Towards a Necessary and Sufficient Condition for Wait-free Synchronization[*] (Extended Abstract)

James H. Anderson and Mark Moir

Department of Computer Science, The University of Maryland
College Park, Maryland 20742-3255 U.S.A.

Abstract. We define a class of shared objects called *snapshot objects*, and give a necessary and sufficient condition for the existence of a wait-free implementation of such objects from atomic registers. Snapshot objects can be accessed by means of a read operation that returns the entire object state, or by a set of operations that do not return values. Our condition for the existence of a wait-free implementation requires that for any pair of operation invocations, either the two invocations commute or one overwrites the other.

1 Introduction

The resilient implementation of shared objects is a subject that has received much recent attention. A *shared object* is a data structure that is accessed by a collection of processes by means of a fixed set of operations. An implementation of a shared object is *k-resilient* iff any process can complete any operation in a finite number of steps, provided at most k other processes fail undetectably. An N-process implementation is *wait-free* iff it is $(N-1)$-resilient.

Most prior research on resilient shared objects has focused on implementations based on atomic registers. An *atomic register* is a shared object consisting of a single shared variable that can be either read or written in a single operation [14]. An N-reader, M-writer, L-bit atomic register consists of an L-bit value that can be read by N processes and can be written by M processes.

Various shared objects have been shown to have wait-free implementations using only the simplest kind of atomic registers, i.e., single-reader, single-writer, single-bit atomic registers. These include multi-reader, multi-writer, multi-bit atomic registers [4, 7, 8, 11, 12, 14, 15, 17, 18, 19, 20, 21], multi-reader, multi-writer counters [5, 6], and multi-reader, multi-writer composite registers [1, 2, 3, 13]. A *composite register* is a generalization of an atomic register, consisting of a set of registers that may be written individually or read collectively in a single

[*] Work supported, in part, by NSF Contract CCR 9109497 and by the Center of Excellence in Space Data and Information Sciences. Both authors are now with the Department of Computer Science, The University of North Carolina at Chapel Hill, Chapel Hill, North Carolina 27599-3175

snapshot operation. Other objects have been shown to have no resilient implementation from atomic registers, including multiple-register assignment objects [9], multiply-add registers [5], and read-modify-write objects [9].

In this paper, we consider the problem of precisely characterizing those shared objects that can be implemented in a resilient manner from single-reader, single-writer, single-bit atomic registers. We take a significant step towards obtaining such a characterization by considering a class of shared objects called *snapshot objects*, and by establishing a necessary and sufficient condition on such objects for the existence of a wait-free implementation using only atomic registers. Snapshot objects can be modified by a fixed set of operations that do not return values, and can be read in their entirety by any process by means of a special *Read* operation. The condition we establish for such objects – hereafter called the *Resiliency Condition* – requires that for each pair of operation invocations, either the two invocations commute or one overwrites the other. Informally, two invocations *commute* if the order in which they are applied is irrelevant, and an invocation p *overwrites* an invocation q if invoking q and p in sequence results in the same object state as just invoking p by itself.

In the first part of this paper, we show that the Resiliency Condition is necessary by showing that any snapshot object that fails to satisfy it can be used to implement a 1-resilient consensus protocol. Loui and Abu-Amara [16] have shown that such a protocol cannot be implemented using only atomic registers. It follows that if a snapshot object does not satisfy the condition, then it has no 1-resilient implementation from atomic registers. With regard to sufficiency, Aspnes and Herlihy [6] have presented an algorithm that provides a wait-free implementation for a large class of objects. This class includes any snapshot object that satisfies the Resiliency Condition, thereby showing the condition to be sufficient. Thus, a given snapshot object either has a wait-free implementation from atomic registers, or does not even have a 1-resilient implementation.

Unfortunately, Aspnes and Herlihy's implementation has unbounded space complexity. In the second part of this paper, we present a bounded, wait-free implementation for a class of snapshot objects that satisfy the condition. The requirement for an object to be in this class is that the relationship between each pair of operations is *static*: that is, for any pair of operations P and Q, either each invocation of P commutes with each invocation of Q, or each invocation of P (Q) overwrites each invocation of Q (P). This class of objects turns out to be quite large and includes many objects for which bounded, wait-free implementations have been presented in the literature. It also includes many objects for which no bounded, wait-free implementation was previously known. An example of such an object is one consisting of four variables w, x, y, and z, with atomic operations for reading all four variables, or writing all four variables, or writing w and x, or writing y and z, or writing any individual variable. That such an object has a wait-free implementation may be surprising, given that multiple-register assignments cannot be implemented from atomic registers without waiting [9].

The remainder of this paper is organized as follows. In Section 2, we define some notation, and in Section 3, we formally state the Resiliency Condition and show that it is necessary. In Section 4, we describe the bounded, wait-free implementation discussed above. Concluding remarks appear in Section 5.

2 Definitions

A *snapshot object* is a data structure that is shared by a collection of processes. Each state of the object is an integer value; one state is designated as the *initial state*. A process may inspect or change the state of the object by invoking one of a fixed set of operations. It is not necessarily the case that all processes can invoke all operations. Each operation takes zero or more parameters, and when applied to the object with those parameters, may change the object's state. We call each such application an *invocation* of the operation. Only one operation, the *Read* operation, returns a value. The *Read* operation, which can be invoked by any process, takes no parameters, and returns the state of the object.

An implementation of an object consists of a set of program fragments that processes execute in order to invoke operations. Within such a program fragment, the only shared objects that may be invoked are atomic registers (or objects that may in turn be implemented from atomic registers with the desired resiliency – see below). The primary correctness condition for an implementation of an object is that of *linearizability* [10].

A process may fail by stopping undetectably, in which case it is said to be *faulty*. A process that does not fail is *nonfaulty*. As described in the introduction, an implementation of an object is said to be *k-resilient* iff each nonfaulty process is able to correctly complete all invocations in a finite number of steps, provided at most k processes fail. An implementation of an object for N processes is said to be *wait-free* iff it is $(N-1)$-*resilient*.

Unless specified otherwise, we henceforth denote operations using P and Q, and invocations of operations using p and q. We define an *object history* as a sequence $s_0 \xrightarrow{p_0} s_1 \xrightarrow{p_1} s_2 \cdots \xrightarrow{p_{n-1}} s_n$, where s_0 is the initial state of the object, and for each object state s_i, s_{i+1} is the state of the object reached by applying invocation p_i when the object is in state s_i. We denote the last state of the object in an object history H as $\langle H \rangle$. We use H_j to denote the j-th invocation in H and $H|_i$ to denote the sequence of invocations from H that are invoked by process i. We denote the object history obtained by appending an invocation p to an object history H as $H \cdot p$. For an object history H and two invocations p and q, p and q are said to *commute after* H iff $\langle H \cdot p \cdot q \rangle = \langle H \cdot q \cdot p \rangle$. Similarly, p is said to *overwrite* q *after* H iff $\langle H \cdot q \cdot p \rangle = \langle H \cdot p \rangle$.

3 Necessary and Sufficient Condition

In this section, we state a condition on snapshot objects and show that it is necessary and sufficient for the existence of a resilient implementation.

Resiliency Condition: For every history H of the object and for every pair of invocations p and q of different processes, either p and q commute after H, or one overwrites the other after H.

Theorem (Necessary and Sufficient Condition): A snapshot object has a wait-free implementation iff it satisfies the Resiliency Condition. Furthermore, if an object does not satisfy the Resiliency Condition, then it not only has no wait-free implementation, it does not even have a 1-resilient implementation.

Proof: The sufficiency of this condition follows from results presented by Aspnes and Herlihy in [6], where it is shown that any object (and thus any snapshot object) satisfying the condition has an unbounded, wait-free implementation. Aspnes and Herlihy's condition is slightly stronger in that they require every pair of invocations to commute or overwrite, even for invocations of the same process. However, their algorithm is still correct if our weaker condition is satisfied.

In the remainder of the proof, we show that the Resiliency Condition is necessary. Specifically, we show that any snapshot object that does not satisfy the condition can be used to solve 1-resilient consensus. It has been shown by Loui and Abu-Amara [16] that 1-resilient consensus cannot be solved using only atomic registers. Thus, an object that does not satisfy our condition has no 1-resilient implementation using only atomic registers. We first define the 1-resilient consensus problem, and then describe an algorithm that solves 1-resilient consensus using any snapshot object that does not satisfy the condition.

1-Resilient Consensus Problem: To solve 1-resilient consensus, each nonfaulty process decides either *true* or *false* by assigning a local decision variable, which it may write only once. Provided at most one process fails, every nonfaulty process must eventually decide, according to the following conditions.

- *Agreement*: Each nonfaulty process is required to choose the same value.
- *Nontriviality*: There is a run in which all nonfaulty processes decide *true* and a run in which all nonfaulty processes decide *false*. □

If an object X does not satisfy the Resiliency Condition, then there is an object history H for X, and invocations p and q by different processes such that p and q do not commute after H, and neither overwrites the other after H. Call the processes that invoke p and q, u and v, respectively. We now describe how X can be used to solve 1-resilient consensus.

Assuming, for a moment, that all the invocations in H have been applied to X in order, we can solve 1-resilient consensus as follows. Process u invokes p and then reads the state of the object. Similarly, process v invokes q and then reads the state of the object. The only states that can be returned by u's read are $\langle H \cdot p \rangle$, $\langle H \cdot p \cdot q \rangle$, and $\langle H \cdot q \cdot p \rangle$. Note that $\langle H \cdot p \rangle \neq \langle H \cdot q \cdot p \rangle$ because p does not overwrite q, and $\langle H \cdot p \cdot q \rangle \neq \langle H \cdot q \cdot p \rangle$ because p does not commute with q. Thus, u can determine from the state it reads whether q was invoked first. Similarly, v can determine whether q was invoked first. We call this technique *tie-breaking* and the invocations p and q *tie-breakers*.

In the above description of the tie-breaking technique, we assumed that the invocations in H had been applied to X in order. The problem remains of "replaying" the invocations in H. This is difficult because we have no way of synchronizing the processes to ensure that the invocations are applied in the correct order. The key to solving this problem is to attempt to replay the invocations in the correct order, and to detect any errors in the replaying, taking appropriate action to ensure that a consistent decision is still reached.

The operations in H are replayed as follows. We use a bounded, wait-free counter CTR, and a composite register $LIST$ of size $L+1$, where L is the number of invocations in H. It follows from the results in [1, 2, 3, 5, 6, 13] that such objects can be implemented from atomic registers in a wait-free (and therefore 1-resilient) manner. Each process i performs the invocations in $H|_i$ in the order in which they appear in H. To perform an invocation in H, process i uses the *Replay* procedure shown in Fig. 1. If, after all calls to *Replay* have taken place, the $LIST$ variable has values 1 through L in positions 1 through L, then the object history H was correctly replayed. If two invocations in H occur out of order, then the associated increment and read statements of *Replay* are also executed out of order, so $LIST$ will not contain the correct values. We use this fact to detect the situation in which H is correctly replayed. Note that it is possible that H is correctly replayed, and the values in $LIST$ are incorrect, so it will appear as if H were not correctly replayed. This can happen if the increment of CTR for one invocation happens before the read of CTR for the previous invocation, but the invocations themselves occurred in the correct order. The important point is that whenever it is detected that H was replayed correctly, H was replayed correctly. We now describe what each process does after it has finished performing all of its invocations from H.

After replaying their invocations in H, processes u and v determine a decision value by executing *Finish_u* and *Finish_v* (Fig. 1), respectively; this value is recorded in the variable DV. Each other process, after replaying its invocations in H, waits for one of u and v to update DV, and then returns the value decided. At most one process can fail, so one of u and v eventually updates DV.

In some circumstances, processes u and v decide on a default value, and in other circumstances they use their tie-breaking invocations p and q to modify the object, and then reach a decision based on the state of the object. It is important to note that one process might be using its tie-breaker while the other is "dropping out" with a default value. When a process drops out, it does not apply its tie-breaker invocation.

We now show that the algorithm satisfies the Agreement requirement by considering all possible ways in which processes u and v determine a decision value during invocations of *Finish_u* and *Finish_v*, respectively. In the proof, we consider the cases of $LIST$ being incorrect and $LIST$ being incomplete separately. To be incorrect, some element i of $LIST$ must contain a non-zero value other than i. To be incomplete, some element of $LIST$ must still contain zero. It is important to note that each element of $LIST$ is written only once; thus, if $LIST$ is incorrect, then it will continue to be incorrect, and if $LIST$ is complete and correct, then it will continue to be complete and correct.

Case 1 - Process u reads LIST$[L+1] \neq 0$*, or reads incorrect* LIST*, or reads incomplete* LIST*.* Process u drops out and returns *true*. Either process v drops out and returns *true*, or process v applies its tie-breaker. If process v applies its

```
shared variables
  $LIST$ : array$[1..L+1]$ of $1..L+1$;
  $DV$ : { $none$, $true$, $false$ };
  $CTR$ : $0..L$
initially
  $(\forall j : 1 \le j \le L+1 :: LIST[j] = 0) \wedge$
  $DV = none \wedge CTR = 0$
local variables
  $x$ : array$[1..L+1]$ of $1..L+1$;
  $y$ : $1..L$;
  $val$ : integer
```

```
procedure $Replay(j : 1..L)$
  $CTR := CTR + 1$;
  invoke $H_j$ on $X$;
  $y := CTR$;
  $LIST[j] := y$;
  return
```

```
procedure $Finish\_u()$ returns boolean
  $x := LIST$;
  if $x[L+1] = 0 \wedge$
   $(\forall j : 1 \le j \le L :: x[j] = j)$ then
     invoke $p$ on $X$;
     read $val$ from $X$;
     $DV := (val = \langle H \cdot q \cdot p \rangle)$;
     return($DV$)
  else
     $DV := true$;
     return($true$)
   fi
```

```
procedure $Finish\_v()$ returns boolean
  $LIST[L+1] := 1$;
  $x := LIST$;
  if $(\forall j : 1 \le j \le L :: x[j] = j)$ then
     invoke $q$ on $X$;
     read $val$ from $X$;
     $DV := (val \ne \langle H \cdot p \cdot q \rangle)$;
     return($DV$)
  else
     $DV := true$;
     return($true$)
  fi
```

Fig. 1. Replaying a Sequence of Invocations.

tie-breaker, then because u does not apply its tie-breaker, v reads $\langle H \cdot q \rangle$, and thus returns *true*.

Case 2 - Process u reads LIST$[L+1] = 0$ *and* LIST *is complete and correct.* Process u applies its tie-breaker. Process v has not yet set $LIST[L+1]$ to 1, so it has not yet read $LIST$. Thus, when process v reads $LIST$, it is correct and complete, so process v also applies its tie-breaker. As described above, when both processes u and v apply their tie-breaker operations, they both decide on the same value.

Case 3 - Process v reads incorrect LIST. Process v drops out and returns *true*. Either process u reads $LIST$ with the same error, or process u reads $LIST$ before the erroneous value is written, in which case that element of $LIST$ is zero. In either case, process u drops out and returns *true*.

Case 4 - Process v reads incomplete LIST. Process v drops out and returns *true*. If process u reads $LIST$ before process v, then it also reads $LIST$ incomplete, and drops out returning *true*. If process u reads $LIST$ after process v, then it reads $LIST[L+1] = 1$, and drops out, returning *true*.

Case 5 - Process v reads complete and correct LIST. Process v applies its tie-breaker. Either process u drops out and returns *true*, or process u applies its tie-breaker. If process u drops out, then process v reads $\langle H \cdot q \rangle$, and thus also returns *true*. If process u applies its tie-breaker, then the values decided upon by processes u and v are the same, as in Case 2.

In each case, both processes decide on the same value, so the Agreement requirement is fulfilled. The Nontriviality requirement is straightforward because if the calls to *Replay* occur sequentially and in the correct order according to H, and if both u and v apply their tie-breakers, then both decide *false* if process u applies its tie-breaker first, and *true* if process v applies its tie-breaker first. $\square$

4 Implementation of a Subclass of Snapshot Objects

In this section, we describe a bounded, wait-free implementation for a subclass of the snapshot objects shown by Aspnes and Herlihy [6] to have wait-free, but unbounded, implementations. We first describe the subclass of objects implemented, and then give an informal description of the implementation itself.

4.1 Subclass of Snapshot Objects Implemented

In the Resiliency Condition stated in Section 3, the relationship between operations is *dynamic* in the sense that invocations of a pair of operations P and Q may commute after one history, while an invocation of one overwrites an invocation of the other after another history. We provide a bounded implementation for snapshot objects in which operations are related *statically*. That is, for any pair of operations P and Q, either each invocation of P commutes with each invocation of Q after every history, or each invocation of P overwrites each invocation of Q after every history, or each invocation of Q overwrites each invocation of P after every history. We say that P and Q commute, P overwrites Q, or Q overwrites P, respectively. We also require several other properties of the operations being implemented. We define these properties next, and then give an example object, to which we refer throughout the description of the implementation.

We require that for each operation P, there is a function f that takes two arguments – a state of the object v and a list x of parameters for the operation – and returns the state of the object that is the result of applying P with parameter list x to v. Note that for each operation P, either P overwrites P or P commutes with P. If P does not overwrite P, in which case P commutes with P, then we require that three functions, *initialize*, *compose*, and *apply* exist, such that

$$(\forall v, x :: \mathit{apply}(v, \mathit{initialize}(x)) = f(v, x)), \text{ and}$$
$$(\forall v, x, i :: \mathit{apply}(v, \mathit{compose}(i, x)) = f(\mathit{apply}(v, i), x)).$$

The intuition here is that the *initialize* function takes a parameter list and returns an *intermediate form*, which is a bounded integer value representing that parameter list. The intermediate form is used to represent the composition of parameters from multiple invocations of the operation. The *compose* function takes a value of the intermediate form i and the parameter list x for another invocation of the operation, and returns a new value of the intermediate form that represents the composition of x with the parameters represented by i. The *apply* function takes a state of the object v and a value of the intermediate form i and returns a new state of the object that is the result of applying the operation to the object in state v with each of the parameters represented by i. For example, for three invocations with parameters x, y, and z of an operation represented by function f, the new object state is $f(f(f(v,x),y),z)$. By applying the axioms stated above, this expression can be transformed into $\mathit{apply}(v, \mathit{compose}(\mathit{compose}(\mathit{initialize}(x),y),z))$. This allows the combined effect of these invocations to be computed without knowing the state of the object in advance, which is important to our implementation. Many natural operations satisfy these requirements, and in particular, if the size of the object is bounded, these functions can be constructively shown to exist for any operation that commutes with itself. The snapshot objects implemented here are further restricted to have finitely many operations and finitely many processes. The following example object satisfies all of our requirements.

Example: Our example object is comprised of four 16-bit registers. We use the notation $\langle w,x,y,z\rangle$ to denote states of the object, and ignore the details of coding states of the object to and from natural numbers. The object's initial state is $\langle 0,0,0,0\rangle$. The following operations are available: *CLR* (clear all registers), *SET*(u) (set all registers to u), *CNZ* (clear the leftmost non-zero register), and *RNZ* (reverse the bits of the leftmost non-zero register). As with all snapshot objects, we also have a *Read* operation that reads the entire object. For this example, *CLR* and *SET* overwrite all four operations (including themselves), *CNZ* commutes with *CNZ*, *RNZ* commutes with *RNZ*, and *CNZ* overwrites *RNZ*. □

4.2 Implementation Overview

Before proceeding with a detailed description of the implementation, a brief overview is in order. We divide operations into *overwriting* and *commuting classes*, and define a partial order over these classes. We order the classes in any total order consistent with this partial order. Invocations write to a composite register. The total ordering, along with a system of tags maintained in the composite register, allows the state of the object to be computed from an atomic snapshot of the composite register. We now describe how the classes and partial order for a particular object are constructed.

4.3 Constructing Overwriting and Commuting Classes

Any operation that does not overwrite itself (and thus commutes with itself) is put into a commuting class of its own. We then define the overwriting classes to be the maximal subsets of the remaining operations such that all operations in an overwriting class overwrite each other. Every operation is in exactly one class, because each operation that was not put into a commuting class overwrites at least itself, and thus is in an overwriting class. Next, we define a relation $\prec$ over the classes as follows: $A \prec B$ *iff* $(\exists P, Q : P \in A, Q \in B :: P \ \mathit{overwrites}\ Q) \wedge A \neq B$. The *overwrites* relation is transitive [6], so $\prec$ is transitive and irreflexive, and thus defines an irreflexive partial order over the overwriting and commuting classes. We label the classes 1 to C according to some arbitrary total order that extends the partial order defined by the $\prec$ relation. It is important to note that for any pair of operations P and Q in different classes, if P overwrites Q then P's class comes before Q's class in this labeling. The implemented object is specified using four data structures, *anc*, *ow*, *func*, and *init*, which are described below.

Example (continued): Operations *CNZ* and *RNZ* do not overwrite themselves, and are thus put into commuting classes – call them S and T, respectively. The remaining operations – *CLR* and *SET* – overwrite each other, so we have just one overwriting class U containing both operations. The $\prec$ relation gives $U \prec S$, $U \prec T$, and $S \prec T$. There is only one total order (U, S, T) that extends this partial order, so the classes S, T, and U are labeled 2, 3, and 1, respectively. □

4.4 Data Structures

The *anc* data structure is a square array of boolean values. Letting A (B) denote the i-th (j-th) class in the total order, $\mathit{anc}[i, j]$ is true *iff* $A \prec B$. Thus, *anc* represents each class's *ancestors* in the partial order – that is, the set of classes whose operations overwrite operations in that class. The *ow* data structure is a one-dimensional array of boolean values. The value of $\mathit{ow}[i]$ is true *iff* the i-th class is an overwriting class. The *func* data structure is a $C \times M$ array of functions, where M is the number of operations in the largest overwriting class, or three, whichever is greater. These functions, which represent the operations, are of different types for overwriting and commuting classes. For an overwriting class, we have the functions that represent the operations in that class. For a commuting class, the first three positions contain the *apply*, *compose*, and *initialize* functions, respectively (which is why M is at least three). The *init* data structure represents the initial state of the object.

Example (continued): In our example, *anc* and *ow* are defined as follows.

$$\mathit{anc} = \begin{array}{|c|ccc|} \hline & 1 & 2 & 3 \\ & (U) & (S) & (T) \\ \hline 1\ (U) & 0 & 1 & 1 \\ 2\ (S) & 0 & 0 & 1 \\ 3\ (T) & 0 & 0 & 0 \\ \hline \end{array} \qquad \mathit{ow} = \begin{array}{|ccc|} \hline 1 & 2 & 3 \\ (U) & (S) & (T) \\ \hline 1 & 0 & 0 \\ \hline \end{array}$$

The following functions represent operations *CLR* and *SET* in class U.

$\mathit{func}[1,1](\mathit{obj}, u) \equiv \langle 0, 0, 0, 0 \rangle$
$\mathit{func}[1,2](\mathit{obj}, u) \equiv \langle u, u, u, u \rangle$

The first clears all registers, and the second sets all registers to the value of the parameter. The following functions represent the operation *CNZ* in class S.

$$
\begin{array}{lll}
func[2,1](obj,i) & \equiv obj \text{ with first } i \text{ non-zero registers cleared} & \{\ apply\ \} \\
func[2,2](i,u) & \equiv min(i+1,4) & \{\ compose\ \} \\
func[2,3](u) & \equiv 1 & \{\ initialize\ \}
\end{array}
$$

The *initialize* function starts a count of how many registers to clear. The *compose* function increments the count, and the *apply* function clears the number of registers that the count indicates. The following functions represent the operation *RNZ* in class T.

$$
\begin{array}{lll}
func[3,1](obj,i) & \equiv \begin{cases} obj \text{ with first non-zero reg. reversed} & \text{if } i=1 \\ obj & \text{if } i \neq 1 \end{cases} & \{\ apply\ \} \\
func[3,2](i,u) & \equiv 1-i & \{\ compose\ \} \\
func[3,3](u) & \equiv 1 & \{\ initialize\ \}
\end{array}
$$

The *initialize* function notes that the bits of the leftmost non-zero register are to be reversed. The *compose* function toggles the value of the intermediate form to indicate whether the bits of the leftmost non-zero register should be reversed. If the value of the intermediate form is 1, then the *apply* function reverses the bits of the leftmost non-zero register, otherwise the object is unchanged. □

Having described how the implemented object is specified, we now describe the composite register used in the implementation, and the algorithm itself.

4.5 Composite Register Used in the Implementation

The composite register used by the implementation is a $C \times 2N$ array R of *cells*, where N is the number of processes. Each cell contains the following fields: *valid*, *op*, *pars*, *tag*, *atags*, *seen*, *cnt*, and *done*. The *valid* field is a boolean value that is used to invalidate a cell, signifying that the invocations it represents were overwritten while in progress. The *op* field is used to identify the last operation performed in the cell. The *pars* field records the intermediate form of composing parameters in a commuting class, and the most recent parameter in an overwriting class. The *tag* field is used by classes "lower" in the partial order to detect when they have been overwritten by this operation, and by cells in the same overwriting class to determine which is the most recent invocation. The *atags* (ancestor tags) fields are used to record copies of the values of the *tag* fields of all cells, and are examined in order to detect whether the invocations represented in this cell have been overwritten. The *seen* field is used to record tags currently in use by other cells in this class. The *cnt* field is used to record how many consecutive times the tags recorded in the *seen* field have been seen. The *seen* and *cnt* fields are used to bound the size of the *tag* fields. The *done* field is used to indicate that an invocation has completed.

```
type
    $tag\_type$ = record  $seq$: integer;  $alt$: boolean;  $uniq$: $0..4CN(N+1)+1$  end;
        { $seq$ field in above definition can be restricted to range over $0..10N$ }

    $cell\_type$ = record  $valid$: boolean;  $op$: $1..M$;  $pars$: $0..B$;  $tag$: $tag\_type$;
        $atags$: array$[1..2]$ of array$[1..C,\ 0..2N-1]$ of $tag\_type$;
        $seen$: array$[0..2N-1]$ of $0..4CN(N+1)+1$;  $cnt$: array$[0..2N-1]$ of $0..3$;
        $done$: boolean  end; { Object states, etc. are assumed to range within $0..B$ }

shared constant
    $anc$: array$[1..C,\ 1..C]$ of boolean;   { Indicates ancestors in partial order }
    $ow$: array$[1..C]$ of boolean;           { True for overwriting classes }
    $func$: array$[1..C,\ 1..M]$ of $0..B \times 0..B \rightarrow 0..B$;  { Initialize, compose, apply }
    $init$: $0..B$                            { Initial value of object }

shared variable
    $R$  : array$[1..C,\ 0..2N-1]$ of $cell\_type$;  { Representation of shared object }

initially
    $(\forall i:\ 1 \le i \le C :: (\forall j:\ 0 \le j < 2N :: \neg R[i,\ j].valid \ \wedge\ R[i,\ j].tag.seq = 0))$

definitions
    $sametags(v, tags, j) \ \equiv\ (\forall i : anc[i,j] :: (\forall n : current(v,i,n) :: v[i,n].tag = tags[i,n]))$

    $alive(v,cl,i) \ \equiv\ v[cl,i].done \ \wedge\ v[cl,i].valid \ \wedge\ (ow[cl] \ \Rightarrow\ (\forall n : n \ne i \wedge n \ne i \oplus N \wedge$
        $v[cl,\ n].done :: v[cl,\ n].seen[i] \ne v[cl,\ i].tag.uniq \ \vee\ v[cl,\ n].cnt[i] \ne 4))$

    $current(v,\ j,\ k) \ \equiv\ alive(v,\ j,\ k) \ \wedge\ sametags(v,\ v[j,\ k].atags[0],\ j) \ \wedge$
        $sametags(v,\ v[j,\ k].atags[1],\ j) \ \wedge\ (ow[j] \ \Rightarrow\ (\forall n :\ alive(v,\ j,\ n) \ \wedge$
        $sametags(v,\ v[j,\ n].atags[0],\ j) \ \wedge\ sametags(v,\ v[j,\ n].atags[1],\ j) ::$
        $(v[j,\ n].tag.seq,\ n) \le (v[j,\ k].tag.seq,\ k)))$

local variables
    $v$ : array$[1..C,\ 0..2N-1]$ of $cell\_type$;   { Local copy of object representation }
    $outval$: $0..B$;                              { Output value for $Read$ }
    $i$: $1..C$;  $k$: $0..2N-1$;                   { Counters }
    $q$, $r$: array$[1..C]$ of $0..2N-1$;            { Cell to write in first (second) phase }
    $cell$: $cell\_type$                            { Local copy of cell to be written }
```

Fig. 2. Variable declarations.

4.6 The Algorithm

The implementation consists of two procedures, *Read* and *Invoke*, which are listed in Fig. 3. Associated variable declarations are given in Fig. 2. In this section, we give a brief description of the algorithm. A formal correctness proof will be given in the full paper.

The *Read* procedure determines the state of the object by starting with the

```
**procedure** $Read$ () **returns** $0..B$
    **read** $v := R$;                              { Take snapshot }
    $outval := init$;                               { Start with initial object value }
    **for** $i := 1$ **to** $C$, $k := 0$ **to** $2N-1$ **do**      { For each class and cell ...}
        **if** $current(v,\ i,\ k)$ **then**                  { If not overwritten ... }
            $outval := func[i,\ v[i,\ k].op](v[i,\ k].pars,\ outval)$  { ... apply operation }
        **fi**
    **od**;
    **return**($outval$)                             { Return computed object value }

**procedure** $Invoke$ ($p : 0..N-1;\ cl : 1..C;\ op : 1..M;\ inval : 0..B$)
    **if** $ow[cl]\ \wedge\ r[cl] = p$ **then** $q[cl],\ r[cl] := N+p,\ N+p$ **else** $q[cl],\ r[cl] := p,\ p$ **fi**;
    **if** $\neg ow[cl]$ **then** $q[cl],\ r[cl] := N+p,\ p$ **fi**;

    **read** $v := R$;                              { First phase: take snapshot }
    $cell := v[cl, q[cl]]$;                         { Initialize cell to write }
    $cell.tag.uniq := (min\ n ::\ (\forall j,\ h,\ k ::\ v[j,\ h].atags[0][cl,\ r[cl]].uniq \neq n\ \wedge$
        $v[j,h].atags[1][cl, r[cl]].uniq \neq n\ \wedge\ v[j,h].seen[k]\ \neq n\ \wedge\ v[cl, q[cl]].tag.uniq \neq n))$;
    $cell.tag.alt,\ cell.done := v[cl,\ r[cl]].tag.alt,\ false$;
    **for** $i := 1$ **to** $C$, $k := 0$ **to** $2N-1$ **do** $cell.atags[0][i,\ k] := v[i,\ k].tag$ **od**;
    **for** $k := 0$ **to** $2N-1$ **do**                 { Record observed tags }
        **if** $v[cl,\ q[cl] \oplus N].seen[k] \neq v[cl,\ k].tag.uniq$ **then** $cell.cnt[k] := 0$
        **else** $cell.cnt[k] := min(v[cl,\ q[cl] \oplus N].cnt[k] + 1,\ 4)$
        **fi**;
        $cell.seen[k] := v[cl,\ k].tag.uniq$            { Record $uniq$ values seen }
    **od**;
    **write** $R[cl,\ q[cl]] := cell$;          { Reserve ancestor tags, $uniq$, copy other $uniq$'s }

    **read** $v := R$;                              { Second phase: take snapshot }
    $cell := v[cl,\ r[cl]]$;                        { Initialize cell to write }
    $cell.op,\ cell.done,\ cell.atags[0],\ cell.tag.alt,\ cell.tag.uniq,\ cell.tag.seq :=$
        $op,\ true,\ v[cl,\ q[cl]].atags[0],\ \neg cell.tag.alt,\ v[cl,\ q[cl]].tag.uniq,\ 0$;
    **for** $i := 1$ **to** $C$, $k := 0$ **to** $2N-1$ **do** $cell.atags[1][i,\ k] := v[i,\ k].tag$ **od**;
                                                    { Compose subsequent parameter }
    **if** $\neg ow[cl]\ \wedge\ current(v, cl, r[cl])$ **then** $cell.pars := func[cl, 2](v[cl, r[cl]].pars, inval)$
    **elseif** $\neg ow[cl]\ \wedge\ \neg current(v,\ cl,\ r[cl])$ **then** $cell.pars := func[cl,\ 3](inval)$
    **else** $cell.pars := inval$
    **fi**;
                                                    { Already overwritten? }
    $cell.valid := (ow[cl]\ \Rightarrow\ (\forall n :\ n \neq r[cl]\ \wedge\ n \neq r[cl] \oplus N ::$
        $v[cl,\ n].seen[r[cl]] \neq v[cl,\ r[cl]].tag.uniq\ \vee\ v[cl,\ n].cnt < 2))$;
    **if** $ow[cl]$ **then**                           { Choose $seq$ greater than all alive }
        $cell.tag.seq := (max\ s ::\ (\exists n ::\ (alive(v,\ cl,\ n)\ \wedge\ v[cl,\ n].tag.seq = s))) + 1$
    **fi**;
    **write** $R[cl,\ r[cl]] := cell$;               { Invocation takes effect now, if at all}
    **return**
```

Fig. 3. *Read* and *Invoke* procedures.

initial state of the object, and "applying" all operations that have been invoked, with the following relaxations: any invocation that has been overwritten need not be applied, and the order of invocations of operations that commute need not be preserved. The *Invoke* procedure takes as arguments the process number of the invoking process, the number of the class containing the operation, an index into the *func* array indicating which function to use to apply the operation, and a value representing the parameters to the invocation. The *Invoke* procedure records information in the composite register R so that the *Read* procedure can compute the current state of the object from a single snapshot of R.

Within an overwriting class, only the "most recent" invocation affects the state of the object. To identify this invocation, we use a technique originated by Vitanyi and Awerbuch in [21]. In particular, a sequence number is associated with each invocation. Conceptually, these sequence numbers grow without bound, and the "most recent" invocation is the one with the largest sequence number. To bound the sequence numbers, we use a technique similar to that employed by Anderson in [3] and by Li, Tromp, and Vitanyi in [15]. The idea is to have each invocation choose a value *uniq* that is stored along with its sequence number. Each invocation also makes a copy of each other cell's *uniq* value. The *uniq* value is chosen so as to be distinct from all corresponding copies. When the *uniq* value associated with a given sequence number has been copied by "several" successive invocations, that sequence number is considered "dead", and is no longer compared to other sequence numbers. The algorithm ensures that an invocation has been overwritten before its sequence number is "killed", and that all sequence numbers that have not been killed remain within a bounded range.

Within a commuting class, multiple invocations of an operation are "collapsed" to appear as one invocation. The technique used for this purpose is similar to one employed by Anderson and Grošelj in [5], where a bounded, wait-free algorithm is presented for a counter object that can be modified using *Reset* and *Increment* operations. In [5], *Increment* invocations that have been subsequently overwritten by a *Reset* are detected as follows. *Increment* invocations make a copy of a tag value stored by the most recent *Reset* invocation, and *Reset* invocations choose a tag value that is not currently stored as a copy. If an *Increment* invocation's tag matches that of the most recent *Reset*, then it is considered to have occurred after that *Reset*; otherwise, the *Increment* is considered to have been overwritten by some *Reset* invocation.

Because tags are required to be bounded, complications can arise when a tag's value "wraps around". This problem is dealt with in [5] by having the *Increment* operation execute in two phases. For details, see [5] or the full version of this paper. This technique is generalized in several ways in our algorithm. First, note that each class is potentially an ancestor and a descendant, so operations must perform roles analogous to those of both the *Reset* and *Increment* operations described above. A further complication arises because an invocation that overwrites another invocation in a lower class may itself be overwritten by an invocation in a higher class. We defer a detailed explanation of this complication to the full paper.

5 Concluding Remarks

We have given a necessary and sufficient condition on snapshot objects for the existence of a resilient implementation from atomic registers. The condition only implies the existence of a wait-free implementation with unbounded space complexity. However, we have given an implementation for a large class of such objects that has polynomial space complexity. Our results show that there is no middle ground in the resiliency spectrum for snapshot objects: either a particular snapshot object has a wait-free implementation, or it does not even have a 1-resilient implementation.

An interesting question is whether such a condition exists for more general objects. A major stumbling block in extending our necessity proof is that it depends vitally on a *Read* operation that returns the object state in its entirety. In our implementation, the *Read* operation is used to detect which of the tie-breaker operations was applied first. Note that if a process could read only part of the object's state, it might be impossible to implement such a tie-breaking mechanism. Also, there are objects that can be used to solve 1-resilient consensus that do not provide a *Read* operation; as an example of such an object, consider a boolean variable, initially false, that is accessed only via test-and-set operations.

Acknowledgement: We would like to thank Yehuda Afek, Michael Merritt, and Gadi Taubenfeld for helpful discussions that led to the writing of this paper.

References

1. Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit, "Atomic Snapshots of Shared Memory", *Proceedings of the Ninth Annual Symposium on Principles of Distributed Computing*, 1990, pp. 1-14.
2. J. Anderson, "Composite Registers", *Distributed Computing*, Vol. 6, 1993, pp. 141-154. Preliminary version appeared in *Proceedings of the Ninth Annual Symposium on Principles of Distributed Computing*, 1990, pp. 15-30.
3. J. Anderson, "Multi-Writer Composite Registers", submitted to *Distributed Computing*.
4. J. Anderson and M. Gouda, "A Criterion for Atomicity", *Formal Aspects of Computing: The International Journal of Formal Methods*, Vol. 4, No. 3, May 1992, pp. 273-298.
5. J. Anderson and B. Grošelj, "Beyond Atomic Registers: Bounded Wait-Free Implementations of Nontrivial Objects", *Science of Computer Programming*, 1992, pp. 192-237. Preliminary version appeared as "Pseudo Read-Modify-Write Operations: Bounded Wait-Free Implementations", *Proceedings of the Fifth International Workshop on Distributed Algorithms*, Lecture Notes in Computer Science 579, Springer-Verlag, October 1991, pp. 52-70.
6. 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.

7. 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.
8. 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.
9. M. Herlihy, "Wait-Free Synchronization", *ACM Transactions on Programming Languages and Systems*, Vol. 13, No. 1, 1991, pp. 124-149.
10. 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.
11. A. Israeli and M. Li, "Bounded Time-Stamps", *Proceedings of the 28th IEEE Symposium on Foundations of Computer Science*, 1987, pp. 371-382.
12. L. Kirousis, E. Kranakis, and P. Vitanyi, "Atomic Multireader Register", *Proceedings of the Second International Workshop on Distributed Computing*, Springer Verlag Lecture Notes in Computer Science 312, 1987, pp. 278-296.
13. L. Kirousis, P. Spirakis, and P. Tsigas, "Reading Many Variables in One Atomic Operation: Solutions with Linear or Sublinear Complexity", *Proceedings of the Fifth International Workshop on Distributed Algorithms*, Lecture Notes in Computer Science 579, Springer-Verlag, October 1991, pp. 229-241.
14. L. Lamport, "On Interprocess Communication, Parts I and II", *Distributed Computing*, Vol. 1, 1986, pp. 77-101.
15. 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.
16. M. Loui, H. Abu-Amara, "Memory Requirements for Agreement Among Unreliable Asynchronous Processes", *Advances in Computing Research*, Vol. 4, 1987, pp. 163-183.
17. 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.
18. 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.
19. A. Singh, J. Anderson, and M. Gouda, "The Elusive Atomic Register, Revisited", *Proceedings of the Sixth Annual Symposium on Principles of Distributed Computing*, 1987, pp. 206-221. Expanded version to appear in *Journal of the ACM.*
20. 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.
21. 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.