Beyond Atomic Registers: Bounded Wait-Free Implementations of Nontrivial Objects*

James H. Anderson† Bojan Grošelj‡

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

January 1991
Revised September 1991, June 1992

Abstract

We define a class of operations called *pseudo read-modify-write* (PRMW) operations, and show that nontrivial shared data objects with such operations can be implemented in a bounded, wait-free manner from atomic registers. A PRMW operation is similar to a "true" read-modify-write (RMW) operation in that it modifies the value of a shared variable 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. We consider a class of shared data objects that can either be read, written, or modified by an associative, commutative PRMW operation, and show that any object in this class can be implemented without waiting from atomic registers. The implementations that we present are polynomial in both space and time and thus are an improvement over previously published ones, all of which have unbounded space complexity.

Keywords: assertional reasoning, atomicity, atomic register, composite register, concurrency, counter, linearizability, read-modify-write, shared variable, snapshot, UNITY

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

---

*To appear in *Science of Computer Programming*. Preliminary version was presented at the Fifth International Workshop on Distributed Algorithms, Delphi, Greece, October 1991.

†Work supported in part by an award from the General Research Board, University of Maryland, and by NSF Contract CCR 9109497. E-mail: jha@cs.umd.edu.

‡On leave from: The Center for Advanced Computer Studies, University of Southwestern Louisiana, Lafayette, Louisiana 70504. E-mail: bojan@cs.umd.edu.

1 Introduction

The implementation of 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. Traditionally, shared data objects have been implemented by using mutual exclusion, with each operation corresponding to a "critical section." Although conceptually simple, such implementations suffer from two serious shortcomings. First, they are not very resilient: if a process experiences a halting failure while accessing such a data object, then the data object may be left in a state that prevents subsequent accesses by other processes. Second, such implementations may unnecessarily restrict parallelism. This is especially undesirable if operations are time-consuming to execute.

As a result of these two shortcomings, there has been much interest recently in wait-free implementations of shared data objects. 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.

One of the major objectives of researchers in this area has been to characterize those shared data objects that can be implemented without waiting in terms of single-reader, single-writer, single-bit atomic registers. An *atomic register* is a shared data object consisting of a single shared variable that can either be read or written in a single operation [21]. An N-reader, M-writer, L-bit atomic register consists of an L-bit variable that can be read by N processes and written by M processes. It has been shown in a series of papers that multi-reader, multi-writer, multi-bit atomic registers can be implemented without waiting in terms of single-reader, single-writer, single-bit atomic registers [6, 9, 10, 17, 18, 21, 22, 24, 26, 27, 28, 29, 30]. This work shows that, using only atomic registers of the simplest kind, it is possible to solve the classical readers-writers problem without requiring either readers or writers to wait [14].

Another shared data object of interest is the composite register, a data object 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. As illustrated in Figure 1, an operation of such a register either writes a value to a single component, or reads the values of all components. Afek et al. [2] and Anderson [3, 4] have shown that composite registers can be implemented from atomic registers without waiting. This work shows that, using only atomic registers of the simplest kind, it is possible to implement a shared memory that can be read in its entirety in a single "snapshot" operation, without resorting to mutual exclusion.

In this paper, we consider the important question of whether there exist other nontrivial shared data objects that can be implemented from atomic registers without waiting. We define a class of operations called *pseudo read-modify-write* (PRMW) operations and consider a corresponding class of shared data objects called *PRMW objects*. This class of objects includes such fundamental objects as counters, shift registers, and multiplication registers. A PRMW object consists of a single shared variable that can either be read, written, or modified by an associative, commutative PRMW operation. The PRMW operation takes its name from the classical read-modify-write (RMW) operation as defined in [19]. The RMW operation has the form "$temp,\ X := X,\ f(X)$,"

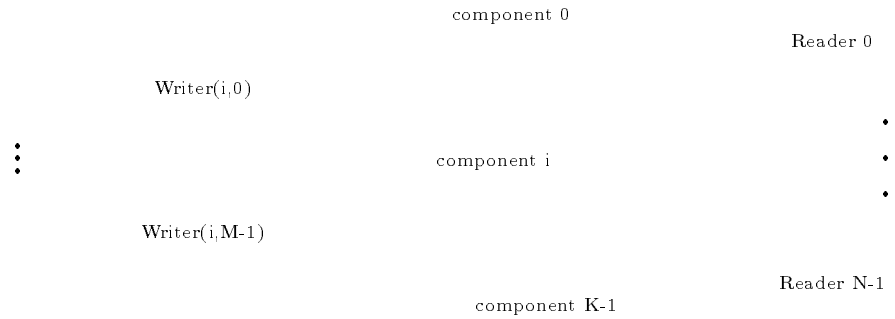

Figure 1: Composite register structure (only writers for component i are depicted).

where X is a shared variable, *temp* is a private variable, and f is a function. Executing this operation has the effect of modifying the value of X according to f, and returning the original value of X in *temp*. The PRMW operation has the form "$X := f(X)$," and differs from the RMW operation in that the value of X is not returned.

We prove that any PRMW object can be implemented from atomic registers in a wait-free manner. We establish this result by first considering the problem of implementing a counter without waiting. A *counter* is a PRMW object whose value can be read, written, or incremented by an integer value.[1] We first show that counters can be implemented from composite registers without waiting, and then show that our implementation can be generalized to apply to any PRMW object. Given the results of [2, 3, 4], this shows that any PRMW object can be implemented without waiting using only atomic registers. Our results stand in sharp contrast to those of [5, 15], where it is shown that RMW operations cannot, in general, be implemented from atomic registers without waiting.

The problem of implementing PRMW objects without waiting has been studied independently by Aspnes and Herlihy in [7]. Aspnes and Herlihy give a general, wait-free implementation that can be used to implement any PRMW object. A counter implementation, which is obtained by optimizing the general implementation, is also given. Both of these implementations have unbounded space complexity: the first uses a graph of unbounded size to represent the history of the implemented data object, and the second uses unbounded timestamps. Our counter implementation and its generalization are polynomial in space and time.

The rest of the paper is organized as follows. In the next section, we formally define the problem of implementing a counter from composite registers. The counter implementation mentioned above is described in Section 3. The correctness proof for the implementation is given in Section 4. In contrast to almost every other paper in the literature on wait-free algorithms (including some written by the first author), our proof is assertional rather than operational. Thus, as a secondary contribution, this paper serves as an example of how to apply assertional techniques in reasoning about wait-free implementations, a task that is complicated

[1] Note that decrementing can be defined in terms of incrementing; thus, a counter actually supports four operations: read, write, increment, and decrement.

by the fact that the primary correctness condition for such implementations (i.e., linearizability [16]) refers to operational concepts such as histories, operations, and precedence relationships. In Section 5, we discuss several issues pertaining to our implementation, and show that the implementation can be generalized to implement any PRMW object. Concluding remarks appear in Section 6.

2 Problem Definition

In this section, we consider the problem of implementing a counter from composite registers, and give the conditions that such an implementation must satisfy to be correct. An implementation consists of a set of N processes along with a set of variables. Each process is a sequential program comprised of atomic statements. Informally, an atomic statement is a language construct whose execution is semantically indivisible. We assume a repertoire of atomic statements that is typical of most sequential programming languages. We do not give a complete list of such statements, but do give a restriction below that limits the manner in which atomic statements may affect the variables of an implementation.

Each process of an implementation consists of a main program and three procedures, called *Read*, *Write*, and *Increment*. Each such procedure has the following form:

procedure *name*(*inputs*)
    body;
    return(*outputs*)
end

where *name* is the name of the procedure, *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. The Read, Write, and Increment procedures of a process constitute its interface with the implemented counter. The Read procedure is invoked to read the value of the counter; the value read is returned as an output parameter. The Write procedure is invoked to write a new value to the counter; the value to be written is specified as an input parameter. The Increment procedure is invoked to increment the value of the counter; the value to add is given as an input parameter. A process invokes its procedures only from its main program. We leave the exact structure of each process's main program unspecified, but do assume that each process repeatedly invokes its three procedures in an arbitrary, serial manner. As an example of the syntax we use for specifying the variables and procedures of an implementation, see Figures 2 and 3.

Each variable of an implementation is either private or shared. A *private variable* is defined only within the scope of a single process, whereas a *shared variable* is defined globally and may be accessed by more than one process. For simplicity, we stipulate that each process may access shared variables only within its Read, Write, and Increment procedures, and not within its main program. The procedures and variables of each process are required to satisfy the following two restrictions.

- *Atomicity Restriction:* Each shared variable is required to correspond to a component of some composite register. Thus, each atomic statement within a procedure may either write a single shared variable or read one or more shared variables, but not both. In the latter case, the shared variables must all correspond to the components of a single composite register.

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

We now define several concepts that are needed to state the correctness condition for an implementation. These definitions apply to a given implementation. A *state* is an assignment of values to the variables of the implementation. (Each process's "program counter" is considered to be a private variable of that process.) One or more states are designated as *initial states*. Each execution of an atomic statement of a process is called an *event*. 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 implementation 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. We assume that in any history, the first event of each process occurs as the result of executing an atomic statement in that process's main program. (In other words, each process's program counter must be initialized so that it equals a location within the main program of that process.)

The subsequence of events in a history corresponding to a single procedure invocation is called an *operation*. An operation of a Read (respectively, Write or Increment) procedure is called a *Read operation* (respectively, *Write operation* or *Increment operation*).[2] Each history defines an irreflexive partial order on operations: an operation p precedes another operation q in this ordering iff each event of p occurs before all events of q in the history.

As mentioned above, each Read procedure has an output parameter that returns the value read from the counter; the value returned by a Read operation is called the *output value* of that operation. As also mentioned above, each Write (Increment) procedure has an input parameter that specifies the value to be written (added) to the counter; the value written (added) to the counter by a Write (Increment) operation is called the *input value* of that operation. We designate one integer value as the *initial value* of the implemented counter.

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 define what it means for a history of an implementation to be "linearizable." Linearizability provides the illusion that each operation is executed instantaneously, despite the fact that it is actually executed as a sequence of events. It can be shown that the following definition is equivalent to the more general definition of linearizability given by Herlihy and Wing in [16], when restricted to the special case of implementing a counter.

Linearizable Histories: A well-formed history h of an implementation is *linearizable* iff the partial order on operations defined by h can be extended to a total order $\prec$ such that for each Read operation r in h, the following condition is satisfied.

- If there exists a Write operation w such that $w \prec r \ \wedge \ \neg\langle \exists v : v \text{ is a Write operation} :: w \prec v \prec r \rangle$,[3] then the output value of r equals that obtained by adding the input value of w to the sum of the input values of all Increment operations ordered between w and r by $\prec$.

[2] In order to avoid confusion, we capitalize the terms "Read," "Write," and "Increment" when referring to the operations of the implemented counter, and leave them uncapitalized when referring to the variables used in the implementation.

[3] The notation of [11] is used for quantification. Thus, for example, $\langle \sum\ j : B(j) :: E(j) \rangle$ denotes the sum $\sum_{j \text{ s.t. } B(j)} E(j)$.

type

$Tagtype$ = **record** $seq : 0..N+1$; $pnum : 0..N-1$ **end**;

$Qtype$ = **record** val: **integer**; $tag : Tagtype$ **end**;

$Htype$ = **record** op: (READ, WRITE, INC); $pnum : 0..N-1$; id : **integer**; val : **integer end**

shared var

Q : **array**$[0..N]$ **of** $Qtype$; { $(N+1)$-component composite register }

H, S : **sequence of** $Htype$; { Auxiliary history variables }

$Ovlap$: **array**$[0..N-1]$ **of boolean** { Auxiliary flag: indicates if an Increment is overlapped by a Write }

initially

$\langle \forall j : 0 \leq j < N :: Q[j].val = 0 \ \wedge \ Q[j].tag = (0, j) \rangle \ \wedge \ Q[N].val = init \ \wedge \ Q[N].tag = (0,0) \ \wedge \ H = S = \emptyset$

Figure 2: Shared variable declarations.

- If no such w exists, then the output value of r equals that obtained by adding the initial value of the counter to the sum of the input values of all Increment operations ordered before r by $\prec$. □

An implementation of a counter is *correct* iff it satisfies the Atomicity and Wait-Freedom restrictions and each of its well-formed histories is linearizable.

3 Counter Implementation

In this section, we present our counter implementation. For now, we assume that the counter stores values ranging over the integers. Later, in Section 5, we consider the case in which the counter stores values over some bounded range. (In the latter case, overflow is a problem.)

The shared variable declarations for the implementation are given in Figure 2 and the procedures for process i, where $0 \leq i < N$, are given in Figure 3. Central to the proof of the implementation is the "history variable" H, which is defined in Figure 2. H is used to totally order the operations of the implemented counter, and is one of several auxiliary variables that are used to facilitate the proof of correctness. H is a sequence of tuples of the form $(op,\ pnum,\ id,\ val)$, where op ranges over {READ, WRITE, INC}, $pnum$ ranges over $0..N-1$, and id and val are integers. Intuitively, H is a "log" of operations that have been performed on the implemented counter, and each tuple records the "effect" of a specific operation. The type of the particular operation is identified by the op field, the process invoking the operation is identified by the $pnum$ field ($pnum$ stands for "process number"), and the id field is used to differentiate between operations of the same type by the same process. The val field is used to record the output value of a Read operation or the input value of a Write or Increment operation. The following notational conventions regarding history variables will be used in the remainder of the paper.

Notational Conventions for History Variables: If X and Y are sequences of tuples, then $X \cdot Y$ denotes the sequence obtained by appending Y onto the end of X. If X is a prefix of Y, then we write $X \sqsubseteq Y$. If z is a tuple, then $X - z$ denotes the sequence obtained by removing all occurrences of z from X. (Note that there

```
private var                                  { Private variables for process i }
        x : Qtype;                           { Local copy of Q }
        seq : 0..N + 1;                      { Sequence number }
        tag : Tagtype;                       { Tag value for Write or Increment operation }
        outval, sum, id : integer;           { Output value; sum of increment input values; auxiliary operation id }
        tuple : Htype                        { Auxiliary variable for recording INC tuple }

procedure Read() returns integer
    0:  read x := Q;                         { Take snapshot; compute output value; update history variable }
        outval := x[N].val + ⟨∑ j : 0 ≤ j < N  ∧  x[j].tag = x[N].tag :: x[j].val⟩;
        H, id := H · (READ, i, id, outval), id + 1;
    1:  return(outval)
end  { Read }

procedure Write(inval : integer)             { Input value passed as a parameter }
    2:  read x := Q;                         { Take snapshot; compute new sequence number and tag }
        select seq such that ⟨∀ j : 0 ≤ j ≤ N :: seq ≠ x[j].tag.seq⟩;      { Such a seq exists }
        tag := (seq, i);
    3:  write Q[N] := (inval, tag);          { Write base component; move pending INC tuples to H; set "overlap" flags }
        H, S, id, Ovlap[0], ..., Ovlap[N − 1] :=
                                  H · S · (WRITE, i, id, inval), ∅, id + 1, true, ..., true;
    4:  return
end  { Write }

procedure Increment(inval : integer)         { Input value passed as a parameter }
        { First phase: read Q and write Q[i] }
    5:  read x := Q;                         { Take snapshot; copy tag; add tuple to list of pending INC tuples }
        tag := x[N].tag;
        S, tuple, id := S · (INC, i, id, inval), (INC, i, id, inval), id + 1;
        { Compare new tag with tag of last Increment operation }
        if x[i].tag = tag then sum := x[i].val else sum := 0 fi;
    6:  write Q[i] := (sum, tag);            { Write increment component }

        { Second phase: read Q and write Q[i] }
    7:  read x := Q;                         { Take snapshot; copy tag; initialize auxiliary "overlap" flag }
        tag := x[N].tag;
        Ovlap[i] := false;
        { Compare tag from first phase with tag from second phase }
        if x[i].tag = tag then sum := x[i].val + inval else sum := 0 fi;
    8:  write Q[i] := (sum, tag);            { Write increment component; update history variable if necessary }
        if tuple ∉ H  ∨  (¬Ovlap[i]  ∧  x[i].tag = tag) then H, S := (H − tuple) · tuple, S − tuple fi;
    9:  return
end  { Increment }
```

Figure 3: Procedures for process i.

may be no occurrences of z in X, in which case $X - z = X$.) The symbol $\emptyset$ is used to denote the empty sequence. □

According to the semantics of a counter, Write and Increment operations change the value of the implemented counter, whereas Read operations do not. This is reflected in the definition of the function *Val*, given next. This function gives the "value" of the implemented counter as recorded by a sequence of READ, WRITE, and INC tuples.

Definition of *Val*: Let i range over $0..N-1$, let n and v be integer values, let *init* be the initial value of the implemented counter, and let α be a sequence of READ, WRITE, and INC tuples. Then, the function *Val* is defined as follows:

$$\begin{aligned} Val(\alpha \cdot (\text{READ},\ i,\ n,\ v)) &\equiv Val(\alpha) \\ Val(\alpha \cdot (\text{WRITE},\ i,\ n,\ v)) &\equiv v \\ Val(\alpha \cdot (\text{INC},\ i,\ n,\ v)) &\equiv Val(\alpha) + v \\ Val(\emptyset) &\equiv init \end{aligned}$$

□

The proof of correctness is based upon the following lemma.

Lemma: If the following two conditions hold, then each well-formed history of the implementation is linearizable.

- *Ordering*: During the execution of each operation (i.e., between its first and last events), an event occurs that appends a single, unique tuple for that operation to H, and this tuple is not subsequently removed from H.
- *Consistency*: The tuples in H are consistent with the semantics of a counter. More precisely, the following assertion is an invariant: $\langle \forall \alpha :: \alpha \cdot (\text{READ},\ i,\ n,\ v) \sqsubseteq H \Rightarrow v = Val(\alpha) \rangle$. □

To see why this lemma holds, consider a well-formed history h. Let α denote the "final" value of H in h: i.e., if h is finite, then $H = \alpha$ in the final state of h, and if h is infinite, then α is infinite and every finite prefix of α is a prefix of H for some infinite sequence of states in h. Define a total order $\prec$ on the operations in h as follows: $p \prec q$ iff p's tuple occurs before q's tuple in α. By the Ordering condition, $\prec$ extends the partial precedence ordering on operations defined by h. By the Consistency condition and the definition of *Val*, $\prec$ is consistent with the semantics of a counter. That is, the output value of each Read operation in h equals that obtained by adding the input value of the most recent Write operation according to $\prec$ (or the initial value of the implemented counter if there is no preceding Write operation) to the sum of the values of all intervening Increment operations according to $\prec$. This implies that h is linearizable.

We justify the correctness of the implementation below by informally arguing that the Ordering and Consistency conditions are satisfied. A formal proof of Consistency (which turns out to be the most significant proof obligation) is given in the next section. Before proceeding, several comments concerning notation are in order.

Notational Conventions for Implementations: Each initial state of the implementation is required to satisfy the **initially** assertion given in Figure 2. (If a given variable is not included in the **initially** assertion, then its initial value is arbitrary. Note that each private variable has an arbitrary initial value.) As in the definition of Val, we use $init$ to denote the initial value of the implemented counter. To make the implementation easier to understand, the keywords **read** and **write** are used to distinguish reads and writes of (nonauxiliary) shared variables from reads and writes of private variables. To simplify the implementation, each labeled sequence of statements is assumed to be a single atomic statement. (Each such sequence can easily be implemented by a single multiple-assignment.) □

Each of the labeled atomic statements in Figure 3 satisfies the Atomicity restriction of Section 2. In particular, no statement writes more than one (nonauxiliary) shared variable, and no statement both reads and writes (nonauxiliary) shared variables. The Wait-Freedom restriction is also satisfied, since each procedure contains no unbounded loops or idle-waiting primitives. With regard to the Atomicity restriction, it should be emphasized that auxiliary variables are irrelevant. These variables are not to be implemented, but are used only to facilitate the proof of correctness: observe that no auxiliary variable's value is ever assigned to a nonauxiliary variable. To make it easier to see how auxiliary variables would be removed from the program text, we have listed each assignment that refers to such variables on a separate line.

We continue our description of the implementation by considering the shared variables as defined in Figure 2. There is only one nonauxiliary shared variable, namely the $(N+1)$-component composite register Q. Each process i, where $0 \leq i < N$, may read all of the components of Q, and may write components $Q[i]$ and $Q[N]$. Each component of Q consists of two fields, val and tag. The val field is an integer "value," and is used to record the input value of an Increment or Write operation. The tag field consists of two fields, seq and $pnum$. The seq field is a "sequence number" ranging over $0..N+1$, and the $pnum$ field is a "process number" ranging over $0..N-1$.

As mentioned previously, a number of shared auxiliary variables are also included in the implementation. The most important of the auxiliary variables is the history variable H, described above. Another shared history variable S is used to hold INC tuples for "pending" Increment operations. The role of S is explained in detail below. The shared auxiliary boolean variable $Ovlap[i]$ indicates whether an Increment operation of process i is "overlapped" by a Write operation.

The components of Q are used in the following manner. The last component $Q[N]$ is the "base component," and is updated whenever a Write operation is performed by any process. Each other component $Q[i]$, where $0 \leq i < N$, is an "increment component," and is updated when process i performs an Increment operation. Informally, the value of the counter is defined to be that obtained by adding the input value of the most recent Write operation — which is stored in the base component $Q[N]$ — to the sum of the input values of all subsequent Increment operations — which are stored in the increment components $Q[0]$ through $Q[N-1]$. In this context, "recent" and "subsequent" are interpreted with respect to the total order $\prec$, which is defined using the history variable H as explained above. The value of the counter is formally defined by the following expression.

$$Q[N].val + \langle \sum j : 0 \leq j < N \ \wedge \ Q[j].tag = Q[N].tag :: Q[j].val \rangle \tag{1}$$

In this expression, $Q[N].val$ represents the input value of the most recent Write operation, and $\langle \sum j : 0 \leq j <$

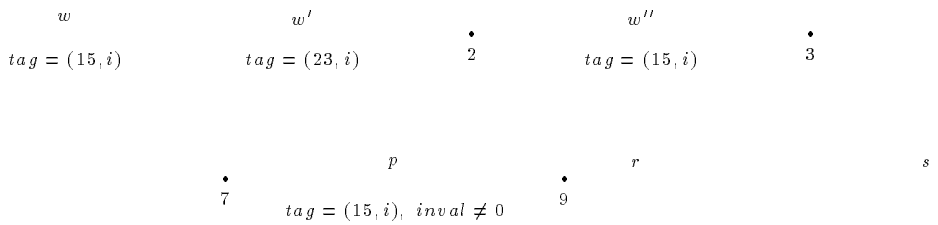

Figure 4: An example history.

$N \wedge Q[j].tag = Q[N].tag :: Q[j].val\rangle$ gives the sum of the input values of all subsequent Increment operations.

With expression (1) in mind, we now consider how Read, Write, and Increment operations are executed in the implementation. A Read operation simply computes the sum defined by (1). Note that, because Q is a composite register, this sum can be computed by reading Q only once.

A Write operation first computes a new tag value, and then writes its input value and tag value to $Q[N]$. The tag value consists of a sequence number and process number. The sequence number is obtained by first reading Q, and then selecting a value differing from any sequence number appearing in the components of Q. Note that, because there are $N+1$ sequence numbers appearing in the $N+1$ components of Q, and because each sequence number ranges over $0..N+1$, such a value exists.

Each Increment operation of process i is executed in two phases. In both phases, Q is read and then $Q[i]$ is written. In each phase, the tag value written to $Q[i]$ is obtained by copying the value read from $Q[N].tag$. If several successive Increment operations of process i obtain the same tag value, then their input values are accumulated in $Q[i].val$ (see the assignments to *sum* in statement 7). It can be shown that the value assigned to $Q[i].val$ by an Increment operation equals the sum of the input values of all Increment operations of process i that are ordered by $\prec$ to occur after the most "recent" Write operation.

To conclude our description of how Increment operations are executed, we informally describe why two phases are necessary. For the sake of explanation, suppose that we modify the implementation of Figure 3 by removing statements 5 and 6 of the Increment procedure. With this modification, each Increment operation consists of only one phase. Now, consider the history depicted in Figure 4. In this figure, operations are denoted by line segments with "time" running from left to right. Certain statement executions (events) within these operations have been denoted as points, each of which is labeled by the corresponding statement number. In this history, w, w', and w'' are three successive Write operations of process i, p is an Increment operation of another process j, and r and s are Read operations of arbitrary processes. The scenario depicted in this history is outlined below.

- w assigns the tag value $(15, i)$ to $Q[N].tag$.
- p reads Q, obtaining $(15, i)$ as its tag value.
- w' assigns the tag value $(23, i)$ to $Q[N].tag$.
- w'' reads Q. Note that, at this point, no tag field in Q equals $(15, i)$. w'' selects the value 15 for its sequence number.

- p writes to $Q[j]$, assigning a nonzero value to $Q[j].val$ and the value $(15, i)$ to $Q[j].tag$.
- w'' writes to $Q[N]$, assigning the value $(15, i)$ to $Q[N].tag$. Note that, at this point, $Q[j].val \neq 0 \ \wedge \ Q[N].tag = Q[j].tag$.

Now, consider the problem of linearizing the operations in Figure 4. Because Read operation r finishes execution before w'' writes to $Q[N]$, by (1), we must linearize r to precede w''. By the given predecence ordering, p precedes r. Hence, by transitivity, we must linearize p to precede w''. Now, because the expression $Q[j].val \neq 0 \ \wedge \ Q[j].tag = Q[N].tag$ holds after w'' finishes execution, when s computes the sum defined by (1), p's input value is included. This implies that we should linearize w'' to precede p, which is a contradiction.

The above problem arises because p assigns a nonzero input value to $Q[j].val$ and an "old" tag value to $Q[j].tag$. To illustrate how this problem is handled in our implementation, consider again the history illustrated in Figure 4, but this time assume that p executes the two-phase Increment procedure of Figure 3. Because w'' chooses $(15, i)$ as its tag value, $Q[j].tag \neq (15, i)$ holds when w'' reads from Q. By the precedence ordering of Figure 4, p's first phase precedes the read by w'' from Q. This implies that p's first phase obtains a tag value that differs from $(15, i)$. Because p obtains different tag values in its two phases, it assigns the value 0 to $Q[j].val$ in its second phase. Thus, p's input value is not included in the sum computed by s. This completes our description of how operations are executed in the implementation.

To formally establish the correctness of the implementation, it suffices to prove that the Ordering and Consistency conditions hold. It is straightforward to show that the Ordering condition holds. Each Read and Write operation appends a unique tuple for itself to H (statements 0 and 3) and such tuples are never removed from H. For Increment operations, the situation is slightly more complicated. When an Increment operation p of process i reads Q in its first phase (statement 5), a unique tuple for that operation is appended to the history variable S. S contains tuples for "pending" Increment operations. The tuple for p is subsequently appended to H either by p itself or by an "overlapping" Write operation. In particular, if a write to $Q[N]$ (statement 3) occurs between the execution of statements 5 and 8 by p, then the first such write removes p's tuple from S and appends it to H. On the other hand, if no write to $Q[N]$ occurs in this interval, then p's tuple is removed from S and appended to H when p executes statement 8. To complete the proof, note that an INC tuple can be removed from H only by the operation to which that tuple corresponds (statement 8), and in this case, the tuple is reappended. This implies that Ordering is satisfied.

We henceforth limit our attention to the Consistency condition. In the remainder of this section, we outline the proof of Consistency. The formal proof appears in the next section. To see that Consistency holds, first observe that the tuples in H may be reordered only when an INC tuple is removed and reappended (see statement 8). However, prior to being removed, such an INC tuple must be part of a sequence of INC tuples followed by a WRITE tuple (see statement 3). By the definition of Val, removing and reappending such an INC does not invalidate the value of any READ tuple.

To complete the proof, we must show that each READ tuple has a valid value when first appended to H. That is, we must prove that $outval = Val(H)$ whenever such a tuple is appended to H by the execution of statement 0 by any process. This can be established by proving the invariance of the following assertion.

$$Val(H) = Q[N].val + \langle \sum i : 0 \leq i < N \ \wedge \ Q[i].tag = Q[N].tag :: Q[i].val \rangle \tag{2}$$

As explained in Section 4, a given assertion is an *invariant* iff it is initially true and is never falsified. Establishing the invariance of (2) is the crux of the proof; see assertion (I23) of Section 4.

By the definition of the initial state, assertion (2) is initially true. Thus, to prove that it is an invariant, we must show that it is not falsified by the execution of any statement by any operation. Showing that (2) is not falsified by any Increment operation is the most difficult part of the proof. The main thrust of this part of the proof is to show that the following two conditions hold for each Increment operation p: first, if the execution of statement 8 by p increments the right-side of (2) by the input value of p, then it also appends an INC tuple for p to H; second, if the execution of statement 8 by p leaves the right-side of (2) unchanged, then either p's input value is 0 (in which case it really does not matter how p is linearized) or H is not modified (in which case p's INC tuple has already been appended to H by an "overlapping" Write operation).

Showing that (2) is not falsified by the statements of Read and Write operations is somewhat simpler. The statements that must be considered are those that may modify H or Q. For Read and Write operations, there are two statements to check, namely 0 and 3.

Statement 0 does not modify Q, but appends a READ tuple to H. However, appending a READ tuple to H does not change the value of $Val(H)$. Therefore, statement 0 does not falsify (2).

Statement 3 assigns "$Q[N].val := inval$" and "$H := H \cdot S \cdot (\text{WRITE}, i, id, inval)$," and thus, by the definition of Val, establishes the assertion $Val(H) = Q[N].val$. Thus, to prove that (2) is not falsified, it suffices to show that the following assertion is established as well.

$$\langle \forall j : 0 \leq j < N :: Q[j].val \neq 0 \;\Rightarrow\; Q[j].tag \neq Q[N].tag \rangle \tag{3}$$

This is the second key invariant of the proof; see assertion (I9) of Section 4. The importance of (3) should not be overlooked. By the semantics of a counter, each Write operation should have the effect of completely overwriting the previous contents of the implemented counter. Assertions (1) and (3) imply that this is indeed the case.

Given the results of [2, 3, 4] and the results of this section, we have the following theorem.

Theorem: Counters can be implemented in a bounded, wait-free manner from atomic registers. □

4 Proof of Consistency

In this section, we formally prove that our counter implementation satisfies the Consistency condition. We find it convenient to express the implementation in the UNITY programming notation of Chandy and Misra [11]. This version of the implementation is shown in Figure 5. In UNITY, the enabling condition of each assignment is explicitly stated; this eliminates the need for introducing notation for referring to the "control" of each process. A brief description of UNITY notation is presented in an appendix. The following notational conventions will be used in the remainder of this section.

Notational Conventions: Unless otherwise specified, we assume in this section that j and k range over $0..N-1$ and that n and v range over the integers. We use α and β to denote sequences of READ, WRITE, and INC tuples, r to denote a single READ tuple, and w to denote a single WRITE tuple. We use (SS.j) to denote

Program *Read_Write_Increment*

declare

$Q,\ H,\ S,\ Ovlap$: as defined in Figure 2,

x: **array** $[0..N-1][0..N]$ **of** $Qtype$, {See Figure 2 for the definition of $Qtype$}

$value,\ id$: **array** $[0..N-1]$ **of integer**,

$frz,\ wr,\ rd,\ inc$: **array** $[0..N-1]$ **of boolean**,

alt: **array** $[0..N-1]$ **of** $0..1$,

$op[i]$: **array** $[0..N-1]$ **of** {READ, WRITE, INC},

$afterw$: **boolean**

always

$\langle\ [\!]\ i : 0 \le i < N :: i.tag = (\langle \min\ seq : seq \ge 0\ \wedge\ \langle \forall j : 0 \le j \le N :: seq \ne x[i,j].tag.seq \rangle :: seq \rangle,\ i)$

$\quad [\!]\ i.val = x[i,N].val + \langle \sum j : 0 \le j < N\ \wedge\ x[i,j].tag = x[i,N].tag :: x[i,j].val \rangle$

$\quad [\!]\ i.tup = (op[i],\ i,\ id[i],\ value[i])\ \rangle$

$[\!]\ Qval = Q[N].val + \langle \sum j : 0 \le j < N\ \wedge\ Q[j].tag = Q[N].tag :: Q[j].val \rangle$

initially

$\langle\ [\!]\ i : 0 \le i < N :: Q[i],\ frz[i],\ rd[i],\ wr[i],\ inc[i],\ alt[i] = (0,\ (0,i)),\ false,\ false,\ false,\ false,\ 0\ \rangle$

$[\!]\ Q[N],\ afterw,\ H,\ S = (init,\ (0,0)),\ false,\ \emptyset,\ \emptyset$

assign

$\langle\ [\!]\ i : 0 \le i < N ::$ {statements of process i}

(SS) $\quad x[i],\ frz[i],\ afterw := Q,\ true,\ false \quad$ **if** $(rd[i]\ \vee\ wr[i]\ \vee\ inc[i])\ \wedge\ \neg frz[i]$

$\quad \|\ H := H \cdot (\text{READ},\ i,\ id[i],\ Qval) \quad$ **if** $rd[i]\ \wedge\ \neg frz[i]$

$\quad \|\ S := S \cdot i.tup \quad$ **if** $inc[i]\ \wedge\ \neg frz[i]\ \wedge\ alt[i] = 0$

$\quad \|\ Ovlap[i] := false \quad$ **if** $inc[i]\ \wedge\ \neg frz[i]\ \wedge\ alt[i] = 1$

(R) $\quad [\!]\ value[i],\ rd[i],\ frz[i],\ afterw := i.val,\ false,\ false,\ false \quad$ **if** $rd[i]\ \wedge\ frz[i]$

(W) $\quad [\!]\ Q[N],\ wr[i],\ frz[i],\ afterw,\ H,\ S := (value[i],\ i.tag),\ false,\ false,\ true,\ H \cdot S \cdot i.tup,\ \emptyset \quad$ **if** $wr[i]\ \wedge\ frz[i]$

$\quad \|\ \langle \|\ j : 0 \le j < N :: Ovlap[j] := true \quad$ **if** $wr[i]\ \wedge\ frz[i]\ \rangle$

(I) $\quad [\!]\ Q[i] := (0,\ x[i,N].tag) \quad$ **if** $inc[i]\ \wedge\ frz[i]\ \wedge\ x[i,i].tag \ne x[i,N].tag\ \sim$

$\qquad (alt[i] \times value[i] + x[i,i].val,\ x[i,N].tag) \quad$ **if** $inc[i]\ \wedge\ frz[i]\ \wedge\ x[i,i].tag = x[i,N].tag$

$\quad \|\ inc[i],\ alt[i],\ frz[i],\ afterw := (alt[i] = 0),\ 1 - alt[i],\ false,\ false \quad$ **if** $inc[i]\ \wedge\ frz[i]$

$\quad \|\ H,\ S := (H - i.tup) \cdot i.tup,\ S - i.tup$

$\qquad$ **if** $inc[i]\ \wedge\ frz[i]\ \wedge\ alt[i] = 1\ \wedge\ (i.tup \notin H\ \vee\ (\neg Ovlap[i]\ \wedge\ x[i,i].tag = x[i,N].tag))$

(RE) $\quad [\!]\ rd[i],\ id[i],\ op[i] := true,\ id[i]+1,\ \text{READ} \quad$ **if** $\neg rd[i]\ \wedge\ \neg wr[i]\ \wedge\ \neg inc[i]\ \wedge\ \neg frz[i]$

(WE) $\quad [\!]\ wr[i],\ value[i],\ id[i],\ op[i] := true,\ \text{value to write},\ id[i]+1,\ \text{WRITE} \quad$ **if** $\neg rd[i]\ \wedge\ \neg wr[i]\ \wedge\ \neg inc[i]\ \wedge\ \neg frz[i]$

(IE) $\quad [\!]\ inc[i],\ value[i],\ id[i],\ op[i] := true,\ \text{value to add},\ id[i]+1,\ \text{INC} \quad$ **if** $\neg rd[i]\ \wedge\ \neg wr[i]\ \wedge\ \neg inc[i]\ \wedge\ \neg frz[i]$

$\rangle$

end {*Read_Write_Increment*}

Figure 5: Counter implementation in UNITY.

statement (SS) in Figure 5 with variable i instantiated by the value j. In other words, (SS.j) corresponds to statement (SS) of process j. Other statements of process j are denoted similarly. The notation $E^{x_1,\ldots,x_n}_{e_1,\ldots,e_n}$ is used to denote predicate E with each free occurrence of variable x_i replaced by the corresponding expression e_i. The following is a list of symbols we will use ordered by increasing binding power: $\equiv$, $\Rightarrow$, $\vee$, $\wedge$, $(=, \neq, >, <, \in, \notin)$, $+$, $\times$, $\cdot$, $\neg$, . . The symbols enclosed in parentheses have the same priority. □

The equivalence of the original program in Figure 3 and the UNITY program in Figure 5 can easily be established by comparison. In the UNITY program, statement (SS.j) is executed when process j takes a "snapshot" of Q. Statement (R.j) (respectively, (W.j) or (I.j)) is executed when a Read (respectively, Write or Increment) operation is performed by process j. Statement (RE.j) (respectively, (WE.j) or (IE.j)) is executed in order to "enable" a Read (respectively, Write or Increment) operation of process j. A Read operation of process j is performed by executing the sequence of assignments (RE.j); (SS.j); (R.j). A Write operation of process j is performed by executing the sequence of assignments (WE.j); (SS.j); (W.j). An Increment operation of process j is performed by executing the sequence of assignments (IE.j); (SS.j); (I.j); (SS.j); (I.j). Variables $frz[j]$ and $alt[j]$ are used to enforce the sequential execution of statements of process j. Variable $frz[j]$ is set to true when process j executes statement (SS.j) to take a snapshot and is set to false when process j executes statements (R.j), (W.j), and (I.j); this has the effect of "freezing" process j from taking subsequent snapshots until the values read during this snapshot are subsequently used in one of the latter statements. Variable $alt[j]$ is used to "alternate" between the two phases of an Increment operation of process j. Variable $rd[j]$ (respectively, $wr[j]$ or $inc[j]$) is set to true when a Read (respectively, Write or Increment) operation is performed by process j.

Several variables are introduced in the **always** section as a shorthand for various expressions. Variable $i.tag$ corresponds to process i's tag value, and is assumed to be of type *Tagtype* (see Figure 2). Variable $i.tup$ corresponds to process i's current tuple, and is assumed to be of type *Htype*. Variable $i.val$ corresponds to the value of the implemented counter as determined by process i's last read from Q, and variable $Qval$ gives the value of the counter as defined by the components of Q; these variables are assumed to range over the integers.

Before giving the proof of Consistency, we first introduce some terminology. Let A and B be predicates over program variables and let s be a statement. Then, $\{A\}\ s\ \{B\}$ is true iff the following condition holds: if A holds immediately before the execution of s, then B holds immediately after the execution of s. For a given program, predicate A is *stable* iff $\{A\}\ s\ \{A\}$ holds for each statement s of that program. For a given program, a predicate is *invariant* iff that predicate is stable and initially true. To prove that an assertion is invariant, it suffices to prove that it is initially true and is not falsified by the effective execution of any statement (as explained in the appendix, only *effective* statement executions change the program state).

In order to show that Consistency holds, we first present a number of invariants. The following six simple invariants, which are stated without proof, follow directly from the program text.

invariant $\quad (\neg rd[j] \wedge \neg wr[j]) \vee (\neg rd[j] \wedge \neg inc[j]) \vee (\neg wr[j] \wedge \neg inc[j])$ (I0)

invariant $\quad \langle \forall k : 0 \leq k \leq N :: x[j,k].tag \neq j.tag \rangle$ (I1)

invariant $k \neq j \;\Rightarrow\; k.tag \neq j.tag$ (I2)

invariant $\langle \forall z : z \in S :: z.op = \text{INC} \rangle$ (I3)

invariant $\langle \forall z : z \in H \cdot S \;\wedge\; z.pnum = j :: z.id \leq id[j] \rangle$ (I4)

invariant $(wr[j] \;\Rightarrow\; op[j] = \text{WRITE}) \;\wedge\; (rd[j] \;\Rightarrow\; op[j] = \text{READ}) \;\wedge\; (inc[j] \;\Rightarrow\; op[j] = \text{INC})$ (I5)

To see that (I0) and (I5) hold, examine statements (RE.j), (WE.j), and (IE.j): these are the only statements that may establish $rd[j]$, $wr[j]$, or $inc[j]$ or that may modify $op[j]$. (I1) and (I2) follow by the definition of $j.tag$ and $k.tag$. (I3) is an invariant because only INC tuples are appended to S. (I4) holds because $id[j]$ is never decremented — note that, when a tuple with process number j is first appended to H or S, its id field equals $id[j]$. For each of the remaining invariants, a formal correctness proof is given.

invariant $frz[j] \;\Rightarrow\; Q[N].tag \neq j.tag \;\wedge\; Q[j] = x[j,j]$ (I6)

Proof: Initially $frz[j]$ is false, and hence (I6) is true. To prove that (I6) is stable, it suffices to consider only those statements that may establish $frz[j]$ or falsify $Q[N].tag \neq j.tag \;\wedge\; Q[j] = x[j,j]$. The statements to check are (SS.j), (W.j), (I.j), and (W.k), where $k \neq j$. By the axiom of assignment, each effective execution of (W.j) and (I.j) falsifies $frz[j]$, and hence these statements do not falsify (I6). For statement (SS.j), the following assertions hold.

$\{(rd[j] \;\vee\; wr[j] \;\vee\; inc[j]) \;\wedge\; \neg frz[j]\}$ (SS.j) $\{Q[N].tag = x[j,N].tag \;\wedge\; Q[j] = x[j,j]\}$
, by the axiom of assignment.

$\{(rd[j] \;\vee\; wr[j] \;\vee\; inc[j]) \;\wedge\; \neg frz[j]\}$ (SS.j) $\{Q[N].tag \neq j.tag \;\wedge\; Q[j] = x[j,j]\}$
, by previous assertion and (I1).

$\{\neg((rd[j] \;\vee\; wr[j] \;\vee\; inc[j]) \;\wedge\; \neg frz[j]) \;\wedge\; \text{I6}\}$ (SS.j) $\{\text{I6}\}$, (SS.j) not effective with this precondition.

The last two of the above assertions imply that $\{\text{I6}\}$ (SS.j) $\{\text{I6}\}$ holds.

Next, consider statement (W.k), where $k \neq j$. For this statement, the following assertions hold.

$\{wr[k] \;\wedge\; frz[k] \;\wedge\; \text{I6}^{Q[N].tag}_{k.tag}\}$ (W.k) $\{\text{I6}\}$, by the axiom of assignment.

$\text{I6} \;\Rightarrow\; \text{I6}^{Q[N].tag}_{k.tag}$, by (I2)

$\{wr[k] \;\wedge\; frz[k] \;\wedge\; \text{I6}\}$ (W.k) $\{\text{I6}\}$, by previous two assertions.

$\{(\neg wr[k] \;\vee\; \neg frz[k]) \;\wedge\; \text{I6}\}$ (W.k) $\{\text{I6}\}$, (W.k) not effective with this precondition.

From the last two of these assertions, it follows that $\{\text{I6}\}$ (W.k) $\{\text{I6}\}$ holds. □

invariant $frz[j] \;\wedge\; frz[k] \;\Rightarrow\; x[k,k].tag \neq j.tag \vee\; x[k,N].tag \neq j.tag$ (I7)

Proof: Initially $frz[j]$ and $frz[k]$ are both false, and hence (I7) is true. To prove that (I7) is stable, we must consider only those statements that establish $frz[j]$ or $frz[k]$ or falsify the right-side of the implication. The only statements to consider are (SS.j) and (SS.k).

First, consider statement (SS.j). To see that this statement does not falsify (I7), consider the following assertions.

$\{(rd[j] \vee wr[j] \vee inc[j]) \wedge \neg frz[j]\}$ (SS.j) $\{\neg frz[k] \vee (frz[k] \wedge Q[k].tag = x[j,k].tag)\}$
, by the axiom of assignment

$frz[k] \wedge Q[k].tag = x[j,k].tag \Rightarrow x[k,k].tag \neq j.tag$, by (I6), $frz[k]$ implies $Q[k].tag = x[k,k].tag$; by (I1), $x[j,k].tag \neq j.tag$.

$\{(rd[j] \vee wr[j] \vee inc[j]) \wedge \neg frz[j]\}$ (SS.j) $\{\neg frz[k] \vee x[k,k].tag \neq j.tag\}$
, by previous two assertions

$\{\neg((rd[j] \vee wr[j] \vee inc[j]) \wedge \neg frz[j]) \wedge$ I7$\}$ (SS.j) $\{$I7$\}$, (SS.j) not effective with this precondition.

The last two of these assertions imply that $\{$I7$\}$ (SS.j) $\{$I7$\}$ holds.

Finally, consider the statement (SS.k), and assume that $k \neq j$. In this case, we can establish our proof obligation as follows.

$\text{I7}^{frz[k],\ x[k]}_{true,\ Q} = (frz[j] \Rightarrow Q[k].tag \neq j.tag \vee Q[N].tag \neq j.tag)$, by the definition of (I7).

$= true$, by the invariance of (I6).

$\{(rd[k] \vee wr[k] \vee inc[k]) \wedge \neg frz[k] \wedge \text{I7}^{frz[k],\ x[k]}_{true,\ Q}\}$ (SS.k) $\{$I7$\}$
, by the axiom of assignment.

$\{(rd[k] \vee wr[k] \vee inc[k]) \wedge \neg frz[k]\}$ (SS.k) $\{$I7$\}$, by previous assertion and above derivation.

$\{\neg((rd[k] \vee wr[k] \vee inc[k]) \wedge \neg frz[k]) \wedge$ I7$\}$ (SS.k) $\{$I7$\}$, (SS.k) not effective with this precondition.

The last two of the above assertions imply that $\{$I7$\}$ (SS.k) $\{$I7$\}$ holds. □

invariant $frz[j] \wedge Q[k].val \neq 0 \Rightarrow Q[k].tag \neq j.tag$ (I8)

Proof: Initially $frz[j]$ is false, and hence (I8) is true. To prove that (I8) is stable, it suffices to consider only those statements that may establish $frz[j]$, $Q[k].val \neq 0$, or $Q[k].tag = j.tag$. The statements to check are (SS.j) and (I.k). Because each effective execution of (I.j) establishes $\neg frz[j]$, we may assume in the latter case that $k \neq j$. To see that $\{$I8$\}$ (SS.j) $\{$I8$\}$ holds, observe the following.

$\{(rd[j] \vee wr[j] \vee inc[j]) \wedge \neg frz[j]\}$ (SS.j) $\{x[j,k].tag = Q[k].tag\}$
, by the axiom of assignment.

$\{(rd[j] \vee wr[j] \vee inc[j]) \wedge \neg frz[j]\}$ (SS.j) $\{Q[k].tag \neq j.tag\}$, by (I1) and previous assertion.

$\{\neg((rd[j] \ \vee \ wr[j] \ \vee \ inc[j]) \ \wedge \ \neg frz[j]) \ \wedge \ \text{I8}\}$ (SS.j) {I8} , (SS.j) not effective with this precondition.

The last two of these assertions imply that {I8} (SS.j) {I8} holds. Next, consider statement (I.k), where $k \neq j$. For this statement, we have the following.

$\text{I8}^{Q[k].val, \ Q[k].tag}_{0, \ x[k,N].tag} = (frz[j] \ \wedge \ 0 \neq 0 \ \Rightarrow \ x[k,N].tag \neq j.tag)$, by the definition of (I8).

$= true$, predicate calculus.

$\{inc[k] \ \wedge \ frz[k] \ \wedge \ x[k,k].tag \neq x[k,N].tag \ \wedge \ \text{I8}^{Q[k].val, \ Q[k].tag}_{0, \ x[k,N].tag}\}$ (I.k) {I8}
, by the axiom of assignment.

$\{inc[k] \ \wedge \ frz[k] \ \wedge \ x[k,k].tag \neq x[k,N].tag\}$ (I.k) {I8} , by previous assertion and above derivation. (A0)

$frz[k] \ \wedge \ x[k,k].tag = x[k,N].tag \ \wedge \ \text{I8}^{Q[k].val, \ Q[k].tag}_{alt[k] \times value[k] + x[k,k].tag, \ x[k,N].tag}$

$= frz[k] \ \wedge \ x[k,k].tag = x[k,N].tag \ \wedge$
$((frz[j] \ \wedge \ (alt[k] \times value[k] + x[k,k].val \neq 0)) \ \Rightarrow \ x[k,N].tag \neq j.tag)$
, by the definition of (I8).

$= frz[k] \ \wedge \ x[k,k].tag = x[k,N].tag$, by the invariance of (I7).

$\{inc[k] \ \wedge \ frz[k] \ \wedge \ x[k,k].tag = x[k,N].tag \ \wedge \ \text{I8}^{Q[k].val, \ Q[k].tag}_{alt[k] \times value[k] + x[k,k].val, \ x[k,N].tag}\}$ (I.k) {I8}
, by the axiom of assignment.

$\{inc[k] \ \wedge \ frz[k] \ \wedge \ x[k,k].tag = x[k,N].tag\}$ (I.k) {I8} , by previous assertion and above derivation. (A1)

$\{inc[k] \ \wedge \ frz[k]\}$ (I.k) {I8} , by (A0) and (A1).

$\{(\neg inc[k] \ \vee \ \neg frz[k]) \ \wedge \ \text{I8}\}$ (I.k) {I8} , (I.k) not effective with this precondition.

The last two of these assertions imply that {I8} (I.k) {I8} holds. □

By the definition of $Qval$, the following invariant shows that each Write operation has the effect of completely overwriting the previous contents of the implemented counter.

invariant $afterw \ \Rightarrow \ \langle \forall k :: Q[k].val \neq 0 \ \Rightarrow \ Q[k].tag \neq Q[N].tag \rangle$ (I9)

Proof: Initially *afterw* is false, and hence (I9) is true. To prove that (I9) is stable, it suffices to consider those statements they may establish *afterw* or falsify the right-side of the implication. The statements to consider are

(W.j) and (I.j). Each effective execution of (I.j) establishes $\neg afterw$. Therefore, this statement does not falsify (I9). For statement (W.j), the following assertions hold.

$frz[j] \Rightarrow \langle \forall k :: Q[k].val = 0 \ \vee \ Q[k].tag \neq j.tag \rangle$, by (I8).

$\Rightarrow (true \ \Rightarrow \ \langle \forall k :: Q[k].val \neq 0 \ \Rightarrow \ Q[k].tag \neq j.tag \rangle)$, predicate calculus.

$= \ \mathrm{I9}^{Q[N].tag, \ afterw}_{j.tag, \ true}$, by the definition of (I9).

$\{wr[j] \ \wedge \ frz[j] \ \wedge \ \mathrm{I9}^{Q[N].tag, \ afterw}_{j.tag, \ true}\}$ (W.j) {I9} , by the axiom of assignment.

$\{wr[j] \ \wedge \ frz[j]\}$ (W.j) {I9} , by previous assertion and above derivation.

$\{(\neg wr[j] \ \vee \ \neg frz[j]) \ \wedge \ \mathrm{I9}\}$ (W.j) {I9} , (W.j) not effective with this precondition.

The last two of these assertions imply that {I9} (W.j) {I9} holds. □

invariant $\neg inc[j] \ \Rightarrow \ alt[j] = 0$ (I10)

Proof: Initially $alt[j] = 0$, and hence (I10) holds. To prove that (I10) is stable, it suffices to consider only those statements that modify $inc[j]$ and $alt[j]$. The statements to check are (I.j) and (IE.j). Each effective execution of (IE.j) establishes $inc[j]$; hence, this statement does not falsify (I10). For statement (I.j), we have the following.

$\{(\neg inc[j] \ \vee \ \neg frz[j]) \ \wedge \ \mathrm{I10}\}$ (I.j) {I10} , (I.j) not effective with this precondition.

$\{inc[j] \ \wedge \ frz[j] \ \wedge \ alt[j] = 0\}$ (I.j) $\{inc[j]\}$, by the axiom of assignment.

$\{inc[j] \ \wedge \ frz[j] \ \wedge \ alt[j] = 1\}$ (I.j) $\{alt[j] = 0\}$, by the axiom of assignment.

These three assertions imply that {I10} (I.j) {I10} holds. □

invariant $inc[j] \ \wedge \ (frz[j] \ \vee \ alt[j] = 1) \ \Rightarrow \ j.tup \in H \ \vee \ j.tup \in S$ (I11)

Proof: Initially $inc[j]$ is false, and hence (I11) holds. To prove that (I11) is stable, we must check those statements that may establish $inc[j] \ \wedge \ (frz[j] \ \vee \ alt[j] = 1)$ or falsify $j.tup \in H \ \vee \ j.tup \in S$. The statements to check are (SS.j), (W.j), (I.j), (RE.j), (WE.j), (IE.j), and (W.k), where $k \neq j$. By (I0), $\neg inc[j]$ holds prior to each effective execution of (W.j), and by the program text, $\neg inc[j]$ holds prior to each effective execution of (RE.j) and (WE.j). Thus, by the axiom of assignment, $\neg inc[j]$ holds after each effective execution of these statements, and hence (I11) is not falsified. By the axiom of assignment, statement (W.k) cannot falsify $j.tup \in H \ \vee \ j.tup \in S$. Thus, this statement also does not falsify (I11). Next, consider statement (SS.j). For this statement, the following assertions hold.

$\{\neg inc[j]\}$ (SS.j) $\{\neg inc[j]\}$, by the axiom of assignment.

$\{inc[j] \wedge frz[j] \wedge \text{I11}\}$ (SS.j) $\{\text{I11}\}$, (SS.j) not effective with this precondition.

$\{inc[j] \wedge \neg frz[j] \wedge alt[j] = 0\}$ (SS.j) $\{j.tup \in S\}$, by the axiom of assignment.

$\{inc[j] \wedge \neg frz[j] \wedge alt[j] = 1 \wedge \text{I11}\}$ (SS.j) $\{\text{I11}\}$, precondition implies $j.tup \in H \vee j.tup \in S$; postcondition follows by the axiom of assignment.

These four assertions imply that $\{\text{I11}\}$ (SS.j) $\{\text{I11}\}$ holds.

Next, we establish that $\{\text{I11}\}$ (I.j) $\{\text{I11}\}$ holds. For statement (I.j), the following assertions hold.

$\{(\neg inc[j] \vee \neg frz[j]) \wedge \text{I11}\}$ (I.j) $\{\text{I11}\}$, (I.j) not effective with this precondition.

$\{inc[j] \wedge frz[j] \wedge alt[j] = 0 \wedge \text{I11}\}$ (I.j) $\{\text{I11}\}$, precondition implies $j.tup \in H \vee j.tup \in S$; postcondition follows by the axiom of assignment.

$\{inc[j] \wedge frz[j] \wedge alt[j] = 1\}$ (I.j) $\{j.tup \in H\}$, by the axiom of assignment.

These three assertions imply that $\{\text{I11}\}$ (I.j) $\{\text{I11}\}$ holds.

Finally, for statement (IE.j), we have the following.

$\{(rd[j] \vee wr[j] \vee inc[j] \vee frz[j]) \wedge \text{I11}\}$ (IE.j) $\{\text{I11}\}$, (IE.j) not effective with this precondition.

$\{\neg rd[j] \wedge \neg wr[j] \wedge \neg inc[j] \wedge \neg frz[j]\}$ (IE.j) $\{\neg frz[j] \wedge alt[j] = 0\}$, by (I10), precondition implies $alt[j] = 0$; postcondition follows by the axiom of assignment.

These two assertions imply that $\{\text{I11}\}$ (IE.j) $\{\text{I11}\}$ holds. □

invariant $inc[j] \wedge \neg frz[j] \wedge alt[j] = 1 \Rightarrow x[j, N].tag = Q[j].tag$ (I12)

Proof: (I12) holds initially because $inc[j]$ is initially false. To prove that (I12) is stable, we must consider those statements that may establish $inc[j] \wedge \neg frz[j] \wedge alt[j] = 1$ or falsify $x[j, N].tag = Q[j].tag$. The statements to consider are (SS.j), (R.j), (W.j), (I.j), and (IE.j). By (I0), $\neg inc[j]$ holds after each effective execution of (R.j) and (W.j). By the axiom of assignment, $x[j, N].tag = Q[j].tag$ holds after each effective execution of (I.j). By (I10), $alt[j] = 0$ holds after each effective execution of statement (IE.j). By the axiom of assignment, each effective execution of (SS.j) establishes $frz[j]$. It follows, then, that (I12) is stable. □

invariant $inc[j] \wedge (frz[j] \vee alt[j] = 1) \wedge j.tup \notin H \Rightarrow x[j, N].tag = Q[N].tag$ (I13)

Proof: Initially $inc[j]$ is false, and hence (I13) is true. To prove that (I13) is stable it suffices to consider those

statements that may establish $inc[j] \wedge (frz[j] \vee alt[j] = 1) \wedge j.tup \notin H$ or falsify $x[j,N].tag = Q[N].tag$. The statements to check are (SS.j), (R.j), (W.j), (I.j), (RE.j), (WE.j), (IE.j), and (W.k), where $k \neq j$. By the axiom of assignment, each effective execution of (SS.j) establishes $x[j,N].tag = Q[N].tag$. By (I0), $\neg inc[j]$ holds prior to each effective execution of (R.j) or (W.j), and by the program text, $\neg inc[j]$ holds prior to each effective execution of (RE.j) or (WE.j); thus, by the axiom of assignment, $\neg inc[j]$ holds after each effective execution of these statements. By (I10), $\neg frz[j] \wedge alt[j] = 0$ holds after each effective execution of (IE.j). It follows, then, that these statements do not falsify (I13). The remaining statements to consider are (I.j) and (W.k). For statement (I.j), the following assertions hold.

$\{inc[j] \wedge frz[j] \wedge alt[j] = 0 \wedge \text{I13}^{inc[j],\ frz[j],\ alt[j]}_{true,\ false,\ 1}\}$ (I.j) {I13}
, by the axiom of assignment.

$inc[j] \wedge frz[j] \wedge \text{I13} \Rightarrow \text{I13}^{inc[j],\ frz[j],\ alt[j]}_{true,\ false,\ 1}$, by the definition of (I13).

$\{inc[j] \wedge frz[j] \wedge alt[j] = 0 \wedge \text{I13}\}$ (I.j) {I13} , by previous two assertions.

$\{inc[j] \wedge frz[j] \wedge alt[j] = 1\}$ (I.j) $\{\neg inc[j]\}$, by the axiom of assignment.

$\{(\neg inc[j] \vee \neg frz[j]) \wedge \text{I13}\}$ (I.j) {I13} , (I.j) not effective with this precondition.

The last three of these assertions imply that {I13} (I.j) {I13} holds.

Finally, consider statement (W.k), $k \neq j$. For this statement, the following assertions hold.

$\text{I13}^{Q[N].tag,\ H}_{k.tag,\ H \cdot S \cdot k.tup} = (inc[j] \wedge (frz[j] \vee alt[j] = 1) \wedge j.tup \notin H \cdot S \cdot k.tup \Rightarrow x[j,N].tag = k.tag)$
, by the definition of (I13).

$= true$, by (I11), left-side of previous implication is false.

$\{wr[k] \wedge frz[k] \wedge \text{I13}^{Q[N].tag,\ H}_{k.tag,\ H \cdot S \cdot k.tup}\}$ (W.k) {I13} , by the axiom of assignment.

$\{wr[k] \wedge frz[k]\}$ (W.k) {I13} , by previous assertion and above derivation.

$\{(\neg wr[k] \vee \neg frz[k]) \wedge \text{I13}\}$ (W.k) {I13} , (W.k) not effective with this precondition.

The last two of these assertions imply that {I13} (W.k) {I13} holds. □

invariant $inc[j] \wedge frz[j] \wedge alt[j] = 1 \wedge j.tup \notin H \Rightarrow x[j,j].tag = x[j,N].tag$ (I14)

Proof: Initially $inc[j]$ is false, and hence (I14) is true. To prove that (I14) is stable, it suffices to consider those statements that may establish $inc[j] \wedge frz[j] \wedge alt[j] = 1 \wedge j.tup \notin H$ or falsify $x[j,j].tag = x[j,N].tag$. The statements to consider are (SS.j), (R.j), (W.j), (I.j), (RE.j), (WE.j), (IE.j), and (W.k), where $k \neq j$. By the axiom of assignment, $\neg frz[j]$ holds after each effective execution of (R.j), (W.j), (I.j), (RE.j), (WE.j),

or (IE.j). Hence, these statements do not falsify (I14). The remaining statements to consider are (SS.j) and (W.k). For statement (SS.j), the following assertions hold.

$\{\neg inc[j]\}$ (SS.j) $\{\neg inc[j]\}$, by the axiom of assignment.

$\{inc[j] \wedge frz[j] \wedge \text{I14}\}$ (SS.j) {I14} , (SS.j) not effective with this precondition.

$\{inc[j] \wedge \neg frz[j] \wedge alt[j] = 0\}$ (SS.j) $\{alt[j] = 0\}$, by the axiom of assignment.

$\{inc[j] \wedge \neg frz[j] \wedge alt[j] = 1 \wedge j.tup \in H\}$ (SS.j) $\{j.tup \in H\}$
, by the axiom of assignment.

$\{inc[j] \wedge \neg frz[j] \wedge alt[j] = 1 \wedge j.tup \notin H\}$ (SS.j) $\{x[j,j].tag = x[j,N].tag\}$
, by (I12) and (I13), precondition implies $Q[j].tag = Q[N].tag$; postcondition follows by the axiom of assignment.

These assertions imply that {I14} (SS.j) {I14} holds.

Finally, consider statement (W.k), $k \neq j$. For this statement, the following assertions hold.

$\text{I14}^H_{H \cdot S \cdot k.tup} = (inc[j] \wedge frz[j] \wedge alt[j] = 1 \wedge j.tup \notin H \cdot S \cdot k.tup \Rightarrow x[j,j].tag = x[j,N].tag)$
, by the definition of (I14).

$= true$, by (I11), left-side of previous implication is false.

$\{wr[k] \wedge frz[k] \wedge \text{I14}^H_{H \cdot S \cdot k.tup}\}$ (W.k) {I14} , by the axiom of assignment.

$\{wr[k] \wedge frz[k]\}$ (W.k) {I14} , by previous assertion and above derivation.

$\{(\neg wr[k] \vee \neg frz[k]) \wedge \text{I14}\}$ (W.k) {I14} , (W.k) not effective with this precondition.

The last two of these assertion imply that {I14} (W.k) {I14} holds. □

invariant $inc[j] \wedge frz[j] \wedge alt[j] = 1 \wedge \neg Ovlap[j] \Rightarrow x[j,N].tag = Q[N].tag$ (I15)

Proof: Initially $inc[j]$ is false, and hence (I15) is true. To prove that (I15) is stable, it suffices to consider those statements that may establish $inc[j]$, $frz[j]$, $alt[j] = 1$, $\neg Ovlap[j]$, or $x[j,N].tag \neq Q[N].tag$. The statements to consider are (SS.j), (R.j), (I.j), (IE.j), and (W.k), where $0 \leq k < N$. By the axiom of assignment, $x[j,N].tag = Q[N].tag$ holds after each effective execution of (SS.j), $\neg frz[j]$ holds after each effective execution of (R.j), (I.j), and (IE.j), and $Ovlap[j]$ holds after each effective execution of (W.k). Hence, (I15) is stable. □

invariant $inc[j] \wedge frz[j] \wedge alt[j] = 1 \wedge j.tup \in H \wedge Ovlap[j] \Rightarrow x[j,j].tag \neq x[j,N].tag \vee x[j,N].tag \neq Q[N].tag$ (I16)

Proof: Initially $inc[j]$ is false, and hence (I16) is true. To prove that (I16) is stable, it suffices to consider those statements that may establish $inc[j]$, $frz[j]$, $alt[j] = 1$, $j.tup \in H$, $Ovlap[j]$, $x[j,j].tag = x[j,N].tag$, or $x[j,N].tag = Q[N].tag$. The statements to consider are (SS.j), (R.j), (W.j), (I.j), (RE.j), (WE.j), (IE.j), and (W.k), where $k \neq j$. By the axiom of assignment, $\neg frz[j]$ holds after each effective execution of (R.j), (W.j), (I.j), (RE.j), (WE.j), and (IE.j). Hence, these statements do not falsify (I16). The remaining statements to consider are (SS.j) and (W.k). For statement (SS.j), the following assertions hold.

$\{\neg inc[j]\}$ (SS.j) $\{\neg inc[j]\}$, by the axiom of assignment.

$\{inc[j] \wedge frz[j] \wedge \text{I16}\}$ (SS.j) {I16} , (SS.j) not effective with this precondition.

$\{inc[j] \wedge \neg frz[j] \wedge alt[j] = 0\}$ (SS.j) $\{alt[j] = 0\}$, by the axiom of assignment.

$\{inc[j] \wedge \neg frz[j] \wedge alt[j] = 1\}$ (SS.j) $\{\neg Ovlap[j]\}$, by the axiom of assignment.

These four assertions imply that {I16} (SS.j) {I16} holds.

Finally, consider statement (W.k), $k \neq j$. For this statement, the following assertions hold.

$\{wr[k] \wedge frz[k] \wedge frz[j]\}$ (W.k) $\{x[j,j].tag \neq Q[N].tag \vee x[j,N].tag \neq Q[N].tag\}$
, by (I7), precondition implies
$x[j,j].tag \neq k.tag \vee x[j,N].tag \neq k.tag$;
postcondition follows by the axiom
of assignment.

$(x[j,j].tag \neq Q[N].tag \vee x[j,N].tag \neq Q[N].tag) = (x[j,j].tag \neq x[j,N].tag \vee x[j,N].tag \neq Q[N].tag)$
, predicate calculus.

$\{wr[k] \wedge frz[k] \wedge frz[j]\}$ (W.k) $\{x[j,j].tag \neq x[j,N].tag \vee x[j,N].tag \neq Q[N].tag\}$
, by previous two assertions.

$\{wr[k] \wedge frz[k] \wedge \neg frz[j]\}$ (W.k) $\{\neg frz[j]\}$, by the axiom of assignment.

$\{(\neg wr[k] \vee \neg frz[k]) \wedge \text{I16}\}$ (W.k) {I16} , (W.k) not effective with this precondition.

The last three of these assertions imply that {I16} (W.k) {I16} holds. □

invariant $inc[j] \wedge frz[j] \wedge x[j,j].tag \neq x[j,N].tag \Rightarrow Q[j].val = 0 \vee Q[j].tag \neq Q[N].tag$ (I17)

Proof: Initially $inc[j]$ is false and hence (I17) is true. To prove that (I17) is stable, it suffices to consider those statements that may establish $inc[j] \wedge frz[j] \wedge x[j,j].tag \neq x[j,N].tag$ or falsify $Q[j].val = 0 \vee Q[j].tag \neq Q[N].tag$. The statements to check are (SS.j), (I.j), (IE.j), and (W.k), where $0 \leq k < N$. By the axiom of assignment, $\neg frz[j]$ holds after each effective execution of (I.j) and (IE.j), and hence, these statements do not

falsify (I17). The remaining statements to consider are (SS.j) and (W.k). For statement (SS.j), we have the following.

$\{\neg inc[j]\}$ (SS.j) $\{\neg inc[j]\}$, by the axiom of assignment.

$\{inc[j] \wedge frz[j] \wedge \text{I17}\}$ (SS.j) {I17} , (SS.j) not effective with this precondition.

$\{inc[j] \wedge \neg frz[j]\}$ (SS.j) $\{inc[j] \wedge frz[j] \wedge (x[j,j].tag \neq x[j,N].tag \Rightarrow Q[j].tag \neq Q[N].tag)\}$, by the axiom of assignment.

These three assertions imply that {I17} (SS.j) {I17} holds.

Finally, consider statement (W.k). For this statement, the following assertions hold.

$\{wr[k] \wedge frz[k]\}$ (W.k) $\{afterw\}$, by the axiom of assignment.

$\{wr[k] \wedge frz[k]\}$ (W.k) $\{Q[j].val = 0 \vee Q[j].tag \neq Q[N].tag\}$, by (I9) and previous assertion.

$\{(\neg wr[k] \vee \neg frz[k]) \wedge \text{I17}\}$ (W.k) {I17} , (W.k) not effective with this precondition.

The last two of these assertions imply that {I17} (W.k) {I17} holds. □

invariant $inc[j] \wedge \neg frz[j] \wedge alt[j] = 0 \Rightarrow \langle \forall z : z \in H \cdot S \wedge z.pnum = j :: z.id < id[j] \rangle$ (I18)

Proof: Initially $inc[j]$ is false, and hence (I18) is true. (I18) could potentially be falsified only by those statements that may establish $inc[j] \wedge \neg frz[j] \wedge alt[j] = 0$, or modify $id[j]$, or append a tuple with process number j to H or S. The statements to consider are (SS.j), (R.j), (W.j), (I.j), (RE.j), (WE.j), (IE.j), and (W.k), where $k \neq j$. By the axiom of assignment, $frz[j]$ holds after each effective execution of (SS.j), and $\neg inc[j]$ holds after each effective execution of (RE.j) or (WE.j). Also, by (I0), $\neg inc[j]$ holds following each effective execution of (R.j) or (W.j). It follows, then, that these statements do not falsify (I18).

The remaining statements to consider are (I.j), (IE.j), and (W.k). For statement (I.j), we have the following three assertions.

$\{(\neg inc[j] \vee \neg frz[j]) \wedge \text{I18}\}$ (I.j) {I18} , (I.j) not effective with this precondition.

$\{inc[j] \wedge frz[j] \wedge alt[j] = 0\}$ (I.j) $\{alt[j] = 1\}$, by the axiom of assignment.

$\{inc[j] \wedge frz[j] \wedge alt[j] = 1\}$ (I.j) $\{\neg inc[j]\}$, by the axiom of assignment.

These three assertions imply that {I18} (I.j) {I18} holds.

For statement (IE.j), the following two assertions hold.

$\{(rd[j] \vee wr[j] \vee inc[j] \vee frz[j]) \wedge \text{I18}\}$ (IE.j) {I18} , (IE.j) not effective with this precondition.

$\{\neg rd[j] \wedge \neg wr[j] \wedge \neg inc[j] \wedge \neg frz[j]\}$ (IE.j) $\{\langle \forall z : z \in H \cdot S \wedge z.pnum = j :: z.id < id[j] \rangle\}$, by (I4) and the axiom of assignment.

These two assertions imply that {I18} (IE.j) {I18} holds.

Finally, consider statement (W.k), where $k \neq j$. For this statement, the following assertions hold.

$\{wr[k] \wedge frz[k] \wedge \text{I18}^{H,\ S}_{H \cdot S \cdot k.tup,\ \emptyset}\}$ (W.k) {I18} , by the axiom of assignment.

I18 $\Rightarrow$ $\text{I18}^{H,\ S}_{H \cdot S \cdot k.tup,\ \emptyset}$, by the definition of (I18), and because $(k.tup).pnum \neq j$.

{$wr[k] \wedge frz[k] \wedge$ I18} (W.k) {I18} , by previous two assertions.

{$(\neg wr[k] \vee \neg frz[k]) \wedge$ I18} (W.k) {I18} , (W.k) not effective with this precondition.

The last two of these assertions imply that {I18} (W.k) {I18} holds. □

The following invariant follows from (I18) and because $(j.tup).pnum = j$ and $(j.tup).id = id[j]$.

invariant $inc[j] \wedge \neg frz[j] \wedge alt[j] = 0 \Rightarrow j.tup \notin H$ (I19)

As the following invariant shows, if an INC tuple for a "pending" Increment operation is in H, then that INC tuple is part of a sequence of INC tuples followed by a WRITE tuple.

invariant $j.tup \in H \wedge inc[j] \wedge (frz[j] \vee alt[j] = 1) \Rightarrow \langle \exists \alpha, \beta, w :: w.op = \text{WRITE} \wedge \alpha \cdot j.tup \cdot \beta \cdot w \sqsubseteq H \wedge \langle \forall z : z \in \beta :: z.op = INC \rangle \rangle$ (I20)

Proof: Initially $inc[j]$ is false, and hence (I20) is true. To prove that (I20) is stable, it suffices to consider those statements that may establish $inc[j] \wedge (frz[j] \vee alt[j] = 1)$ or that may modify $j.tup$ or H. The statements to consider are (R.j), (RE.j), (WE.j), (IE.j), (SS.k), (W.k), and (I.k), where $0 \leq k < N$. By (I0), $\neg inc[j]$ holds after each effective execution of (R.j). By (I10), $\neg frz[j] \wedge alt[j] = 0$ holds after each effective execution of (RE.j), (WE.j), and (IE.j). It follows, then, that these statements do not falsify (I20).

For statement (SS.k), we consider the cases $k \neq j$ and $k = j$ separately. For statement (SS.k), where $k \neq j$, the following assertions hold.

$\{rd[k] \wedge \neg frz[k] \wedge \text{I20}^{H}_{H \cdot (\text{READ}, k, id[k], Qval)}\}$ (SS.k) {I20} , by the axiom of assignment.

I20 $\Rightarrow$ $\text{I20}^{H}_{H \cdot (\text{READ}, k, id[k], Qval)}$, by the definition of (I20) and because $k \neq j$.

{$rd[k] \wedge \neg frz[k] \wedge$ I20} (SS.k) {I20} , by previous two assertions.

{$(\neg rd[k] \vee frz[k]) \wedge$ I20} (SS.k) {I20} , variables in (I20) not modified with this precondition.

The last two of these assertions imply that {I20} (SS.k) {I20} holds. For statement (SS.j), we have the following.

$\{inc[j] \wedge \neg frz[j] \wedge alt[j] = 1 \wedge \text{I20}^{frz[j]}_{true}\}$ (SS.j) {I20} , by the axiom of assignment.

$inc[j] \wedge \neg frz[j] \wedge alt[j] = 1 \wedge \text{I20} \Rightarrow \text{I20}^{frz[j]}_{true}$, by the definition of (I20).

$\{inc[j] \wedge \neg frz[j] \wedge alt[j] = 1 \wedge \text{I20}\}$ (SS.j) {I20} , by previous two assertions.

$\{inc[j] \wedge \neg frz[j] \wedge alt[j] = 0\}$ (SS.j) $\{j.tup \notin H\}$, by (I19) and the axiom of assignment.

$\{inc[j] \wedge frz[j] \wedge \text{I20}\}$ (SS.j) {I20} , (SS.j) not effective with this precondition.

$\{\neg inc[j]\}$ (SS.j) $\{\neg inc[j]\}$, by the axiom of assignment.

The last four of these assertions imply that {I20} (SS.j) {I20} holds.

Now, consider statement (W.k). By (I0) and the axiom of assignment, $\neg inc[j]$ holds after each effective execution of (W.j), and hence this statement does not falsify (I20). For (W.k), where $k \neq j$, the following assertions hold.

$wr[k] \wedge \text{I20} \Rightarrow \text{I20}^{H}_{H \cdot S \cdot k.tup}$, by (I3) and (I5), $S \cdot k.tup$ is a sequence of INC tuples followed by a WRITE tuple.

$\{wr[k] \wedge frz[k] \wedge \text{I20}^{H}_{H \cdot S \cdot k.tup}\}$ (W.k) {I20} , by the axiom of assignment.

$\{wr[k] \wedge frz[k] \wedge \text{I20}\}$ (W.k) {I20} , by previous two assertions.

$\{(\neg wr[k] \vee \neg frz[k]) \wedge \text{I20}\}$ (W.k) {I20} , (W.k) not effective with this precondition.

The last two of these assertions imply that {I20} (W.k) {I20} holds.

For statement (I.k), we consider the cases $k \neq j$ and $k = j$ separately. Let $G \equiv inc[k] \wedge frz[k] \wedge alt[k] = 1 \wedge (k.tup \notin H \vee (\neg Ovlap[k] \wedge x[k,k].tag = x[k,N].tag))$. Note that G is the guard of the last parallel assignment of statement (I.k). For statement (I.k), where $k \neq j$, the following assertions hold.

$\{G \wedge \text{I20}^{H}_{(H - k.tup) \cdot k.tup}\}$ (I.k) {I20} , by the axiom of assignment.

$G \wedge \text{I20} \Rightarrow \text{I20}^{H}_{(H - k.tup) \cdot k.tup}$, by (I5), G implies $k.tup$ is an INC tuple; because $j \neq k$, $j.tup \neq k.tup$.

$\{G \wedge \text{I20}\}$ (I.k) {I20} , by previous two assertions.

$\{\neg G \wedge \text{I20}\}$ (I.k) {I20} , variables in (I20) not modified with this precondition.

The last two of these assertions imply that {I20} (I.k) {I20} holds. For statement (I.j), we have the following.

$\{inc[j] \wedge frz[j] \wedge alt[j] = 0 \wedge \text{I20}^{inc[j],\ frz[j],\ alt[j]}_{true,\ false,\ 1}\}$ (I.j) {I20} , by the axiom of assignment.

$inc[j] \wedge frz[j] \wedge alt[j] = 0 \wedge \text{I20} \Rightarrow \text{I20}^{inc[j],\ frz[j],\ alt[j]}_{true,\ false,\ 1}$, by the definition of (I20).

$\{inc[j] \wedge frz[j] \wedge alt[j] = 0 \wedge \text{I20}\}$ (I.j) $\{\text{I20}\}$, by previous two assertions.

$\{inc[j] \wedge frz[j] \wedge alt[j] = 1\}$ (I.j) $\{\neg inc[j]\}$, by the axiom of assignment.

$\{(\neg inc[j] \vee \neg frz[j]) \wedge \text{I20}\}$ (I.j) $\{\text{I20}\}$, (I.j) not effective with this precondition.

The last three of these assertions imply that $\{\text{I20}\}$ (I.j) $\{\text{I20}\}$ holds. □

The following two invariants follow from (I20), the definition of Val, and the fact that each tuple appended to H is unique (the latter can be stated and proved formally, although we will not do so). In particular, because (I20) implies that each "pending" INC tuple in H is part of a sequence of INC tuples followed by a WRITE tuple, it follows that such an INC tuple can be removed from H without changing the value of $Val(H)$ and also without invalidating the value of any READ tuple in H.

invariant $j.tup \in H \wedge inc[j] \wedge (frz[j] \vee alt[j] = 1) \Rightarrow Val(H) = Val(H - j.tup)$ (I21)

invariant $j.tup \in H \wedge inc[j] \wedge (frz[j] \vee alt[j] = 1) \Rightarrow \langle \forall \alpha, r : r.op = \text{READ} \wedge j.tup \in \alpha \wedge \alpha \cdot r \sqsubseteq H :: Val(\alpha) = Val(\alpha - j.tup) \rangle$ (I22)

The following invariant, which was first given in Section 3, relates the "value" of the history variable H to the value of the implemented counter.

invariant $Val(H) = Qval$ (I23)

Proof: Initially, $H = \emptyset$, $Q[N].val = init$, and $\langle \forall j :: Q[j].val = 0 \rangle$. By the definition of Val and $Qval$, this implies that $Val(H) = init$ and $Qval = init$. Hence, (I23) holds initially. To prove that (I23) is stable, we must check each statement that may possibly modify H or Q. The statements to check are (SS.j), (W.j), and (I.j). For statement (SS.j), the following assertions hold.

$\{rd[j] \wedge \neg frz[j] \wedge \text{I23}^{H}_{H \cdot (\text{READ}, j, id[j], Qval)}\}$ (SS.j) $\{\text{I23}\}$, by the axiom of assignment.

$\text{I23} \Rightarrow \text{I23}^{H}_{H \cdot (\text{READ}, j, id[j], Qval)}$, by the definition of Val, $Val(H) = Val(H \cdot (\text{READ}, j, id[j], Qval))$.

$\{rd[j] \wedge \neg frz[j] \wedge \text{I23}\}$ (SS.j) $\{\text{I23}\}$, by previous two assertions.

$\{(\neg rd[j] \vee frz[j]) \wedge \text{I23}\}$ (SS.j) $\{\text{I23}\}$, variables in (I23) not modified with this precondition.

The last two of these assertions imply that $\{\text{I23}\}$ (SS.j) $\{\text{I23}\}$ holds.

Next, consider statement (W.j). For this statement, we have the following.

$\{wr[j] \wedge frz[j] \wedge H \cdot S = \alpha\}$ (W.j) $\{Q[N].val = value[j] \wedge afterw \wedge H = \alpha \cdot (\text{WRITE}, j, id[j], value[j])\}$

, by (I5), precondition implies $j.tup = (\text{WRITE}, j, id[j], value[j])$; postcondition follows by the axiom of assignment.

$\{wr[j] \ \wedge \ frz[j]\}$ (W.j) $\{Q[N].val = value[j] \ \wedge$
$\langle \forall k :: Q[k].val = 0 \ \vee \ Q[k].tag \neq Q[N].tag \rangle \ \wedge \ Val(H) = value[j]\}$
, by previous assertion, (I9), and the definition of Val.

$\{wr[j] \ \wedge \ frz[j]\}$ (W.j) {I23}
, weakening the postcondition of the previous assertion.

$\{(\neg wr[j] \ \vee \ \neg frz[j]) \ \wedge$ I23$\}$ (W.j) {I23}
, (W.j) not effective with this precondition.

The last two of these assertions imply that {I23} (W.j) {I23} holds.

Finally, consider statement (I.j). For this statement, the following assertions hold.

$\{inc[j] \ \wedge \ frz[j] \ \wedge \ alt[j] = 0 \ \wedge \ x[j,j].tag \neq x[j,N].tag \ \wedge$ I23$^{Q[j]}_{(0,\ x[j,N].tag)}\}$ (I.j) {I23}
, by the axiom of assignment.

$inc[j] \ \wedge \ frz[j] \ \wedge \ alt[j] = 0 \ \wedge \ x[j,j].tag \neq x[j,N].tag \ \wedge$ I23 $\Rightarrow$ I23$^{Q[j]}_{(0,\ x[j,N].tag)}$
, by (I17).

$\{inc[j] \ \wedge \ frz[j] \ \wedge \ alt[j] = 0 \ \wedge \ x[j,j].tag \neq x[j,N].tag \ \wedge$ I23$\}$ (I.j) {I23} (A2)
, by previous two assertions.

$\{inc[j] \ \wedge \ frz[j] \ \wedge \ alt[j] = 0 \ \wedge \ x[j,j].tag = x[j,N].tag \ \wedge$ I23$^{Q[j]}_{(alt[j] \times value[j] + x[j,j].val,\ x[j,N].tag)}\}$ (I.j) {I23}
, by the axiom of assignment.

$(alt[j] = 0 \ \wedge$ I23$^{Q[j]}_{(alt[j] \times value[j] + x[j,j].val,\ x[j,N].tag)}) = (alt[j] = 0 \ \wedge$ I23$^{Q[j]}_{(x[j,j].val,\ x[j,N].tag)})$
, predicate calculus.

$inc[j] \ \wedge \ frz[j] \ \wedge \ alt[j] = 0 \ \wedge \ x[j,j].tag = x[j,N].tag \ \wedge$ I23 $\Rightarrow$ I23$^{Q[j]}_{(x[j,j].val,\ x[j,N].tag)}$
, by (I6), left-side of the implication implies $Q[j].val = x[j,j].val \ \wedge \ Q[j].tag = x[j,N].tag$.

$\{inc[j] \ \wedge \ frz[j] \ \wedge \ alt[j] = 0 \ \wedge \ x[j,j].tag = x[j,N].tag \ \wedge$ I23$\}$ (I.j) {I23} (A3)
, by previous three assertions.

$\{inc[j] \ \wedge \ frz[j] \ \wedge \ alt[j] = 0 \ \wedge$ I23$\}$ (I.j) {I23}
, by (A2) and (A3).

$\{(\neg inc[j] \ \vee \ \neg frz[j]) \ \wedge$ I23$\}$ (I.j) {I23}
, (I.j) not effective with this precondition.

By the last two of these assertions, our remaining proof obligation is to show that $\{inc[j] \wedge frz[j] \wedge alt[j] = 1 \wedge \text{I23}\}$ (I.j) $\{\text{I23}\}$ holds. In the remaining assertions, let $B \equiv inc[j] \wedge frz[j] \wedge alt[j] = 1$ and let $C \equiv j.tup \notin H \vee (\neg Ovlap[j] \wedge x[j,j].tag = x[j,N].tag)$. Note that $B \wedge C$ equals the guard of the last parallel assignment of statement (I.j). We first show that $\{B \wedge C \wedge \text{I23}\}$ (I.j) $\{\text{I23}\}$ holds.

$B \wedge C \wedge Val(H) = v \wedge j.tup \notin H \Rightarrow Val(H - j.tup) = v$, by the definition of "$-$", $j.tup \notin H$ implies $H = H - j.tup$.

$B \wedge C \wedge Val(H) = v \wedge j.tup \in H \Rightarrow Val(H - j.tup) = v$, by (I21).

$B \wedge C \wedge Val(H) = v \Rightarrow inc[j] \wedge Val(H - j.tup) = v$, by previous two assertions and the definition of B.

$\Rightarrow inc[j] \wedge Val((H - j.tup) \cdot (\text{INC}, j, id[j], value[j])) = v + value[j]$, by the definition of Val.

$\Rightarrow Val((H - j.tup) \cdot j.tup) = v + value[j]$, by (I5) and the definition of $j.tup$.

$B \wedge C \wedge Qval = v$

$\Rightarrow B \wedge C \wedge Qval^{Q[j]}_{(x[j,j].val,\ x[j,j].tag)} = v$, by (I6), $B \Rightarrow Q[j] = x[j,j]$.

$\Rightarrow B \wedge C \wedge Qval^{Q[j]}_{(x[j,j].val,\ x[j,N].tag)} = v$, by (I14), $B \wedge C \Rightarrow x[j,j].tag = x[j,N].tag$.

$\Rightarrow x[j,N].tag = Q[N].tag \wedge Qval^{Q[j]}_{(x[j,j].val,\ x[j,N].tag)} = v$, by (I13) and (I15), $B \wedge C \Rightarrow x[j,N].tag = Q[N].tag$.

$\Rightarrow x[j,N].tag = Q[N].tag \wedge (Q[N].val + \langle \sum k : k \neq j \wedge Q[k].tag = Q[N].tag :: Q[k].val \rangle + x[j,j].val) = v$, by the definition of $Qval$.

$\Rightarrow x[j,N].tag = Q[N].tag \wedge$
$(Q[N].val + \langle \sum k : k \neq j \wedge Q[k].tag = Q[N].tag :: Q[k].val \rangle + x[j,j].val + value[j]) = v + value[j]$, predicate calculus.

$\Rightarrow x[j,N].tag = Q[N].tag \wedge Qval^{Q[j]}_{(value[j]+x[j,j].val,\ x[j,N].tag)} = v + value[j]$, by the definition of $Qval$.

$B \wedge C \wedge \text{I23} \Rightarrow \text{I23}^{H,\ Q[j]}_{((H-j.tup)\cdot j.tup),\ (alt[j]\times value[j]+x[j,j].val,\ x[j,N].tag)}$, by above derivations; note that B implies $alt[j] = 1$.

$\{B \wedge C \wedge \text{I23}^{H,\ Q[j]}_{((H-j.tup)\cdot j.tup),\ (alt[j]\times value[j]+x[j,j].val,\ x[j,N].tag)}\}$ (I.j) $\{\text{I23}\}$

, by (I14), precondition implies $x[j,j].tag = x[j,N].tag$; hence, when (I.j) is executed, second alternative is assigned to $Q[j]$.

$\{B \ \wedge \ C \ \wedge$ I23$\}$ (I.j) $\{$I23$\}$, by previous two assertions.

Our remaining proof obligation is to show that $\{B \ \wedge \ \neg C \ \wedge$ I23$\}$ (I.j) $\{$I23$\}$ holds. This is proved next.

$B \ \wedge \ \neg C \ \wedge$ I23

$\Rightarrow B \ \wedge \ \neg C \ \wedge$ I23 $\wedge \ (x[j,j].tag \neq x[j,N].tag \ \vee \ x[j,N].tag \neq Q[N].tag)$
, by (I16), $B \ \wedge \ \neg C \ \Rightarrow$ $x[j,j].tag \neq x[j,N].tag \vee$ $x[j,N].tag \neq Q[N].tag$.

$\Rightarrow B \ \wedge \ \neg C \ \wedge$ I23 $\wedge \ (x[j,j].tag \neq x[j,N].tag \ \vee \ (x[j,j].tag = x[j,N].tag \ \wedge \ x[j,j].tag \neq Q[N].tag))$
, predicate calculus.

$\Rightarrow B \ \wedge \ \neg C \ \wedge$ I23 $\wedge \ (x[j,j].tag \neq x[j,N].tag \vee$
$(x[j,j].tag = x[j,N].tag \ \wedge \ Q[j].tag = x[j,N].tag \ \wedge \ Q[j].tag \neq Q[N].tag))$
, by (I6), B implies $x[j,j].tag = Q[j].tag$.

$\Rightarrow B \ \wedge \ \neg C \ \wedge$ I23 $\wedge \ ((x[j,j].tag \neq x[j,N].tag \ \wedge \ (Q[j].val = 0 \ \vee \ Q[j].tag \neq Q[N].tag)) \vee$
$(x[j,j].tag = x[j,N].tag \ \wedge \ Q[j].tag = x[j,N].tag \ \wedge \ Q[j].tag \neq Q[N].tag))$
, by (I17), $B \ \wedge \ x[j,j].tag \neq x[j,N].tag$ implies $Q[j].val = 0 \ \vee \ Q[j].tag \neq Q[N].tag$.

$\Rightarrow B \ \wedge \ \neg C \ \wedge \ x[j,j].tag \neq x[j,N].tag \ \wedge \ \text{I23}^{Q[j]}_{(0,\ x[j,N].tag)} \vee$
$B \ \wedge \ \neg C \ \wedge \ x[j,j].tag = x[j,N].tag \ \wedge \ \text{I23}^{Q[j]}_{(alt[j] \times value[j] + x[j,j].val,\ x[j,N].tag)}$
, by the definition of (I23). Note that the previous assertion implies $Q[j].val = 0 \vee Q[j].tag \neq Q[N].tag$. This implies that $Q[j]$ doesn't contribute to the value of $Qval$.

$\{B \ \wedge \ \neg C \ \wedge \ x[j,j].tag \neq x[j,N].tag \ \wedge \ \text{I23}^{Q[j]}_{(0,\ x[j,N].tag)}\}$ (I.j) $\{$I23$\}$
, by the axiom of assignment.

$\{B \ \wedge \ \neg C \ \wedge \ x[j,j].tag = x[j,N].tag \ \wedge \ \text{I23}^{Q[j]}_{(alt[j] \times value[j] + x[j,j].val,\ x[j,N].tag)}\}$ (I.j) $\{$I23$\}$
, by the axiom of assignment.

$\{B \ \wedge \ \neg C \ \wedge \ $I23$\}$ (I.j) $\{$I23$\}$, by previous two assertions and above derivation.

This completes the proof that {I23} (I.j) {I23} holds. □

We now use the preceding lemmas to prove that Consistency holds.

invariant $\langle \forall \alpha, k, n, v : \alpha \cdot (\text{READ}, k, n, v) \sqsubseteq H :: v = Val(\alpha) \rangle$ (I24)

Proof: Initially $H = \emptyset$, and hence (I24) holds. To prove that (I24) is stable, we must check each statement that may possibly modify H. The statements to check are (SS.j), (W.j), and (I.j). For statement (SS.j), we have the following.

$\{rd[j] \wedge \neg frz[j] \wedge \text{I24}^{H}_{H \cdot (\text{READ}, j, id[j], Qval)}\}$ (SS.j) {I24} , by the axiom of assignment.

$\text{I24} \Rightarrow \text{I24}^{H}_{H \cdot (\text{READ}, j, id[j], Qval)}$, by the invariance of (I23).

$\{rd[j] \wedge \neg frz[j] \wedge \text{I24}\}$ (SS.j) {I24} , by previous two assertions.

$\{(\neg rd[j] \vee frz[j]) \wedge \text{I24}\}$ (SS.j) {I24} , variables in (I24) not modified with this precondition.

The last two of these assertions imply that {I24} (SS.j) {I24} holds.

Next, consider statement (W.j). For this statement, the following assertions hold.

$wr[j] \wedge frz[j] \wedge \text{I24} \Rightarrow wr[j] \wedge frz[j] \wedge \text{I24}^{H}_{H \cdot S \cdot (\text{WRITE}, j, id[j], value[j])}$, by (I3), $S \cdot (\text{WRITE}, j, id[j], value[j])$ contains no READ tuples.

$\Rightarrow wr[j] \wedge frz[j] \wedge \text{I24}^{H}_{H \cdot S \cdot j.tup}$, by (I5) and the definition of $j.tup$.

$\{wr[j] \wedge frz[j] \wedge \text{I24}^{H}_{H \cdot S \cdot j.tup}\}$ (W.j) {I24} , by the axiom of assignment.

$\{wr[j] \wedge frz[j] \wedge \text{I24}\}$ (W.j) {I24} , by previous assertion and above derivation.

$\{(\neg wr[j] \vee \neg frz[j]) \wedge \text{I24}\}$ (W.j) {I24} , (W.j) not effective with this precondition.

The last two of these assertions imply that {I24} (W.j) {I24} holds.

Finally, consider statement (I.j). Let B and C be as defined in the proof of (I23). Then, the following assertions hold.

$B \wedge C \wedge j.tup \notin H \wedge \text{I24} \Rightarrow \text{I24}^{H}_{H - j.tup}$, by the definition of "$-$", $j.tup \notin H$ implies $H = H - j.tup$.

$B \wedge C \wedge j.tup \in H \wedge \text{I24} \Rightarrow \text{I24}^{H}_{H - j.tup}$, by (I22).

$B \wedge C \wedge \text{I24} \Rightarrow \text{I24}^{H}_{(H - j.tup) \cdot j.tup}$, by (I5) and previous two assertions;

(I5) implies $j.tup$ is an INC tuple.

$\{B \wedge C \wedge \text{I24}^{H}_{(H-j.tup)\cdot j.tup}\}$ (I.j) {I24} , by the axiom of assignment.

$\{B \wedge C \wedge \text{I24}\}$ (I.j) {I24} , by previous two assertions.

$\{(\neg B \vee \neg C) \wedge \text{I24}\}$ (I.j) {I24} , H not modified with this precondition.

The last two of these assertions imply that {I24} (I.j) {I24} holds. □

5 Discussion

In the following subsections, we discuss several issues pertaining to our counter implementation.

5.1 Handling Overflows

In Section 3, we assumed that the value of the implemented counter ranges over the integers. To implement a counter that stores values over some bounded range, our implementation must be modified to prevent overflows. An overflow may result if an Increment operation is performed when the value of the counter is "very close" to the maximum allowed. Overflows can be dealt with in two ways: we can either modify the Increment procedure so that potential overflows are detected and avoided; or we can allow the value of the counter to "wrap around" when an overflow occurs. Incorporating the latter approach into our implementation is straightforward. For example, to implement a counter whose value ranges over $0..L-1$, we need only modify statements 0 and 7 in Figure 3 so that when *outval* and *sum* are computed, addition is performed modulo L.

Overflows can be detected and avoided by testing the value assigned to *sum* by the Increment procedure. Suppose, for example, that each *val* field in Q ranges over $-L..L$. In this case, if $|sum| \leq L$ holds following the execution of statement 7, then $Q[i]$ is updated as before. However, if $|sum| > L$, then an error code is returned to process i and $Q[i]$ is not modified. This approach has the disadvantage that the counter does not have a single "maximum value": for a given value of the counter, an Increment operation by one process may cause an overflow error, while an Increment operation with the same input value by another process does not. This inconsistency results from the fact that overflow for process i depends only on the value of $Q[i].val$, and not on the value of any other component.

5.2 Complexity

The time complexity of an implementation is defined to be the number of reads and writes of composite registers required to execute an operation of the implemented counter. It is easy to see that the time complexity of each Read, Write, and Increment operation in our implementation is $O(1)$. The space complexity of an implementation is defined to be the number of single-reader, single-writer, single-bit atomic registers required to realize the implementation. If the implemented counter stores values ranging over $\{-L, \ldots, L\}$, then by the results of [2, 3, 4], the space complexity is polynomial in L and N.

5.3 Generalizing the Implementation

In the proof of Consistency, we considered an arbitrary well-formed history h, and showed that the partial order on operations defined by h can be extended to a total order $\prec$ that is consistent with the semantics of a counter. In particular, we proved that the output value of each Read operation r in h equals $y + z$, where y and z are defined as follows.

- If there exists a Write operation w such that $w \prec r \ \wedge \ \neg\langle \exists v : v \text{ is a Write operation} :: w \prec v \prec r \rangle$, then y equals the input value of w and z equals the sum of the input values of all Increment operations ordered between w and r by $\prec$.
- If no such w exists, then y equals the initial value of the implemented counter, and z equals the sum of the input values of all Increment operations ordered before r by $\prec$.

In more general terms, the protocol followed in the implementation allows each Read operation to determine two values y and z, where y is the most recently written value according to $\prec$, and z is a function over the input values of all intervening Increment operations according to $\prec$. Note that this protocol does not enable a Read operation to determine the relative ordering of the intervening Increment operations. However, this ordering is irrelevant in determining the value of the counter because Increment operations are defined in terms of addition, which is associative and commutative. This protocol can be generalized to yield the following theorem.

Theorem: Any shared register X that can either be read, written, or modified by a PRMW operation of the form "$X := X \circ v$," where $\circ$ is an operator that is associative and commutative and v is an integer value, can be implemented in a bounded, wait-free manner from atomic registers. □

As an example, consider the problem of implementing a "multiplication register," i.e., one that can either be read, written, or multiplied by an integer value. We can implement such a register by defining the value of the implemented register to be

$$Q[N].val \times \langle \prod j : 0 \leq j < N \ \wedge \ Q[j].tag = Q[N].tag :: Q[j].val \rangle$$

and by modifying statement 0 of the Read procedure accordingly. The procedure used to multiply the value of the register would be similar to the Increment procedure in Figure 3, except that in statements 5 and 7, "$sum := 0$" would be replaced by "$sum := 1$," and in statement 7, "$sum := x[i].val + inval$" would be replaced by "$sum := x[i].val \times inval$" (actually, *prod* would be a more suitable variable name than *sum* in this case). The initialization of the register would be similar to that given in Figure 2, except for the requirement $\langle \forall j : 0 \leq j < N :: Q[j].val = 1 \rangle$.

5.4 Implementing More Powerful Shared Data Objects

One may wonder whether atomic registers can be used to implement even more powerful shared data objects in a wait-free manner, i.e., ones that may be modified by means of numerous PRMW operations. In order to partially address this question, we consider the problem of implementing a register that combines the operations

shared var $X : 1..8$
initially $X = 1$

process 0
private var
  $decide$: **boolean**; { decision variable }
  $y : 1..8$
begin
  0: **increment** $X := X + 3$;
  1: **read** $y := X$;
  2: $decide := (y = 4 \ \vee \ y = 8)$
end

process 1
private var
  $decide$: **boolean**; { decision variable }
  $z : 1..8$
begin
  3: **multiply** $X := X \cdot 2$;
  4: **read** $z := X$;
  5: $decide := (z \neq 2 \ \wedge \ z \neq 5)$
end

Figure 6: A solution to the consensus problem.

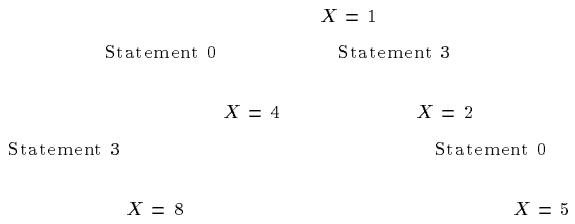

Figure 7: Possible values of X.

of both counters and multiplication registers; we call such a register an *accumulator register*. In the remainder of this subsection, we show that the following theorem holds.

Theorem: Accumulator registers cannot be implemented from atomic registers without waiting. □

The proof is based upon the problem of two-process consensus. In the consensus problem, two processes are required to agree on a common boolean "decision value"; trivial solutions in which both processes agree on a predetermined value are not allowed. It has been shown by Anderson and Gouda [5], by Chor, Israeli, and Li [12], by Herlihy [15], and by Loui and Abu-Amara [23] that two-process consensus cannot be solved in a wait-free manner using only atomic registers. Therefore, to prove that accumulator registers cannot be implemented from atomic registers without waiting, it suffices to prove that accumulator registers can be used to solve two-process consensus in a wait-free manner.

Figure 6 depicts a program that solves two-process consensus without waiting by using a single shared accumulator register X. To see that this program solves the consensus problem, consider Figure 7. This figure depicts the possible values of X; each arrow is labeled by the statement that causes the change in value. Based on this figure, we conclude that if statement 0 is executed before statement 3, then the final value of y equals 4 or 8 and the final value of z differs from 2 and 5, in which case both processes decide on "true." On the other

hand, if statement 3 is executed before statement 0, then the final value of z equals 2 or 5 and the final value of y differs from 4 and 8, in which case both processes decide on "false." Thus, this program solves the consensus problem.

6 Concluding Remarks

We have shown that there exist nontrivial shared data objects with PRMW operations that can be implemented from atomic registers in a bounded, wait-free manner. In particular, we have presented an implementation that can be generalized to implement any shared data object that can either be read, written, or modified by an associative, commutative PRMW operation. Our implementation is polynomial in both space and time, and thus is an improvement over the unbounded implementations of Aspnes and Herlihy [7].

It is interesting to note that our results can be applied to extend the notion of a composite register by allowing an additional associative, commutative PRMW operation on each component. As an example, consider the problem of implementing an array of counters that can be written or incremented individually or read collectively in a single snapshot. An individual counter can be implemented by using a single composite register as described in Section 3. A set of counters that can be read atomically can be implemented in a straightforward fashion by combining the composite registers that implement the individual counters into a single composite register. This approach can be further generalized as discussed in Section 5.3 for the case of other associative, commutative PRMW operations.

The results of this paper provide yet another example of an unbounded wait-free implementation that can be made bounded. An interesting research question is whether it is possible to develop a general mechanism for converting any unbounded wait-free implementation into a bounded one, provided the data object under consideration is syntactically bounded.[4] This question was noted previously by Afek et al. [2].

The correctness proof for our implementation is noteworthy because of the fact that it is assertional, rather than operational. Most proofs of wait-free implementations that have been presented in the literature are based upon operational concepts such as histories and events. Such proofs require one to mentally execute the program at hand (e.g., "... if process i does this, then process j does that ..."), and thus are quite often error prone and difficult to understand. In our proof, auxiliary history variables are used to record the effect of each operation; these history variables serve as a basis for stating the required invariants.

The use of auxiliary history variables in correctness arguments is, of course, not new. Early references include the work of Clint [13] and also Owicki and Gries [25]. More recent references include Abadi and Lamport's work on refinement mappings [1] and Lam and Shankar's work on module specifications [20]. Our use of history variables was, in fact, motivated by the latter paper, where history variables are used to formally specify database serializability. We believe that history variables better facilitate the development of assertional proofs of wait-free implementations than do shrinking functions and their variants [8].

Acknowledgements: We would like to thank A. Udaya Shankar for several helpful discussions on the subject of this paper, and for bringing reference [20] to our attention. We would also like to thank John Tromp for his comments on a draft of this paper. Special thanks go to the anonymous referees for their helpful remarks.

[4] For example, a counter whose value ranges over the integers is syntactically unbounded.

Appendix: UNITY Programming Notation

In this appendix, we describe the UNITY programming notation used in Figure 5. Our treatment will be necessarily brief and our descriptions operational. For a complete description of UNITY, the interested reader is referred to [11].

A UNITY program consists of four sections, namely **declare**, **always**, **initially**, and **assign**. The **declare** section gives variable declarations. The **always** section defines auxiliary variables that are functions of other variables. The **initially** section specifies initial conditions. The **assign** section gives the executable statements of the program.

The operator $[\!]$ specifies nondeterministic choice and the operator $\|$ specifies parallel composition. Thus, "$a := c \;[\!]\; b := d$" denotes *two* separate assignments, while "$a := c \parallel b := d$" denotes a *single* parallel assignment that is equivalent to "$a,\ b := c,\ d$." The operator $[\!]$ is treated as $\wedge$ when it appears in the **always** or **initally** section. Quantified statements are allowed. For example, "$\langle [\!]i : 0 \leq i < N :: a[i] := b[i] \rangle$" is shorthand for "$a[0] := b[0] \;[\!]\; a[1] := b[1] \;[\!]\; \cdots [\!]\; a[N-1] := b[N-1]$," and "$\langle \| i : 0 \leq i < N :: a[i] := b[i] \rangle$" is shorthand for "$a[0],\ a[1],\ \ldots,\ a[N-1] := b[0],\ b[1],\ \ldots,\ b[N-1]$."

A simple multiple-assignment s may have a boolean guard B, denoted "s **if** B." The operational interpretation of such a statement is as follows: if "s **if** B" is executed when B is false, then the program state is not changed, and if it is executed when B is true, then the program state is updated by performing assignment s. In the latter case, the execution of the statement is said to be *effective*. It is important to note that the statements comprising a single parallel assignment may have different guards. Consider, for example, the parallel assignment "s **if** $B \parallel t$ **if** C." If this statement is executed when B and C are both false, then the program state is unchanged; if it is executed when B is true (false) and C is false (true), then the program state is updated by performing the assignment s (t); finally, if it is executed when B and C are both true, then the program state is updated by performing both assignments s and t. The notation "$a := b$ **if** $B \sim c$ **if** C" is used as shorthand for "$a := b$ **if** $B \parallel a := c$ **if** C"; this notation may be used only if B and C never hold at the same time.

Operationally, a UNITY program is executed from state to state; at each state an assignment statement is selected nondeterministically and is executed. (A parallel assignment is considered to be a single assignment statement; in Figure 5, the assignment statements are labeled.) The execution of such a statement may or may not be effective. To ensure progress, statement selection is required to be fair. However, because linearizability is a safety property [16], we have no need to consider such issues.

References

[1] M. Abadi and L. Lamport, "The Existence of Refinement Mappings," *Theoretical Computer Science*, Vol. 82, No. 2, May 1990, pp. 253-284.

[2] 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.

[3] J. Anderson, "Multiple-Writer Composite Registers," Technical Report TR.89.26, Department of Computer Sciences, University of Texas at Austin, September 1989.

[4] J. Anderson, "Composite Registers," *Proceedings of the Ninth Annual Symposium on Principles of Distributed Computing*, 1990, pp. 15-30. To appear in *Distributed Computing*.

[5] J. Anderson and M. Gouda, "The Virtue of Patience: Concurrent Programming With and Without Waiting," Technical Report TR.90.23, Department of Computer Sciences, University of Texas at Austin, August 1987, revised July 1990.

[6] J. Anderson and M. Gouda, "A Criterion for Atomicity," *Formal Aspects of Computing: The International Journal of Formal Methods*, scheduled to appear in 1992.

[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. Also appeared in *Proceedings of the Sixth Annual Symposium on Principles of Distributed Computing*, 1987, pp. 249-259.

[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," *Proceedings of the Sixth Annual Symposium on Principles of Distributed Computing*, 1987, pp. 86-97.

[13] M. Clint, "Program Proving: Coroutines," *Acta Informatica*, Vol. 2, p. 50-63, 1973.

[14] 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.

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

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

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

[18] 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.

[19] C. Kruskal, L. Rudolph, M. Snir, "Efficient Synchronization on Multiprocessors with Shared Memory," *ACM Transactions on Programming Languages and Systems*, Vol. 10, No. 4, October 1988, pp. 579-601.

[20] S. Lam and A. Shankar, "Specifying Modules to Satisfy Interfaces: A State Transition System Approach," Technical Report CS-TR-2082.3, University of Maryland at College Park, 1988. To appear in *Distributed Computing*.

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

[22] 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.

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

[24] 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.

[25] S. Owicki and D. Gries, "An Axiomatic Proof Technique for Parallel Programs I," *Acta Informatica*, Vol. 6, pp. 319-340, 1976.

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

[27] 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.

[28] 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.

[29] 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.

[30] 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.