $w!seq \neq v!seq \oplus 2$. Because each 0-Write operation assigns $seq := seq \oplus 1$, this implies that $w \neq succ(succ(v))$. Because $r!a.seq \neq r!c.seq$, by (8), we have $v!seq \neq v'!seq$. This implies that $v \neq v'$. Therefore, v precedes v', v' precedes or equals w, and either $w = v$ or $w = succ(v)$. This implies that $v' = w$.

Because $r!e.z[1,j] \neq r!z \ \wedge \ r!e.seq \neq r!a.seq \oplus 2 \ \wedge \ r!a.seq \neq r!c.seq$, by the program for the Reader, $r!item[0].val = r!c.val$, $r!item[0].id = r!c.id$, 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,

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

We now establish the existence of the required state. Let t be the state prior to $r\!:\!6$. Because $r!c$ and $r!e$ are both determined by an event of w (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 the program for the Reader, $t \ \models \ (\forall k : k > 0 : Y[k] = r!d[k])$. By (10), this implies that $t \ \models \ (\forall k : k > 0 : Y[k].val = r!item[k].val \ \wedge \ Y[k].id = \phi_k(r))$. $\square$

We now show that the five conditions given in the Shrinking Lemma are satisfied. The preceding lemma is used in the proofs of Integrity, Read Precedence, and Write Precedence.

Proof of 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. $\square$

Proof of 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

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

Let $0 \leq k < K$, and suppose that the last event to write to $Y[k]$ before state t is an event of Write operation v. (v exists because, by assumption, there is a k-Write operation that precedes every Read operation.) Then, by the program 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 $\phi_k(v) = \phi_k(r)$. $\square$

Proof of 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 the last event to write to $Y[k]$ before state t is an event of v, and because state t occurs between $r\!:\!0$ and $r\!:\!9$, r does not precede v. Therefore, because r precedes and 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 the last event to write to $Y[k]$ before state t is an event of v, w precedes or equals v. Therefore, by Uniqueness, $\phi_k(w) \leq \phi_k(v)$. Thus, $\phi_k(w) \leq \phi_k(r)$. □

Proof of Read Precedence: The proof of Read Precedence is based upon the following property: if state t occurs before state u, then for each k, the value of $Y[k].id$ at state u is at least its value at state t. 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 the above property, $(\forall k :: \phi_k(r) \leq \phi_k(s))$. If state u occurs before state t, then $(\forall k :: \phi_k(s) \leq \phi_k(r))$. This implies that Read Precedence is satisfied. □

Proof of 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 $0 \leq i < K$ and $0 \leq j < K$. 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 $v!item.id \leq \phi_i(r)$.

As in the proof of Read Precedence, we use the following property: if state u occurs before state u', then for each k, the value of $Y[k].id$ at state u' is at least its value at state u. By Lemma 2, there exists a state t between the first and last events of r such that

$$t \models (\forall k :: Y[k].id = \phi_k(r)) \quad . \tag{12}$$

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 program 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 the property mentioned above, 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'' either equals or occurs before t'. Therefore, t'' occurs before t. Thus, by the property stated above, 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 (12), we have $t \models Y[i].id = \phi_i(r)$. Therefore, $v!item.id \leq \phi_i(r)$. This establishes our proof obligation. □

4 Concluding Remarks

The construction of this paper, together with the one in [2], shows that we can allow an atomic operation of a concurrent program to either write a single shared variable or read several shared variables (but not both) and the resulting program can be implemented from atomic registers. By contrast, if we allow an atomic operation of a program to either write several shared variables, or to both read and write shared variables, then, in general, such a program cannot be implemented from atomic registers. This result has been proved both by Herlihy [6] and by Anderson and Gouda [3].

Our $K/L/1/N$ construction and the construction of Afek et al. [1, 10] are both based upon the following insight: if a Read operation is overlapped by "too many" Write operations, then it returns the K 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 K components. Afek et al. do not resort to recursion, and as a result, their solution is polynomial in both space and time. As mentioned earlier, the $K/L/M/N$ construction of Afek et al. is based upon multiple-writer atomic registers. By contrast, our $K/L/M/N$ construction (which is obtained by using the construction of [2] along with the one in this paper) uses only single-writer atomic registers. Thus, our $K/L/M/N$ construction can be used to construct a multiple-writer atomic register (the case in which there is only one component).

As stated in the introduction, composite registers can be used to implement a shared variable that can either be read or incremented in one atomic step. A variable that can be incremented by K processes and read by N processes can be implemented by using a single $K/L/1/N$ composite register. Each process that can increment the variable writes to a particular component of the composite register; to increment the value of the variable, a process increments the value of its component. (Since there is only one writer per component, a process can increment the value of its component by maintaining a local copy of its component. Thus, an increment operation can be performed without reading any shared variable.) A process reads the value of the variable by reading all of the components of the composite register and adding together their values.

This approach can be used to implement any operation that can be defined in terms of an operator that is both commutative and associative. For example, because addition is commutative and associative, we can implement operations that increment or decrement. Because multiplication is commutative and associative, we can implement an operation that multiplies by a constant, or one that shifts a string of bits (shifting can be defined in terms of multiplying by 2).

Acknowledgements: I would like to thank Anish Arora, Ken Calvert, and Mohamed Gouda for their comments on an earlier draft of this paper.

Appendix: Proof of the Shrinking Lemma

Shrinking Lemma: A history h is atomic if for each k, where $0 \le k < K$, 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 $0 \le k < K$, there exists a k-Write operation w in h such that $\phi_k(r) = \phi_k(w)$. Furthermore, the value Read by r for component k is the same as the value Written by 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 $\phi_k(w) \le \phi_k(r)$.
- *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 $(\forall k :: \phi_k(r) \le \phi_k(s))$.
- *Write Precedence*: For each Read operation r in h, and each j-Write operation v and k-Write operation w in h, where $0 \le j < K$ and $0 \le k < K$, if v precedes w and $\phi_k(w) \le \phi_k(r)$, then $\phi_j(v) \le \phi_j(r)$.

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 "atomic 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 $\{0, \ldots, K-1\}$. If x precedes y in h, then we write $x \lhd y$. We let $(x \unlhd y) \equiv (x = y \lor x \lhd y)$. 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 $\phi_k(w) \le \phi_k(r)$, 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 $(\exists r :: vBr \land rBw)$.
- E includes all pairs (v, w), such that $v \neq w$ and for some v' and w',

$$\phi_j(v) \le \phi_j(v') \land v' \unlhd w' \land \phi_k(w') \le \phi_k(w) \quad .$$

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

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 obligated 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) totally orders each Read with respect to all Writes, we have $x \neq y$. If xEy, then by the definition of E, $x \neq y$. Therefore, we conclude that relation 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, $(\forall k :: \phi_k(r) \leq \phi_k(s))$. $\square$

Property 2: For each Read operation r and k-Write operation v, $rFv \Rightarrow \phi_k(r) < \phi_k(v)$ and $vFr \Rightarrow \phi_k(v) \leq \phi_k(r)$.

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

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. If $j \neq k$, then $v \neq w$. If $j = k$, then by Uniqueness, $\phi_j(v) < \phi_j(w)$, which implies that $v \neq w$. 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. $\square$

To prove that F is transitive, we are obligated 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

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

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

<u>Case 8</u>: 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 exist an i-Write operation v, j-Write operations w and v', and a k-Write operation w' such that

$$x \ne y \ \wedge \ \phi_i(x) \le \phi_i(v) \ \wedge \ v \trianglelefteq w \ \wedge \ \phi_j(w) \le \phi_j(y)$$

and

$$y \ne z \ \wedge \ \phi_j(y) \le \phi_j(v') \ \wedge \ v' \trianglelefteq w' \ \wedge \ \phi_k(w') \le \phi_k(z) \quad .$$

There are three possibilities to consider: $i = j$, $j = k$, and $i \ne j \ \wedge \ j \ne k$. First, suppose that $i = j$. We show that xEz holds by first proving that $\phi_j(x) < \phi_j(v')$. Because $v \trianglelefteq w$, by Uniqueness, $\phi_j(v) \le \phi_j(w)$. Therefore, by transitivity, $\phi_j(x) \le \phi_j(y)$. Because $x \ne y$, Uniqueness implies that $\phi_j(x) \ne \phi_j(y)$. Thus, $\phi_j(x) < \phi_j(y)$. Thus, by transitivity, $\phi_j(x) < \phi_j(v')$.

We now show that xEz. If $i \ne k$, then because x is an i-Write operation and z is a k-Write operation, $x \ne z$. If $i = k$, then by Uniqueness, $\phi_j(v') \le \phi_j(w')$; thus, by transitivity, $\phi_j(x) < \phi_j(z)$, which implies that $x \ne z$. Therefore, we conclude for the case $i = j$ that

$$x \ne z \ \wedge \ \phi_j(x) < \phi_j(v') \ \wedge \ v' \trianglelefteq w' \ \wedge \ \phi_k(w') \le \phi_k(z) \quad .$$

Thus, xEz.

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

Now suppose that $i \ne j$ and $j \ne k$. In this case, we prove that xEz by first showing that $v \lhd w'$. Because $i \ne j$ and because v is an i-Write operation and w a j-Write operation, we have $v \ne w$. Similarly, because $j \ne k$, we have $v' \ne w'$. Hence, $v \lhd w$ and $v' \lhd w'$. Observe that, by transitivity, $\phi_j(w) \le \phi_j(v')$. Therefore, by Uniqueness, $\neg(v' \lhd w)$. Also, observe that $v \lhd w \ \wedge \ v' \lhd w' \ \Rightarrow \ v' \lhd w \ \vee \ v \lhd w'$. Thus, $v \lhd w'$. Hence, the following expression holds.

$$\phi_i(x) \le \phi_i(v) \ \wedge \ v \lhd w' \ \wedge \ \phi_k(w') \le \phi_k(z)$$

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 an atomic 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

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

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 value Written by v is the same as the value Read by 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 an atomic history. □

References

[1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit, Atomic snapshots, *Proceedings of the Ninth Annual Symposium on Principles of Distributed Computing*, 1990, to appear.

[2] J. Anderson, Multiple-writer composite registers, *Technical Report TR.89.26*, Department of Computer Sciences, University of Texas at Austin, 1989.

[3] J. Anderson and M. Gouda, The virtue of patience: concurrent programming with and without waiting, unpublished manuscript.

[4] B. Bloom, Constructing two-writer atomic registers, *IEEE Transactions on Computers*, vol. 37, no. 12, December 1988, pp. 1506-1514. Also appeared in *Proceedings of the Sixth Annual Symposium on Principles of Distributed Computing*, 1987, pp. 249-259.

[5] 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.

[6] M. Herlihy, Wait-free implementation of concurrent objects, *Proceedings of the Seventh Annual Symposium on Principles of Distributed Computing*, 1988.

[7] M. Herlihy and J. Wing, Axioms for concurrent objects, *Proceedings of the 4th Annual ACM Symposium on Principles of Programming Languages*, 1987.

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

[9] L. Lamport, On interprocess communication, parts I and II, *Distributed Computing*, vol. 1, pp. 77-101, 1986.

[10] M. Merritt, private communication, 1990.

[11] 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.

[12] G. Peterson, Concurrent reading while writing, *ACM Transactions on Programming Languages and Systems*, vol. 5, pp. 46-55, 1983.

[13] 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.

[14] 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.

[15] P. Vitanyi and B. Awerbuch, Atomic shared register access by asynchronous hardware, *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, 1986.