Atomic snapshots

Uri Abraham
Models for concurrency 2014

June 5, 2014

Abstract

We describe the single-writer registers atomic snapshot algorithm of [2].

1 Atomic snapshots

There are N processes $P_1, \ldots, P_N$. Each process P_i has a register on which it is the only writer and which any process can read. The processes can execute two sorts of operations: *Update* and *Scan*. Process P_i executes $\mathit{Update}_i(v)$ operations where v is a parameter of type *Data*, and any process can execute *Scan* operations which return an array of length N of *Data* values. For any *Update* operation U we denote with $\mathit{val}(U)$ the value of the parameter v with which U is invoked, and for every *Scan* operation S, $\mathit{val}(S)$ is the array of length N that S returns and $\mathit{val}(S)[i]$ is the i-th component of that array. The linear specification is simple. If the operations are linearly ordered (and assuming initial $\mathit{Update}_i(v_i)$ operations) every *Scan* operation S returns an array $a = \mathit{val}(S)$ such that for every index $1 \leq i \leq N$, $a[i]$ is the value of the last Update_i operation that precedes S.

When the operations are not linearly ordered (an operation of process P_i may be concurrent with an operation of P_j) the atomicity requirement is that there is a linear ordering $<^*$ that extends the partially ordered precedence relation $<$ and is such that the linear specifications described above hold for $<^*$.

2 The Single-Writer Scan Algorithm

A variant of the algorithm of [2] is in Figure 1. Every process P_i has a register R_i that carries values that are triples with fields $(data, sequence-number, report)$ where *data* is in *Data*, $sequence-number$ is a natural number, and *report* is an array of length N of *Data* values. We say that such a triple is a *register triple*.

```
Update(v) by P_i:
0.  $s := Scan()$;
1.  $seq_i := seq_i + 1$
2.  $R_i := (v, seq_i, s)$
```

```
Scan (by any process)
0.  for $k := 0$ to $N+1$ $a_k := collect()$;
1.  Case 1: for some $k \in \{0 \cdots N\}$ $a_k = a_{k+1}$
        return $(a_k[1].data, \ldots, a_k[N].data)$;
2.  Case 2: for some $1 \leq i \leq N$ there are indexes
    $0 \leq m < n < N+1$ such that
        $a_m[i] \neq a_{m+1}[i]$ and $a_n[i] \neq a_{n+1}[i]$
    return $a_{n+1}[i].report$
```

```
collect:
0.  for every $1 \leq j \leq N$ $a[j] := R_j$;
1.  return $a$;
```

Figure 1: A variant of the single-writer snapshot algorithm of [2].

A *Scan* operation can be a stand-alone *Scan* or a *Scan* that is part of an *Update* operation. Procedure *collect* is called $N+2$ times in any *Scan* execution. A *collect* execution consists in reading the registers R_i for $1 \leq i \leq N$ in any order and returning the values thus collected. After recording the values obtained in these *collect* invocations in local variables $a_0, \ldots, a_{N+1}$, the *Scan* algorithm decides on the snapshot value to return. There are two modes by which a *Scan* operation can return. A *direct* return is a return via Case 1, and if Case 1 does not apply, then an *indirect* return is via Case 2. We argue below (Lemma 2.1) that if Case 1 does not apply then it must be that Case 2 applies, so that any *Scan* operation returns a value (that is an array of *Data* values).

Consider now the *Update* algorithm for P_i. A local variable seq_i (initialize to 0) is increased by 1 in each execution. So that the ℓ-th *Update* execution has ℓ as the value of its sequence number. If U is an *Update* operation (by P_i) invoked with value v, then we write $v = val(U)$. U consists of some *Scan* operation S which is denoted $Scan(S)$, and a subsequent write on register

R_i of the value (v, ℓ, s) where ℓ is the the sequence number of this execution, and s is the value returned by the *Scan* operation (an array of length N of *Data* values).

Lemma 2.1 *If Case 1 does not apply to a Scan operation execution S, then it must be that Case 2 applies.*

Proof. Suppose that a *Scan* operation does not return via Case 1. Then for every $k \in \{0, \ldots, N\}$ $a_k \neq a_{k+1}$. And since procedure *collect* returns an array of length N (of register triples) there is some $i = i_k$ $(1 \leq i \leq N)$ such that $a_k[i] \neq a_{k+1}[i]$. By the pigeon hole principle there are indices $0 \leq m < n \leq N$ such that $i_m = i_n$. That is, $a_m[i] \neq a_{m+1}[i]$ and $a_n[i] \neq a_{n+1}[i]$. So Case 2 holds. □

This short argument exemplifies an idea that is central to our approach to concurrency, and although it is a very simple idea it is a corner stone of the approach described in [1]. The idea is that the correctness proof of a distributed algorithm can be obtained in three phases.

1. The first phase establishes some properties of operation executions by a process P_i that depend only on the algorithm of P_i and not on the interactions of P_i with the other processes or on the properties of the communication devices that the processes employ.

2. The second phase combines the properties obtained in the first stage with the specifications of the communication devices employed, and obtains some abstract, higher level properties of the operation executions.

3. The third phase takes the abstract properties of the second phase and proves that they entail the desired correctness condition. Tarskian system executions serve in the third phase of the proof.

The argument given above (Lemma 2.1) to show that any *Scan* that does not return via Case 1 must return via Case 2, is an example of a first phase argument. The proof of Lemma 2.1 relies only on the code of the *Scan* operation, it does not rely on any knowledge on the *Update* algorithm or on the registers that are used. The proof relies on the knowledge that the reads of register R_j return register triple values, but not on the atomicity of these registers. In other words, Lemma 2.1 holds true even if the values returned by the read actions are determined randomly, and there is absolutely no reference to the *Update* algorithm in this lemma.

We state in the following two lemmas, without proof, all the properties that are obtained in the first phase of the proof: those that are obtained by reference to the *Scan* operations, and those that are obtained by reference to the *Update* operations.

Lemma 2.2 *Any Scan operation S is either direct or indirect.*

1. *If S is direct then there are two collect executions $c_1 < c_2$ in S such that $val(c_1) = val(c_2)$. Say $a = val(c_1)$ is this common array. The return value of S is $val(S) = (a[1].data, \ldots, a[N].data)$.*

2. *If S is indirect then there is some index $1 \leq i \leq N$ and there are four read actions of register R_i in S, $r_1 < r_2 < r_3 < r_4$, such that $val(r_1) \neq val(r_2)$ and $val(r_3) \neq val(r_4)$. Say $a = val(r_4)$. The value of S in this case is $val(S) = a.report$.*

Lemma 2.3 *If U is the ℓ-th Update operation by process P_i, and if $val(U) = v$ is the value of U, then U consists of a Scan execution $S = Scan(U)$ followed by a write action on register R_i of the value $(v, \ell, val(S))$.*

We emphasize that the proof of Lemma 2.2 depends only on the algorithm of the *Scan* operation, and likewise the proof of lemma 2.3 depends only on the algorithm of the *Update* operation. So essentially these are proofs about serial algorithms, and concurrency issues do not enter in these proofs. A more elaborate discussion of this point will be given in the last section.

Now we move to the second phase of the proof. In this phase we combine the statements of lemmas 2.2 and 2.3 with the semantics of atomic registers. In fact, for simplicity we assume that all registers R_i are serial, and leave to the reader to find why we are allowed to do that. So the read/write actions on the different registers are assumed to be linearly ordered by the precedence relation $<$.

Exercise 1 *Show that if we prove atomicity of the Scan/Update operations under the assumption that the registers are serial, then it follows that even if the registers are assumed to be atomic the desired atomicity of the snapshot algorithm follows.*

If r is any read action of register R_i then $\omega(r)$ denotes the last write action w on register R_i such that $w < r$. Following the semantics of serial registers we get that $\mathit{val}(\omega(r)) = \mathit{val}(r)$. It follows immediately that if $r_1 < r_2$ are two read actions of some serial register R then $\omega(r_1) \leq \omega(r_2)$.

Lemma 2.4 *Suppose that r_1 and r_2 are two read actions of some register R_i such that $\mathit{val}(r_1) = \mathit{val}(r_2)$. Then $\omega(r_1) = \omega(r_2)$.*

Proof. If $\omega(r_1) \neq \omega(r_2)$, then for some $\ell_1 \neq \ell_2$ $w_1 = \omega(r_1)$ is in the ℓ_1th *Update* operation execution and $w_2 = \omega(r_2)$ is in the ℓ_2 *Update*. But in this case the sequence field of w_1 is ℓ_1 and the sequence field of w_2 is ℓ_2, and thus $Val(w_1) \neq \mathit{val}(w_2)$, which implies that $\mathit{val}(r_1) \neq \mathit{val}(r_2)$. □

Let S be a direct *Scan* operation. We define $\Omega(S)$ as the sequence of *Update* operations whose *Data* values S return. In details, the definition of $\Omega(S)$ is as follows.

Definition 2.5 *By Lemma 2.2 there are two collect executions $c_1 < c_2$ in S such that $\mathit{val}(c_1) = \mathit{val}(c_2)$, and if $a = \mathit{val}(c_1)$ then $\mathit{val}(S) = (a[1].\mathit{data}, \ldots, a[N].\mathit{data})$. For every index $1 \leq i \leq N$ we let $r^1(i) \in c_1$ be the read action of register R_i in c_1. Let $w_i = \omega(r^1(i))$ be the corresponding write action, and let W_i be the Update operation to which w_i belongs. We then define $\Omega(S) = (W_1, \ldots, W_N)$. We shall use the notation $\Omega_i(S) = W_i$ for $1 \leq i \leq N$.*

Lemma 2.6 *The following properties of the function Ω_i hold (for $1 \leq i \leq N$). For any direct Scan operation S:*

1. *$W_i = \Omega_i(S)$ is an Update operation by P_i, and $\mathit{end}(W_i) < \mathit{end}(S)$.*
$$\mathit{val}(S) = (\mathit{val}(W_1), \ldots, \mathit{val}(W_N)).$$
2. *If V is any Update operation by P_i such that $V < S$ then $V \leq \Omega_i(S)$.*
3. *If V is an Update operation by P_j such that $\mathit{end}(V) < \mathit{end}(\Omega_i(S))$ (i is any index $1 \leq i \leq N$) then $V \leq \Omega_j(S)$.*

Proof. Items 1 and 2 of the lemma are rather easy to prove, and so we show the proof of item 3. Let S be a direct *Scan* operation and suppose that $c_1 < c_2$ are the two *collect* operations in S such that $\mathit{val}(c_1) = \mathit{val}(c_2)$ as in Lemma 2.2(1). Suppose that $W_i = \Omega_i(S)$ and V is an *Update* operation by P_j such that $\mathit{end}(V) < \mathit{end}(W_i)$ (which means that the write action in V precedes the write action in W_i). We have to prove that $V \leq \Omega_j(S)$. By definition of W_i there is a read action r_i^1 of register R_i in c_1 such that $\omega(r_i^1) = w_i$ is the write action in W_i. By definition of $\mathit{end}(W_i)$, $\mathit{end}(W_i) = w_i$. Now let $v = \mathit{end}(V)$ be its write action, and suppose that $v < w_i$. Let r_j^2 be the read in c_2 of register R_j. We have that $v < w_i < r_i^1 < r_j^2$. Hence $v < r_j^2$, and so $v \leq \omega(r_j^2)$. But if r_j^1 is the read action in c_1 of register R_j, then $\mathit{val}(r_j^1) = \mathit{val}(r_j^2)$ (as $\mathit{val}(c_1) = \mathit{val}(c_2)$). Hence $\omega(r_j^1) = \omega(r_j^2)$ (by Lemma 2.4) and hence $v \leq \omega(r_j^1)$. This implies that $V \leq \Omega_j(S)$ as desired. $\square$

Next, we analyze indirect *Scan* operations. Any *Scan* operation S begins with a series of $N+2$ collect operations $c_0, \ldots, c_{N+1}$ (an execution of line 0)[1]. Every *collect* execution contains N read actions of the registers $R_1, \ldots, R_N$ (in any order). We denote with $\mathit{begin}(S)$ the first read action in c_0, and $\mathit{end}(S)$ denotes the last read action in c_{N+1}. If S_1 and S_2 are *Scan* operations, then $S_1 \sqsubset S_2$ means that $\mathit{begin}(S_2) < \mathit{begin}(S_1)$ and $\mathit{end}(S_1) < \mathit{end}(S_2)$. That is, $S_1 \sqsubset S_2$ says that the temporal extension of S_1 is strictly included in the extension of S_2. Since any process is a serial agent that executes its operations in order, if $S_1 \sqsubset S_2$ then S_1 and S_2 belong to different processes. And since there is a finite number of processes, relation $\sqsubset$ is well-founded: any descending sequence of *Scan* operations must be finite. Specifically, if $S_1 \sqsupseteq S_2 \sqsupseteq \cdots \sqsupseteq S_{N+1}$ is a nested sequence of length $N+1$, then there is an index n with $S_n = S_{n+1}$.

Lemma 2.7 *Suppose that S is an indirect Scan operation. Then there is an Update operation U such that the following holds for $P = \mathit{Scan}(U)$.*

1. *val(S)=val(P).*

2. *$P \sqsubset S$.*

Proof. By Lemma 2.2(2), there is some index $1 \leq i \leq N$ and there are four read actions of register R_i in S, $r_1 < r_2 < r_3 < r_4$, such that $\mathit{val}(r_1) \neq \mathit{val}(r_2)$ and $\mathit{val}(r_3) \neq \mathit{val}(r_4)$. Say $a = \mathit{val}(r_4)$. The value of S is $\mathit{val}(S) = a.\mathit{report}$. Let $w = \omega(r_4)$ be the corresponding write action on register R_i. Then $r_3 < w$ (or else $w < r_3$ would imply that $w = \omega(r_3) = \omega(r_4)$ in contradiction to $\mathit{val}(r_3) \neq \mathit{val}(r_4)$). There is an *Update* operation U such that $w = \mathit{end}(U)$.

[1]Actually there is no need to execute all of these operations. The algorithm can stop immediately after the condition that allows Case 1 or Case 2 holds, and this is how the algorithm is originally presented.

That is, w is the write action of U. Let $P = \mathit{Scan}(U)$. To prove the lemma it suffices to show that $\mathit{begin}(S) < \mathit{begin}(P)$ (and $\mathit{end}(S) < w < r_4 \leq \mathit{end}(S)$ is obvious). Suppose on the contrary that $\mathit{begin}(P) < \mathit{begin}(S)$. Then we have that

$$\mathit{begin}(P) < r_1 < r_2 < r_3 < w.$$

Since the last write on register R_i that precede w must be before P, this relation implies that $\omega(r_1) = \omega(r_2)$ which is in contradiction to $\mathit{val}(r_1) \neq \mathit{val}(r_2)$. □

For any indirect *Scan* operation S we define $\mathit{Report}(S)$ as follows. Let U be that *Update* operation that Lemma 2.7 provides, and define $\mathit{Report}(S) = \mathit{Scan}(U)$. So

$$\mathit{val}(S) = \mathit{val}(\mathit{Report}(S)) \text{ and } \mathit{Report}(S) \sqsubset S. \tag{1}$$

For any *Scan* operation S_0 we define the *report sequence* of S_0, as the sequence $S_0 \sqsupseteq S_1 \sqsupseteq \cdots$ by the following inductive definition. Suppose that S_n is defined. If S_n is a direct *Scan* execution then we define $S_{n+1} = S_n$. If S_n is an indirect *Scan* operation then we define $S_{n+1} = \mathit{Report}(S_n)$. Since $S \sqsupset T$ implies that S and T belong to different processes, there is some $n \leq N$ such that $S_0 \sqsupset \cdots \sqsupset S_n = S_{n+1}$ (and $S_n = S_k$ for all $k > n$). That is, $S_0, \ldots S_{n-1}$ are all indirect *Scan* operations but S_n is a direct one. Thus $\Omega(S_n)$ is defined, and we define $\Omega(S_0) = \Omega(S_n)$. We also define $\Omega_i(S_0) = \Omega_i(S_n)$ for $1 \leq i \leq N$.

Theorem 2.8 *The following properties of the function Ω_i hold (for $1 \leq i \leq N$). For any Scan operation S (direct or indirect):*

1. *$W_i = \Omega_i(S)$ is an Update operation by P_i, and $\mathit{end}(W_i) < \mathit{end}(S)$.*

$$\mathit{val}(S) = (\mathit{val}(W_1), \ldots, \mathit{val}(W_N))).$$

2. *If V is any Update operation by P_i such that $V < S$ then $V \leq \Omega_i(S)$.*

3. *If V is an Update operation by P_j such that $\mathit{end}(V) < \mathit{end}(\Omega_i(S))$ (i is any index $1 \leq i \leq N$) then $V \leq \Omega_j(S)$.*

Proof. Consider the report sequence $S = S_0 \sqsupseteq \cdots \sqsupseteq S_n$. $\mathit{val}(S_0) = \mathit{val}(S_n)$ follows from (1)) and likewise $S_n \sqsubseteq S_0$. So the properties that were established in Lemma 2.6 transfer to the present theorem. For example, the argument for item 2 is the following. If V is an *Update* operation by P_i such that $V < S_0$, then $V < S_n$ (follows from $S_n \sqsubseteq S_0$) and hence $V \leq \Omega_i(S)$ by Lemma 2.6. $\square$

We are now ready for the third phase of the proof: showing that the higher-level properties of the Ω function established in Theorem 2.8 imply the snapshot properties. To make it absolutely clear that the statement of this third phase theorem and its proof are independent of the algorithm, we first restate the properties established in Theorem 2.8 in a separate list in Figure 2. There is an important difference between the formulation in Theorem 2.8 and the properties of Figure 2. There are two kinds of *Scan* operations: standalone operations, and *Scan* operations that are part of some *Update* operation. So far there was no need to separate these operations since they have the same properties, and what the theorem says is true both for the standalone operations and the *Scan* operations that are invoked by *Update* operations. In the list of abstract properties of Figure 2 the term "*Scan* events" refer to the standalone *Scan* events, and there is no need to refer to those *Scan* events that are part of the *Update* events. The point is that now we are interested in the *Update*/*Scan* operations, and the fact that the *Update* operations need (for the algorithm to work) to invoke *Scan* operations is a detail that is abstracted away at this stage of the proof.

The system-execution language in which the properties of Figure 2 are stated has the following components. There are two sorts: the *Event* and *Data* sorts. There are unary predicates *Scan* and *Update* and $P_1, \ldots, P_N$ on the *Event* sort. The binary precedence relation symbol $<$. We have the following function symbols. The *val* function returns *Data* values when defined on *Update* events, and on *Scan* events *val* returns N-tuples of *Data* values. We have $\Omega_i : \mathit{Event} \to \mathit{Event}$ for the function that returns the i-th component of Ω. That is, for every *Scan* event S, $\Omega(S) = (\Omega_1(S), \ldots, \Omega_N(S))$. We also have a function *end* defined on the *Event* sort. Intuitively, $\mathit{end}(E)$ denotes the end of event E, and we think of $\mathit{end}(E)$ as some specific event (we would take it as a temporal value in temporal based system executions, but here we prefer to have system executions in which the time-axis is not present). There are two properties of the *end* function which we shall use.

E1 If $X < Y$ are *Scan*/*Update* events and $X < Y$ then $\mathit{end}(X) < \mathit{end}(Y)$.

E2 For every events A and B, if $A \neq B$ then either $\mathit{end}(A) < \mathit{end}(B)$ or else $\mathit{end}(B) < \mathit{end}(A)$.

S1 Each P_i is a serial process. For every *Update* event U there is a unique process index i such that $P_i(U)$.

S2 For every *Scan* event S, $W_i = \Omega_i(S)$ is an *Update* event by P_i, and $end(W_i) < end(S)$.

$$val(S) = (val(W_1), \ldots, val(W_N))).$$

S3 If V is any *Update* operation by P_i and S is any *Scan* event such that $V < S$, then $V \leq \Omega_i(S)$.

S4 If V is an *Update* operation by P_j and S is a *Scan* event such that $end(V) < end(\Omega_i(S))$ (i is any index $1 \leq i \leq N$), then $V \leq \Omega_j(S)$.

Figure 2: The higher level properties of the [2] snapshot algorithm.

Theorem 2.9 *Let M be any system execution that satisfies the properties listed in Figure 2. Then there is a linear ordering on the Scan/Update events that satisfy the linear snapshot specifications of Section 1.*

The proof begins with the following lemma.

Lemma 2.10 *Let S_1 and S_2 be two Scan events and assume that for some $1 \leq i \leq N$ $\Omega_i(S_1) < \Omega_i(S_2)$. Then, for every $1 \leq k \leq N$, $\Omega_k(S_1) \leq \Omega_k(S_2)$.*

Proof. Suppose on the contrary that $\Omega_i(S_1) < \Omega_i(S_2)$, but for some $1 \leq k \leq N$

$$\neg(\Omega_k(S_1) \leq \Omega_k(S_2)). \tag{2}$$

Then the following is deduced.

1. $\Omega_k(S_2) < \Omega_k(S_1)$.
2. Either $end(\Omega_k(S_1) < end(\Omega_i(S_2))$ or $end(\Omega_i(S_2)) < end(\Omega_k(S_1))$.
3. Suppose first that $end(\Omega_k(S_1) < end(\Omega_i(S_2))$. Then $\Omega_k(S_1) \leq \Omega_k(S_2)$.
4. This is in contradiction to $\Omega_k(S_2) < \Omega_k(S_1)$.
5. Now assume that $end(\Omega_i(S_2)) < end(\Omega_k(S_1))$. Then $\Omega_i(S_2) \leq \Omega_i(S_1)$.

6. This is in contradiction to $\Omega_i(S_1) < \Omega_i(S_2)$.

7. Thus in both cases we have reached a contradiction and hence the lemma follows. $\square$

Exercise 2 *Give a detailed justification for each line in the proof given above to Lemma 2.10. Have you used all properties S1–S4 ?*

Let X and Y be two (different) *Scan/Update* events. We define $X <^* Y$ if and only if one of the following items 1 – 4 holds:

O1 X and Y are both *Update* events and $end(X) < end(Y)$.

O2 X is an *Update* event by process P_k, Y a *Scan* event, and $X \leq \Omega_k(Y)$.

O3 X is a *Scan* event, Y an *Update* event by process P_k, and $\Omega_k(X) < Y$.

O4 X and Y are both *Scan* events and one of the following possibilities is the case:

   (a) For some $1 \leq k \leq N$, $\Omega_k(X) < \Omega_k(Y)$, or

   (b) For all $1 \leq k \leq N$ $\Omega_k(X) = \Omega_k(Y)$ and $end(X) < end(Y)$.

We claim that $<^*$ is the required linear ordering that proves that the algorithm implements an atomic *Scan/Update* system. We have to prove the following.

1. Relation $<^*$ extends the precedence relation $<$.

2. $<^*$ is irreflexive.

3. For any two *Scan/Update* events $X \neq Y$, $X <^* Y$ or $Y <^* X$.

4. Relation $<^*$ is a transitive relation on the *Scan/Update* events.

5. For any *Scan* event S, *val*(S) is the correct array of *Data* values the the snapshot specification requires (under the $<^*$ linear ordering). That is, if *val*$(S) = (d_1, \ldots, d_N)$ then, for every $1 \leq k \leq N$, $d_k =$ *val*(W_k) where W_k is the $<^*$ last *Update* event W such that $W <^* S$.

Exercise 3 *Prove these five items using the following guidelines.*

1. Assume that $X < Y$ are *Scan*/*Update* events. Prove (in great details) that $X <^* Y$ for each of the following cases.
    (a) X and Y are both *Update* events.
    (b) X is an *Update* event, Y is a *Scan* event.
    (c) X is a *Scan* event and Y is an *Update* event.
    (d) X and Y are *Scan* events.
2. This is trivial since we define $X <^* Y$ only when $X \neq Y$, and so $X <^* X$ is impossible.
3. Suppose that X and Y are two different *Scan*/*Update* events. Prove that $X <^* Y$ or $Y <^* X$ in each of the following cases.
    (a) X and Y are both *Update* events.
    (b) X and Y are both *Scan* events.
    (c) One of X and Y is a *Scan* and the other an *Update* event. Say X is a *Scan* and Y an *Update* event.
4. Assume that $X <^* Y <^* Z$. Prove that $X <^* Z$ in each of the following cases.
    (a) X and Y are both *Update* events.
        i. Z is an *Update* event.
        ii. Z is a *Scan* event.
    (b) X is an *Update* event, Y is a *Scan* event.
        i. Z is an *Update* event.
        ii. Z is a *Scan* event.
    (c) X is a *Scan* event and Y is an *Update* event.
        i. Z is an *Update* event.
        ii. Z is a *Scan* event.
    (d) X and Y are *Scan* events. Here you can use Lemma 2.10.
        i. Z is an *Update* event.
        ii. Z is a *Scan* event.

5. Finally, prove that $<^*$ is correct. (This is immediate, but nevertheless give the details.)

Discussion. It may seem quite strange that the correctness proof that was given here to the snapshot algorithm did not consider the notion of state. One may rightly argue that in order to prove the correctness of any algorithm it is necessary to explicate the connection between the text of the algorithm and the resulting executions, and since states are necessary for such an explication no correctness proof can do without them. Indeed states are necessary, but my claim here is that only local states are necessary, and they suffice–we can do without global states. A local state is a function defined on the local variables and which assigns to each such variable a value in its type.

In order to prove Lemma 2.2, for example, we have to consider the algorithm of the *Scan* operation in isolation. The local state variables of the *Scan* operation are the variables k, j, PC (for the program counter), and we also have $N^2 + 2N$ variables $a_k[j]$ for $0 \leq k \leq N+1$ and $1 \leq j \leq N$. The values of these $a_k[j]$ variables are register triples with fields $(data, sequence-number, report)$ where *data* is in *Data*, $sequence-number$ is a natural number, and *report* is an array of length N of *Data* values (see section 2). Registers R_i that the *Scan* operation reads are not among its local state variables. This is a characteristics feature of the *non-restricted* semantics that we give here to the standalone *Scan* semantics. A way to think about it is by realizing that the correctness of Lemma 2.2 can be proved even when the values returned by reads of registers R_j are determined randomly (but within their types). The resulting semantics is said to be "unrestricted" in [1]. In the unrestricted semantics, each process executes its algorithm but the registers that are read return arbitrary values (and the write actions are not recorded in register variables since the registers are not state variables). The restricted semantics is is obtained by imposing on the unrestricted semantics the specifications of the registers (atomicity), and the correctness proof thus relies on the combination of the unrestricted semantics and the specifications of the communication devices.

References

[1] U. Abraham. *Models for Concurrency.* Gordon and Breach, (1999).

[2] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. 1993. Atomic snapshots of shared memory. J. ACM 40, 4 (September 1993), 873-890.

[3] D. Imbs, M. Raynal, A simple snapshot algorithm for multicore systems. Proceedings of the 5th IEEE Latin-American Symposium on Dependable Computing (LADC11), 2011 (IEEE Press, New York, 2011), pp. 1723