A note on the Elusive Atomic Register

Uri Abraham
Department of Mathematics and Computer Science
Ben-Gurion University, Be'er Sheva, Israel.

June 10, 2001

Abstract

A time-efficient version of the *Elusive Atomic Register* of Singh, Anderson, and Gouda [SAG] is presented and its correctness is proved. The concurrent time-complexity of the protocol is 6 for the Reader and 3 for the Writer (it is in the order of the number of Readers in the protocol of [SAG]).

Our aim is to advocate a "behavioral" approach to the correctness proofs of protocols, and this protocol serves as an example. System executions, and not histories (sequences of states) are the structures to which the correctness conditions refer to. An advantage of system executions is the ease of dealing with both the higher–level and lower–level views (granularities) of the runs (Lamport [1986]). We exploit this advantage and our proof is clearly divided into a higher and a lower level. The higher–level part of the proof does not refer to the protocol but to abstract properties, which are in fact shared by another protocol presented in this paper (a variant of HV1 of Haldar and Vidyasankar [1991]), and thus the length of the correctness proof of the second protocol is reduced.

Categories and Subjects Descriptors: D.2.4 [**Software Engineering**]: Programs Verification—*correctness proofs*; F.3.1 [**Logic and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs.

1 Preface

The *Elusive Atomic Register* by Singh, Anderson, and Gouda [1994] ([SAG] in short) describes a protocol for implementing a single-writer, multiple-reader atomic register from single-writer, single-reader atomic registers. The correctness proof is so rigorous and formal that it could well be used as an example for careful presentation and verification of protocols.

Yet my aim is to suggest a different proof methodology, which is perhaps more intuitive, but still quite formal. To do this I will prove the correctness of a similar protocol, using as much as possible the terminology of [SAG], in order to facilitate the comparison. The essence of the difference between the two approaches is that the main notion used by [SAG] is that of a state, while the main notion used here is that of an event. The proof of [SAG] can therefore be called "assertional" (making assertions about states) while the proof here is "behavioral" (it deals with relations between events).

An advantage of our approach is that it allows properties that involve concurrent events, such as regularity of registers. In contrast, [SAG] can only deal with atomic, or serial, registers. This is connected with the "time efficiency" of our protocol in the following manner: A prominent feature of the protocol of [SAG] is the judicious use of the ordering of the read/write events in serial loop executions such as

$$\textbf{for } k := M \text{ to } 1 \textbf{ do write } R_k := \textit{value} \textbf{ od}, \tag{1}$$

where R_k is a register. Now such a serial execution has the disadvantage of being slow, compared to a possible concurrent execution

$$\textbf{forall } k \textbf{ concurrently do write } R_k := \textit{value} \textbf{ od}. \tag{2}$$

Our aim is to introduce a modification of the algorithm of [SAG] that allows such concurrent executions. In fact, we will replace (1) by a single write onto a regular multiple-reader register, and the resulting protocol still works correctly with only one additional read (crucially placed). As the model of [SAG] cannot deal naturally with such concurrent executions (or with regular registers), the correctness of our protocol cannot even be expressed in this model, and we must extend it to allow for concurrent executions of regular read/write statements.

Another advantage of our approach is its clear separation between the lower–level and the higher–level stages of the proof. Since the higher–level part of the correctness proof is shared by another protocol (presented in Section 5) a certain economy in the proof of the second protocol is thus achieved.

2 On system executions

We first describe the model of computation used by [SAG] (namely histories of sequences of states), and we then expand it to allow read and write on regular registers. Then, on top of these extended histories, system executions are built which will later be used in the proof.

A *state* is a map from variables to values. In particular, the values of some variables tell which instructions are *enabled* at the state. Typically, several instructions may be enabled at any state, and each instruction is (identified with) a binary relation on the states. When two states are thus related by an instruction I then we say that I may cause the passage from the first to the second state. The possibility that the two states are equal is not excluded. A *history* is a sequence (finite or of order-type of the natural numbers) of states, each resulting from the previous one by an execution of an instruction that was enabled at that state (except for the first state which is the initial state). The execution of an instruction that caused the passage from one state to the following is called an *event* in the history. (We will call it a *lower-level* event, to distinguish such events from the higher-level events or *operations* as they are called in [SAG].)

The notation $S_1 \overset{e}{\to} S_2$ denotes the fact that S_1 and S_2 are consecutive states in the history and event e caused the passage from S_1 to S_2.

An *operation* is the set of lower-level events obtained by collecting the events in a history corresponding to some complete program execution. This model will be extended to allow for regular registers and concurrent executions such as in (2).

The lower-level events in any history are linearly ordered, and this "temporal" order is denoted with $\prec$. That is, $e \prec f$ (e precedes f) iff the pair of states (s, s') and (t, t') connected by e and (respectively) f are such that s comes before t in the history. The notation $e \preceq f \equiv (e \prec f) \vee (e = f)$ is also used. The operations, which contain several events, are not linearly ordered, but a partial precedence relation, still denoted with $\prec$, can be defined on them by:

$$A \prec B \text{ iff } \forall a \in A \forall b \in B \; (a \prec b) \tag{3}$$

We write $A \not\succ B$ for $\neg(B \prec A)$. We write $A \preceq B$ for $A \prec B \vee A = B$. To each (lower–level) event we can associate two states: the one preceding the event and the one following it. Then, as the set of states is serial, to each (finite) operation X two states may be associated: The first (initial) state of X, i.e., the state previous to the first event of X, and the last one. (Since the operations are not necessarily consecutive sets of events, we cannot say that an operation caused the change from the initial to the last state.) The relation $A \prec B$ on operations can now be defined more concretely: $A \prec B$ iff the last state of A comes before, or is the same as, the first state of B. Similarly, $A \not\succ B$ is equivalent to: The first state of A comes before the last state of B. The following can easily be checked:

1. On the operations of a given history, the precedence relation $\prec$ is an irreflexive partial ordering.

2. If A, B, C, and D are operations in the history such that

   $$A \prec B, \; B \not\succ C, \text{ and } C \prec D,$$

   then

   $$A \prec D.$$

   We will call this the Russell–Wiener property.

3. Assuming that the operations are pairwise disjoint and finite (sets of events) the following finiteness property holds.

   > *For any operation X in the history, there are only finitely many operations Y in that history for which $X \prec Y$ does not hold.*

   We will call this the Lamport finiteness property.

In fact it is not necessary to assume that the operations are pairwise disjoint, and it suffices to assume that each (lower-level) event belongs to only finitely many operations in order to deduce Lamport finiteness property.

Now we describe the semantics of regular registers, and later the semantics of concurrent executions such as (2). If Y is a not an atomic register, then two events rather than one, are associated with each execution of "**read** $x := Y$" or "**write** $Y := value$" (Y is the name of a register, and x is a local variable). These are of type *start* and *conclude*. If, at a state S_1, an instruction "**read** $x := Y$" is enabled (for example) then the pair $S_1 \xrightarrow{st} S_2$ is possible in the history, where st is the "*start* **read** $x := Y$" event. Then, at State S_2, the corresponding *conclude* event, cd, is enabled, and remains enabled until the step $S_1' \xrightarrow{cd} S_2'$ is taken in the history. The value of the local variable x is undefined in the states between S_2 and S_1', and its value at S_2' is determined in accordance with the specifications of regular registers, as described below. This value is called the return value of the read execution.

Just as complete program executions were collected into operations, so corresponding pairs of *start conclude* read/write events are collected to form *compound* read/write events. The precedence relation $\prec$ is defined on these compound events and the operations by the formula (3), as for operations. For example, if $A_1 = \{st_1, cd_1\}$, $A_2 = \{st_2, cd_2\}$ are such compound events, then $A_1 \prec A_2$ iff event cd_1 precedes event st_2 in the history. The regularity of register Y in history H can now be defined following Lamport [1986]. (We use in this definition the function *value*, which gives the value of a read/write compound event.)

Definition 2.1 *Register Y is* regular *in a given history H iff for any (compound) read r of Y there is a (compound) write w on Y such that*

1. *$value(r) = value(w)$,*
2. *$w \not\succ r$ and there is no write w' onto Y such that $w \prec w' \prec r$.*

Equivalently, a register is regular in H if there is a function, ω, defined on the (compound) read events of the register, and taking values that are (compound) write events of the register, such that the following holds:

1. *$value(r) = value(\omega(r))$,*
2. *$\omega(r) \not\succ r$ and there is no write w' onto Y such that $\omega(r) \prec w' \prec r$.*

Such a function ω is called a regular return function.

The notion of a regular register leads quite naturally to the useful notion of a *regular function*.

Definition 2.2 *Suppose that P and Q are sets of compound events (or operations) then a function f, from P into Q, is* regular *iff for all $x \in dom(f)$;*

1. *$f(x) \not\succ x$.*
2. *There is no $q \in Q$ such that $f(x) \prec q \prec x$.*

The following property of regular functions will be frequently used:

Lemma 2.3 *If f is a regular function from P into a serial set Q (i.e., linearly ordered by $\prec$), then for every $x_1, x_2 \in P$, if $x_1 \prec x_2$ then $f(x_1) \preceq f(x_2)^+$. (Here q^+, for $q \in Q$ denotes the successor of q in Q if it exists. So that the statement $q_1 \preceq q_2^+$ means "$q_1 \preceq q_2$ or q_2^+ exists and $q_1 \preceq q_2^+$".)*

Proof. Since Q is serial, if the lemma does not hold, then $f(x_2)^+ \prec f(x_1)$. Yet $f(x_1) \not\succ x_1$ by the regularity of f, and since $x_1 \prec x_2$, $f(x_2)^+ \prec x_2$ follows from the Russell–Wiener property. But then $f(x_2) \prec f(x_2)^+ \prec x_2$ contradicts the regularity of f. ■

The execution of a concurrency statement such as

$$\textbf{forall } i \textbf{ concurrently do } \phi_i \textbf{ od} \tag{4}$$

will also be modelled by a pair of *start* and corresponding *conclude* events. When a statement such as (4) is enabled at a specific state S_1, then a *start* event, *st*, may occur, and then each ϕ_i is enabled. When all of these statements are executed, then the corresponding *conclude* event may occur, and the statement (4) is done. Now *operations* can be defined even for programs that contain concurrent instructions: A *start* and *conclude* events are associated with each execution of the program, and together with all the other events that correspond to the program, the operation is formed. The first and last states of an operation X in a history can be defined as the first state of the *start* event, and the last state of the *conclude* event of X, respectively.

We often use formulas involving the terms $begin(X)$ and $end(X)$, which refer to the *start* and *conclude* events of X in the following manner.

Definition 2.4 *$A \prec end(B)$ means that all the lower-level events that compose A precede the conclude event of B in the history. In this case, for any X, if $B \prec X$ then $A \prec X$. Similarly, $begin(A) \prec B$ implies that if $X \prec A$ then $X \prec B$.*

A simple but useful observation is the following.

Lemma 2.5 *If* $A \prec B \not\vdash C$*, then* $A \prec end(C)$*. If* $A \not\vdash B \prec C$*, then* $begin(A) \prec C$*.*

We continue to use some of the notations of [SAG]. A single Writer process and M reading processes, Reader k, for $1 \le k \le M$, are assumed. The ith operation of the Writer is denoted Writer : i. If $V =$ Writer : i, we would use V^+ to denote Writer : $i+1$. Similar notation is used for the Reader process: If R is a Reader k operation, for example, then R^+ denotes the successor of R by Reader k (if it exists). If P is an operation, either by the Writer or one of the Readers, then $P : i$ denotes the execution (event) of the statement at line i in operation P. This event could be a compound event, containing a *start* and *conclude* subevents. (If this statement is a concurrent **forall** loop, then $P : i.j$ denotes the jth branch of that loop in P).

If z is any variable accessed by operation P in executing its program, then $(P!z)$ denotes the final value of z, as assigned by P. For example, since variable y is set only once, in line 4 of the Reader's program, if R is any Read operation, then $(R!y)$ denotes the value returned by the read event $R : 4$. We will expand this notation and use $(R!\tau)$ to denote the value in R of a propositional form τ (built with variables, equality symbol, logical connectives and (bounded) quantifiers).

The following definition formalizes our discussion so far.

Definition 2.6 *Let H be a history (a sequence of states and events connecting adjacent states as described above). The following two-sorted structure, $\mathcal{H}$, is called* a higher-level system-execution derived from H*. The form (signature) of $\mathcal{H}$ is*

$$\mathcal{H} = (E, N, \prec, F_1, \ldots, F_n, P_1, \ldots, P_k).$$

The universe of $\mathcal{H}$ consists of two sets (being a two-sorted structure): E is a set of (higher-level) events, and N is the set of fixed objects. Each member of E is a finite subset of lower-level events such that no such event belong to infinitely many members of E. E includes:

1. *All compound read/write events in H. As defined, a compound event consists of two lower-level (start and conclude) events.*

2. *All operations in H. An operation is a set of events in H that describe an execution of the protocol (or is an initialization operation).*

3. *Parts of operations that are important enough to be introduced into E. For example, in every Write operation we will identify and single out its G part.*

4. *Singletons containing the start and conclude event of operations and their parts introduced in 2 and 3 are also in E. This gives sense to expressions containing $end(X)$ and $begin(X)$.*

The set N contains all those values that are not events and are needed to describe the execution. For example N contains the natural numbers (to enable an expression such as Writer : i *which denotes the ith operation of* Writer*). The precedence relation $\prec$ on the operations has already been described. And, finally, each F_l is a function and each P_m a predicate on $E \cup N$. The exact description of the functions and predicates will be given as we go deeper into the proof, but one function, ω, is introduced now, it is the regular return function as described in Definition 2.1.*

The following definition is quoted from [SAG]; we find here a problem, and propose the use of system-executions to correct it.

> **Definition**. Let e be the event corresponding to the execution of the statement **read** $z := Y$ in an operation p, where z is a local variable and Y is a shared variable. If the value that p reads from Y is written by operation q, then we say that operation q *determines* $p!z$.

We understand this definition as a relation on triples (q, p, z), rather than as a relation on pairs (p, v) where v is the value of $p!z$. For concreteness, let us fix the local variable z and the shared variable Y (register) and assume that every operation contains a single read statement, **read** $z := Y$, that involves z. Then q *determines* p iff the value that p reads from Y is written by operation q. But now, since it is possible for several operations, indeed for infinitely many operations, to write the same value, there could be infinitely many q's such that q *determines* p. It is clear however that the intention of the authors is to define a functional relation, so that for every p there is just one q which determines p. (For if we read this definition as it is, then the observation following it in [SAG] is certainly false.) The necessary amendment to this definition is obvious and simple: say that q *determines*$_Y$ p iff there is in q an event w which is a write onto register Y, and there is in p a (single) read event r of Y, such that $w \prec r$ and there is no write onto Y, w', in the history such that $w \prec w' \prec r$. This amendment, in fact, introduces a function, let us call it Ω_Y, which assigns to every Reader operation p a Writer operation $q = \Omega_Y(p)$ such that q *determines*$_Y$ p. Now this is surely a trivial correction, but it shows the usefulness of using functions on the higher-level events.

Let us describe now the plan for the rest of the paper. The following section contains the time efficient protocol. To prove its correctness, we will assume a history H resulting from an execution of the protocol, and define the higher-level system-execution $\mathcal{H}$ derived from H, containing some functions and predicates which will be used in the proof. Then we will prove some simple properties of $\mathcal{H}$, and finally, at the higher-level stage of the proof, we will only use these properties (and not any other details of the protocol) to prove the atomicity of the implemented register. Section 5 contains another protocol which shares with the first the same higher–level properties.

3 A time efficient protocol

There are M reading processes, Reader i, for $1 \leq i \leq M$, and one writing process, the Writer. Each process contains, besides an initial event, executions of the protocol described below (the executions by a single process are serially ordered). The protocol implements a single-writer, multiple-reader atomic register from single-writer, multiple-reader regular registers. It is in the use of regular registers that our protocol is different from the protocol of [SAG], which uses single-writer, single-reader atomic registers. The registers (shared variables) used are the following:

1. *WR* is regular, written by Writer and read by all Readers. It carries values of *WRtype*, where:
   WRtype = **record** *old*, *new* : *valtype*; *oldseq*, *seq* : **array**$[1..M]$ **of** $0..4$; *alt* : $0..2$; *done*: **boolean end**.

2. *RW*$[i]$ is regular, written by Reader i and read by Writer. It carries values from $\{0, \ldots, 4\}$.

3. $RR[i]$ is regular, written by Reader i and read by all Readers. It carries values of *RRtype* which are either the special "masking" value $*$, or else in **record** $seq: 0..4$; $alt: 0..2$ **end**.

The initial values of the $RR[i]$ registers are $*$, that of the $RW[i]$ registers is 0, and for the WR register we fix an initial message in the *old* and *new* fields, the value **true** in field *done*, and 0 for the other fields.

We prefer to think of the initial values of the registers as being determined by initial write events. Such an agreement has an advantage even in case of atomic registers in that it allows to say that the value of any read r of a register is the value of the last write event preceding it; without such an agreement we would have to distinguish between reads that have no writes before them (and then they return the initial value) and other reads. So, Writer : 0 and Reader : 0 are not executions of the protocol, but initializing write events on the registers. These events precede any other event. We will say Read or Write events when referring to executions of the protocols, and Reader or Writer events when the initializing writes are also included.

The initial values of the local variables are as follows: *new* is the initial message, *alt* and *oldseq*$[i]$ are 0. The (private) variables *new*, *alt*, and x_0 retain their values from the previous operation.

program Writer(*val* : *valtype*)
begin

0: (*old*, *new*) := (*new*, *val*); *alt*:= *alt*+1 $\pmod 3$;

1: **forall** k $(1 \leq k \leq M)$ **concurrently do read** $seq[k]$:= $RW[k]$ **od**;

2: **write** WR := (*old*, *new*,*oldseq*, *seq*, *alt*, **false**); *oldseq* := *seq*;

3: **write** WR := (*old*, *new*,*oldseq*, *seq*, *alt*, **true**);
end.

---

program Reader($i : 1..M$) returns *valtype*;
define:

$sensitive \equiv y.seq[i] \in \{x_0.oldseq[i],\ x_0.seq[i]\} \wedge (x_1.alt = y.alt \vee x_1.alt = y.alt + 1 \pmod 3)$;

$direct \equiv sensitive \wedge y.done$;

$cued \equiv \ sensitive \wedge \exists k(1 \leq k \leq M) \ v[k] = (y.seq[k], y.alt);$

$flag \equiv \ direct \vee cued;$

begin

0: $r := \{x_0.oldseq[i], x_0.seq[i]\}$; **read** $x_0 := WR$;
   find$d \in \{0,1,2,3,4\} \setminus (\{x_0.oldseq[i], x_0.seq[i]\} \cup r)$;

1: **write** $RW[i] := d$

2: **read** $x_1 := WR$;

3: **forall** k $(1 \leq k \leq M)$ **concurrently do**

   read $v[k] := RR[k]$ **od**;

4: **read** $y := WR$;

5: **if** $flag$ **then write** $RR[i] := (y.seq[i], y.alt)$ **else write** $RR[i] := *$ **fi**;

6: **if** $flag$ **then return** $y.new$ **else return** $y.old$ **fi**;
   end.

Line 3 of the Writer protocol (the last write) is called the G part of the protocol. Lines 0–4 of the Reader protocol are called the F part of the protocol. The F part is thus all the instructions before the write on $RR[i]$ and the return.

3.1 Defining the higher-level view

Consider a history H (a sequence of states) that describes executions of our Reader/Writer protocol. Each process, Writer or Reader i, executes its protocol a finite or infinite number of times. While the $i+1$ operation of each process follows its ith operation, no ordering is imposed on the events of different processes. Let $\mathcal{H}$ be the resulting higher-level system execution with set of events E, obtained by forming the pairs of type (*start*, *conclude*) that constitute compound read/write events, and collecting the operations and parts of operations as defined below. Let ω be the regular return function defined on these read events (Definition 2.1). In the following we are going to define (1) four unary predicates on the Read operations, called *sensitive*, *direct*, *cued*, and *flag*, (2) the functions Ω_i for $0 \le i \le M$, (3) the function G, and (4) a relation "R_2 *cued* R_1" defined on Read operations.

Definition 3.1 *1.* Read *i operation R is said to be in the sensitive mode iff* $(R!sensitive)$ *holds.*

So we have defined here a predicate, and it is to make our discourse more natural that we use the expression "R is sensitive" rather than sensitive(R).

Let R be any Read *operation. We say that R is in the direct mode iff* $(R!direct)$.

Read *operation R is said to be in the* cued *mode iff* $(R!cued)$ *holds.*

Read *operation R is said to be in the* flag *mode iff* $(R!flag)$ *holds.*

2. The write event associated with line 3 of the Writer *program is so important that we give it a special status. For every Write operation W the singleton $\{W : 3\}$ is added to E, and we define $G(W) = \{W : 3\}$, and call it "the uncertainty region" of W. Remark that for every Write operation W*

$$begin(W) \prec G(W) \prec end(W).$$

3. The function Ω_0 is defined on the Read events, and is taking Writer *values: If R is a Read event, then $\Omega_0(R)$ is the unique* Write *operation U such that U contains the write event $w = \omega(R : 4)$. (We can say then that U determines$_{WR}$ $(R : 4)$)*

The regularity of register WR implies that Ω_0 is regular. According to Definition 2.1 this means that $\Omega_0(R) \not\vdash R$ and that there is no Writer operation W such that $\Omega_0(R) \prec W \prec R$. In fact, $\Omega_0(R) \not\vdash F(R)$.

4. The functions Ω_k, for any k such that $1 \le k \le M$, are defined on the sensitive Read *operations. The range of Ω_k is* Reader k. *The definition is the following: If R is a sensitive* Read *operation, then $\Omega_k(R)$ is the unique* Reader k *operation such that $\Omega_k(R)$ determines$_{RR[k]}$ R. That is, let r be $R : 3.k$ (the unique read of register $RR[k]$ in R), and then $\Omega_k(R)$ is the unique* Reader k *operation that contains the write event $w = \omega(R : 3.k)$.*

Since the reads of registers $RR[k]$ exist in any Read operation, we could have defined $\Omega_k(R)$ for any Read operation R, but we are going to use $\Omega_k(R)$ only for sensitive Reads, and so we prefer to think of Ω_k as a partial function only defined on sensitive Reads.

The regularity of $RR[k]$ implies that $\Omega_k(R)$ is regular.

5. *It is useful to define on the Write operations functions that reflect the reads of line 1 (namely, the reads of registers $RW[k]$). We mark these functions by Ω'_k. Given any Write operation, W, $R = \Omega'_k(W)$ is the Reader k operation that contains $\omega(W : 1.k)$. That is $(R : 1)$ $determines_{RW[k]}$ W.*

6. *If a Read R is in cued mode then R is sensitive and for some k*

$$v[k] = (y.seq[k], y.alt) \tag{5}$$

   holds in R. If $S = \Omega_k(R)$, then we say that R is cued by S.

The relation "R is *cued* by S" can be expressed in the following equivalent form. Let us first make a definition.

Definition 3.2 *For each k, $1 \leq k \leq M$, the following equivalence relation $\sim_k$ is defined on the* Write *operations. For any* Write *operations U and V,*

$$U \sim_k V \text{ iff } (U!(seq[k], alt)) = (V!(seq[k], alt)).$$

In words, U and V are $\sim_k$ equivalent if they have the same alt value and obtain the same value in reading $RW[k]$.

Now the following lemma is quite easy to prove.

Lemma 3.3 *For a* Read *R, if $S = \Omega_k(R)$, then R is cued by S iff S is in flag mode and $\Omega_0(R) \sim_k \Omega_0(S)$.*

Proof. Assume that R is *cued* by S. Then S is in *flag* mode, for otherwise S writes the value $*$ on its $RR[k]$ register and the defining equality (5) cannot hold then.

Since R is *cued* by S, then R is in the *cued* mode (and hence in the *sensitive* mode) and (5) holds. Since $S = \Omega_k(R)$, the value of $v[k]$ obtained by R in reading $RR[k]$ is the value written by $S : 5$, namely

$$S!(y.seq[k], y.alt).$$

This and (5) show that $S!(y.seq[k], y.alt) = R!(y.seq[k], y.alt)$. Thus $\Omega_0(S) \sim_k \Omega_0(R)$. The other direction of the lemma is left to the reader. ∎

In [SAG] the return function Φ returns integers, and for that reason we had to introduce the integers into $\mathcal{H}$. I feel that the treatment here would be more cohesive if instead of integers the return function would give directly Write operations. Our *return function* will be called Ω (with no index). The definition of $W = \Omega(R)$ for a Read i operation, R, is done in two cases: If $R!flag$ then $W = \Omega_0(R)$. If, $\neg(R!flag)$, then W is such that $W^+ = \Omega_0(R)$. Such $W = \Omega_0(R)^-$ can be found because if $\Omega_0(R) = \text{Writer} : 0$ is the initial Write, then R is in *direct* mode (see the following lemma). In any case $\Omega(R) \preceq \Omega_0(R) \preceq \Omega(R)^+$.

Lemma 3.4 *1. If R is not in the flag mode, then $\Omega(R) \prec end(R)$ (and thus $G(\Omega(R)) \prec end(R)$).*

2. If $\Omega_0(R)$ is the initial write, then R is in direct mode.

Proof. Item (1) follows from $\Omega(R) \prec \Omega_0(R) \not\vdash R$, and Lemma 2.5. To prove (2) let R be a Read i operation and assume that $\Omega_0(R) = \text{Writer} : 0$ is the initial write, let $r_0 = R : 0$, $r_2 = R : 2$, $r_4 = R : 4$. Then by definition of Ω_0, $\omega(r_4)$ is the initializing write on WR, with values $seq[i] = 0$, $alt = 0$, and $done = \textbf{true}$. Hence, if we show that R is *sensitive*, then R is in *flag* mode. We claim first that $(R!x_1.alt)$ is either 0 or 1, and hence that condition $(R!x_1.alt = y.alt \vee x_1.alt = y.alt + 1 \pmod 3)$ holds. If this is not the case, and $(R!x_1.alt) = 2$, then $\omega(r_2)$ is not in $(\text{Writer} : 0)$ nor in $(\text{Writer} : 1)$ which wrote values 0 and 1 in field *alt*. Hence $W = \text{Writer} : 2$ exists, and $\omega(r_2)$ is in W. The regularity of WR implies $W \not\vdash r_2$. Now, $(\text{Writer} : 1) \prec W \not\vdash r_2 \prec r_4$ implies $(\text{Writer} : 1) \prec r_4$ and hence $\omega(r_4)$ cannot be the initial write!

We claim next that $R!\,0 \in \{x_0.oldseq[i], x_0 seq[i]\}$, and thus conclude that R is sensitive. Well, if $\omega(r_0)$ is the initializing write, then both fields *oldseq* and *seq* contain seroz. If $\omega(r_0)$ is the first write in $(\text{Writer} : 1)$ then the value of $oldseq[i]$ is 0 (the initial value). The same argument as in the preceding paragraph shows that there are no other alternatives for $\omega(r_0)$. ∎

3.2 Basic properties

We are going to establish now some basic properties of the relations and functions defined so far.

It is quite obvious from their definitions that the functions Ω_i are regular. This regularity follows from the corresponding properties of ω, which reflect our assumption that the registers used are regular.

Lemma 3.5 *If R_1 and R_2 are* Read *operations such that $R_2 = \Omega_k(R_1)$, then $(R_2 : 4) \prec (R_1 : 4)$, and thus $F(R_2) \prec end(F(R_1))$.*

Proof. This is an immediate consequence of the Russell–Wiener property applied to $(R_2 : 4) \prec (R_2 : 5) \not\vdash (R_1 : 3.k) \prec (R_1 : 4)$.

Lemma 3.6 *If* Read *operation R is in the direct mode and $V = \Omega_0(R)$, then $(V : 3) \not\vdash (R : 4)$.*

Proof. If R is in the *direct* mode, then $(R!direct)$, and thus $(R!y.done)$ holds. This means that $R : 4$ returns the value true in field *done*. Let $w = \omega(R : 4)$ be the write onto *WR* that determines $R : 4$. Then, by definition of Ω_0, w is in V. Necessarily, w is the second write in V (that is $w = V : 3$). The regularity of the register *WR* implies that $w \not\vdash R : 4$. ∎

Lemma 3.7 *Suppose that V is a* Write *operation and R a* Read *operation such that $(V : 2) \prec R$. Then*

1. *$V \preceq \Omega_0(R)$, and if $F(R) \prec end(V)$ then $V = \Omega_0(R)$.*
2. *If $V = \Omega_0(R)$, then R is sensitive, and for every k $T = \Omega_k(R)$ is defined.*
3. *If $\Omega(R) \prec V$ then $\forall\ k$ $T = \Omega_k(R)$ is defined and $F(T) \prec end(V)$.*

Proof. If $(V : 2) \prec R$, then $(V : 2) \prec (R : 4)$, and thus $(V : 2) \preceq \omega(R : 4)$ by the regularity of the *WR* register. Thus $V \preceq \Omega_0(R)$ by definition of Ω_0 (and seriality of Writer). Now assume that $F(R) \prec end(V)$, and then $\omega(R : 4)$ is either $(V : 2)$ or $(V : 3)$ so that $V = \Omega_0(R)$ in this case. This proves (*1*).

To prove (*2*), assume that $(V : 2) \prec R$ and $V = \Omega_0(R)$. Then $\omega(R : 4)$ is in V, and hence $(R : 4) \not\vdash (V^+ : 2)$, and thus each of the first two reads of R (namely $R : 0$ and $R : 2$) is either determined by the writes in V or by $V^+ : 2$, but not by later writes. Thus ($R!sensitive$) holds because either $x_0.seq = y.seq$ or $x_0.oldseq = y.seq$, and either $y.alt = x_1.alt$ or $y.alt + 1 = x_1.alt \pmod 3$. (To see this, check the two possibilities for x_0: first that it is determined by the writes $(V : 2)$ or $(V : 3)$, and second by $(V^+ : 2)$. Similarly check the possibilities for x_1.)

Now we prove (*3*), and assume that $(V : 2) \prec R$ and $\Omega(R) \prec V$. By (*1*), $V \preceq \Omega_0(R)$. In fact, $V = \Omega_0(R)$, for otherwise $V \prec \Omega_0(R)$ and then $\Omega(R) \prec V \prec \Omega_0(R)$ would imply $\Omega(R)^+ \prec \Omega_0(R)$, and this contradicts $\Omega_0(R) \preceq \Omega(R)^+$. So $V = \Omega_0(R)$, and then (*2*) implies that R is *sensitive*. So for each k, $T = \Omega_k(R)$ is defined. $(T : 4) \prec (R : 4)$ by Lemma 3.5.

We will prove that $F(T) \prec end(V)$ by proving that $T : 4 \prec end(V)$ as follows. We claim that $(R : 5) \not\succ V$, and then Lemma 2.5 implies that $T : 4 \prec end((V))$. So assume on the contrary that $V \prec R : 5$. Then $V \prec R : 4$ follows. But now, as we have already shown that $V = \Omega_0(R)$, $\omega(R : 4) = (V; 3)$. As R is *sensitive*, this implies that $V = \Omega(R)$ in contradiction to our assumption. ∎

Lemma 3.8 *Suppose that V is a Write operation and R a Read operation such that $V \prec R$. Then $V \preceq \Omega(R)$.*

Proof. Since $V \prec R$ implies $(V : 2) \prec R$, Lemma 3.7(1) gives $V \preceq \Omega_0(R)$. If $V \prec \Omega_0(R)$, then $V \preceq \Omega(R)$ follows since either $\Omega(R) = \Omega_0(R)$ or $\Omega(R)^+ = \Omega_0(R)$. If $V = \Omega_0(R)$, then R is *sensitive* (by Lemma 3.7(2)) and $\omega(R : 4) = V : 3$, which implies that R is in *direct* mode. Thus $V = \Omega(R)$. ∎

Let us denote by R^+, for any Read i operation R, the successor Read i operation, that is $R \prec R^+$ and there is no Read i operation in between. Similarly, V^+ denotes the successor Write operation of V. Note that we are not assuming that there are infinitely many Write or Read operations and the successor of R needs not always exist. Our aim is to prove that if R is in the *flag* mode, then R cannot precede the uncertainty region of $\Omega_0(R)$, but we must first prove the following four lemmas.

Lemma 3.9 *Let R be a* Read *i operation, and V a* Write *operation. If $R = \Omega'_i(V)$ or $R^+ = \Omega'_i(V)$, and $V \sim_i \Omega_0(R)$, then R is not sensitive.*

Proof. Let $(R!y.seq[i]) = v$. To prove that R is not *sensitive* we will show that

$$v \notin (R!\{x_0.oldseq[i], x_0.seq[i]\}). \tag{6}$$

The definition of $\Omega_0(R)$ implies that

$$(R!y.seq) = (\Omega_0(R)!seq).$$

Since $V{\sim_i}\Omega_0(R)$, $(V!seq[i]) = (\Omega_0(R)!seq[i])$. So $v = (V!seq[i])$. Now $(V!seq[i])$ is determined at line 1 of V, and since any Read writes on its $RW[i]$ register the value of variable d,

$$v = (V!seq[i]) = (\Omega_i'(V)!d).$$

We consider two cases:
Case 1: $\Omega_i'(V) = R$. Then $v = (R!d)$ follows, and hence (6), because $R!d \notin \{x_0.oldseq[i], x_0.seq[i]\}$.
Case 2: $\Omega_i'(V) = R^+$. Then $v = (R^+!d)$. But $(R^+!d) \notin (R^+!r)$, and

$$R^+!r = R!\{x_0.oldseq[i], x_0.seq[i]\}.$$

Hence (6) follows. ∎

Lemma 3.10 *If* Read i *operation* S *is sensitive,* V *is a* Write *operation such that* $\Omega_0(S) \prec V$*, and* $\Omega_0(S){\sim_i}V$*, then*

$$S^{++} \;\; \textit{exists and} \;\; S^{++} \preceq \Omega_i'(V).$$

Proof. Set $V_0 = \Omega_0(S)$. Then $V_0 \prec V$ by assumption. Let us first rule out the possibility that $V_0^+ = V$. Indeed, this is so because the *alt* variable has different values in V_0 and V_0^+, and hence $V_0 \not\sim_i V_0^+$. Thus $V_0^+ \prec V$. From the definition of $V_0 = \Omega_0(S)$ it follows that $(S:4) \not\to V_0^+$. Let u be $S:1$, the write onto register $RW[i]$. Since $u \prec (S:4) \not\to V_0^+ \prec V$, $u \prec V$ follows. Thus

$$S \preceq \Omega_i'(V).$$

By the previous lemma neither $S = \Omega_i'(V)$ nor $S^+ = \Omega_i'(V)$ is possible, because S is *sensitive*, and hence $S^{++} \preceq \Omega_i'(V)$. ∎

Lemma 3.11 *If* R *is a sensitive* Read i *operation and* $V = \Omega_0(R)$*, then for every* k

$$\Omega_k'(V) \preceq (\Omega_k(R))^+.$$

Proof. Let $s = V : 2$ (the first write on *WR* in V), and $r = R : 2$ (the second read of *WR* in R, which determines the value of x_1). The main point is to prove that

$$s \not\succ r. \tag{7}$$

Because this implies that $V : 1.k \prec R : 3.k$ (by the Russell–Wiener property), and thence $\Omega'_k(V) \preceq \Omega_k(R)^+$ follows by contradicting the assumption $\Omega_k(R)^+ \prec \Omega'_k(V)$. Indeed, in such a case, $\Omega_k(R)^+ \prec \Omega'_k(V) \not\succ (V : 1.k) \prec R : 3.k$ would imply $\Omega_k(R) \prec \Omega_k(R)^+ \prec R : 3.k$, which is not possible as $\Omega_k(R)^+$ contains a write on register $RR[k]$.

Assume that (7) does not hold, so that $r \prec s$. Let V_0 be the Write operation containing $\omega(r)$. Then $V_0 \prec V$ (since $\omega(r) \prec s$ and s is the first write on *WR* in V). Observe first that $V_0^+ = V$ is not possible, because, in such a case, $V_0!alt + 1 = V!alt \pmod 3$ implies that $(R!x_1.alt) + 1 = (R!y.alt) \pmod 3$, and this would contradict the assumption that R is *sensitive*. Thus

$$V_0^+ \prec V.$$

But $r \not\succ V_0^+$ (since $V_0^+ \prec r$ would contradict the regularity of ω). Let $w = R : 1$ be the write onto $RW[i]$. So, since $w \prec r \not\succ V_0^+ \prec V$, $w \prec V$ follows. We are going to conclude that $R = \Omega'_i(V)$, and thence, by lemma 3.9 applied to $V = \Omega_0(R)$, that R is not *sensitive*, which brings the required contradiction.

Let $u = V : 1.i$ be the read of $RW[i]$ in V. We have seen that $w \prec V$, and thence $w \prec u$, and we will show that

$$u \prec end(R) \tag{8}$$

which imply that $w = \omega(u)$.

Precedence relation (8) follows from $u \prec s \not\succ R : 4$ and Lemma 2.5 ($s \not\succ R : 4$ is a consequence of the regularity of ω). ∎

Lemma 3.12 *If R is cued by $S = \Omega_k(R)$, then $\Omega_0(S) = \Omega_0(R)$.*

Proof. Observe first that $S = \Omega_k(R)$ implies that $(S : 4) \prec (R : 4)$ (Lemma 3.5). The regularity of Ω_0 and Lemma 2.3 imply that

$$\Omega_0(S) \preceq \Omega_0(R)^+. \tag{9}$$

Since R is cued by S, Lemma 3.3 shows that

$$S \text{ is in the } \textit{flag} \text{ mode, and } \Omega_0(S) \sim_k \Omega_0(R).$$

Since no Write is $\sim_k$ with its successor, $\Omega_0(S) = \Omega_0(R)^+$ is not possible. Hence, from (9) above,

$$\Omega_0(S) \preceq \Omega_0(R).$$

We want to prove equality, and assume towards a contradiction that

$$\Omega_0(S) \prec \Omega_0(R). \tag{10}$$

Since R is *cued*, R is sensitive, and Lemma 3.11 implies that, for $V = \Omega_0(R)$,

$$\Omega'_k(V) \preceq S^+. \tag{11}$$

Since S is *sensitive* and $\Omega_0(S) {\sim_k} \Omega_0(R)$, Lemma 3.10 and formula (10) imply that

$$S^{++} \preceq \Omega'_k(V).$$

Hence, from (11), $S^{++} \preceq S^+$, which contradicts $S^+ \prec S^{++}$. ∎

Now we can prove the main result.

Corollary 3.13 *If R is a* Read *operation in the flag mode, and $V = \Omega_0(R)$, then $(V : 3) \not\vdash (R : 4)$.*

Proof. Assume that R is in the *flag* mode, and then it is either in the *direct* mode or in the *cued* mode. In the first case, Lemma 3.6 implies the desired conclusion, that $(V : 3) \not\vdash (R : 4)$. So assume now that R is in the *cued* mode.

Claim 3.14 *There exists a finite sequence $R = R_0, R_1, \ldots, R_m$ of* Read *operations, such that R_i is cued by R_{i+1}, and the final Read, R_m, is in the direct mode.*

Proof. Define Read operations R_i in the *flag* mode by induction on i, starting with $R_0 = R$. If R_i is not in the *direct* mode, then it is in the *cued* mode, and then for some k, a Read k operation R_{i+1} can be found such that R_i is *cued* by $R_{i+1} = \Omega_k(R_i)$. Since R_{i+1} is in the *flag* mode, it is either in the *direct* mode (in which case we are done) or else in the *cued* mode, and then we continue. Since $(R_{i+1} : 4) \prec (R_i : 4)$ follows from Lemma 3.5, the sequence thus defined is necessarily finite. (A formal proof of this, in general system executions, would use Lamport finiteness property which implies in case this sequence is infinite that for some m, $R \prec R_m$, in contradiction to $R_m : 4 \prec R : 4$.) This proves the claim.

Now Lemma 3.12 implies that $V = \Omega_0(R_m) = \Omega_0(R_i)$ for all $0 \leq i \leq m$. The construction is such that $(R_{i+1} : 4) \prec (R_i : 4)$ for all i, and thence $(R_m : 4) \prec (R_0 : 4)$. Since R_m is in the *direct* mode, $(V : 3) \not\vdash (R_m : 4)$ (by Lemma 3.6), and thus $(V : 3) \not\vdash (R_0 : 4)$. ∎

The properties that we have proved in the lemmas above are rearranged in a 'specification list' which will be used to prove the atomicity of the register.

3.3 Higher level specifications

The following is a list of statements in a language involving the functions *value*, Ω, Ω_i (for $0 \leq i \leq M$), G, F, and the predicates Writer, Reader i, $1 \leq i \leq M$, and *flag*. In square brackets we may write comments.

S_0 The Writer operations are serial, and each Reader i is serial. Seriality of a process means here that $\prec$, restricted to the operations of the process, is a linear ordering, and also that an initial event preceding all non initial events exists in each process. We say 'Write' and 'Read' (i.e., Read i for some i) to emphasize that we refer to non initial events.

S_1

1. The function Ω_0 is defined on the set of Read operations, and it returns Writer operations. Ω_0 is regular.
2. For every $1 \leq k \leq M$, the function Ω_k is defined on a set of Read operations, and it returns Reader k operations. Ω_k is regular.
3. The function G is defined on the Write operations: For every Write operation W,
$$begin(W) \prec G(W) \prec end(W).$$
4. The function F is defined on the Read operations: For every Read operation R, $begin(R) \prec F(R) \prec end(R)$, and $\Omega_0(R) \not\vdash F(R)$.

[For simplicity of expression, we may write $G(X)$ and $F(X)$ also for an initializing event X; in this case we mean $G(X) = X$, $F(X) = X$.]

S_2 The return function Ω gives for every Read operation R a Writer operation $\Omega(R)$ such that: (1) The value returned by R is the value written by $\Omega(R)$. (2) For every Write operation V, if $G(V) \prec R$, then $V \preceq \Omega(R)$ [Lemma 3.8]. (3) If R is not in *flag* mode, then there is no Writer event W such that $\Omega(R) \prec W \prec \Omega_0(R)$ (but $\Omega(R)^+ = \Omega_0(R)$ is not excluded).

S_3 If Read operation R is in *flag* mode, then the following hold for $V = \Omega_0(R)$:

1. $\Omega(R) \preceq V$.
2. $G(V) \not\succ R$. [If V is the initial Writer event, then $V \prec R$ holds.]

[Corollary 3.13]

S_4 If Read operation R is not in the *flag* mode, then $G(\Omega(R)) \prec \mathit{end}(R)$. [See Lemma 3.4]

S_5 If $R_2 = \Omega_k(R_1)$ and R_2 is in *flag* mode, where R_1 and R_2 are Read operations such that $\Omega_0(R_2) = \Omega_0(R_1)$, then R_1 is in *flag* mode. [See Lemma 3.3.]

S_6 Suppose that V and S are a Write and Read operations such that $\mathit{begin}(G(V)) \prec S$, then the following hold:

1. $V \preceq \Omega_0(S)$, and if S is in *flag* mode then $V \preceq \Omega(S)$.
2. If $\Omega(S) \prec V$, then for every $1 \leq k \leq M$, $T = \Omega_k(S)$ is defined and $F(T) \prec \mathit{end}(V)$. [See Lemma 3.7(3).]

S_7 Let V be a Write and T a Read operation such that $\mathit{begin}(G(V)) \prec T$ and $F(T) \prec \mathit{end}(V)$. If T is not in *flag* mode, then for every k, $S = \Omega_k(T)$ is defined and $F(S) \prec \mathit{end}(F(T))$. [Lemma 3.7(1)(2) and Lemma 3.5.]

The following section is the higher-level part of the proof where the atomicity of Ω is derived from the specifications. There are different possible choices for a specification list as above that hold for the protocol on one hand and imply atomicity on the other. Our choice is motivated by the desire to find a single list that will also handle the second protocol of Section 5. The following remarks exemplify how this has shaped the list. Corollary 3.13 concludes that $(V:3) \not\succ (R:4)$. But in a general specification list we cannot refer to the third line of a protocol, and thus we formulate in $S_3(2)$ a weaker property, namely that $G(V) \not\succ R$. Similarly, in S_6, the condition on V and S is that $\mathit{begin}(G(V)) \prec S$, while Lemma 3.7 is stronger—it only requires

$V:2 \prec S$. To give another example, in $S_2(3)$ we stated a weaker property than $\Omega(R) \preceq \Omega_0(R) \preceq \Omega(R)^+$ because in the second protocol $\Omega_0(R) \prec \Omega(R)$ is possible.

4 The specifications imply atomicity

Theorem 4.1 *In any system-execution, $\mathcal{H}$, if there are functions and relations that satisfy the specification list of the last section, then the register implemented through the return function Ω is atomic.*

By atomicity we mean the following three properties, similar to those in [SAG] (which follow Lamport [1986]).

Integrity For each Read operation R in $\mathcal{H}$, the value returned by R is the value written by the Write operation $\Omega(R)$.

Proximity The return function Ω is regular.

Precedence For any two Read operations R and S, if $R \prec S$, then $\Omega(R) \preceq \Omega(S)$.

These three properties and the specifications involve some informal English, but it is clear that they can be written in first-order logic. We will give enough details to show that the derivation of atomicity from the specifications can be carried in first-order logic if the following axioms are assumed.

A_1 The relation $\prec$ is an irreflexive partial ordering that satisfies the Russell-Wiener property. $A \preceq B$ is a shorthand for $A \prec B \vee A = B$. $A \not\succ B$ is the negation of $B \prec A$.

A_2 Functions *end* and *begin* are partial functions. $A \prec \mathit{end}(B) \longrightarrow \forall X(B \prec X \longrightarrow A \prec X)$. Similarly, $\mathit{begin}(A) \prec B$ implies that for all X, if $X \prec A$ then $X \prec B$. If $A \not\succ B \prec C$ then $\mathit{begin}(A) \prec C$. If $A \prec B \not\succ C$, then $A \prec \mathit{end}(C)$. (See Lemma 2.5.)

A_3 Since Lamport finiteness property is not a first-order statement, and as we need it, the following 'minimality axiom' is assumed for a certain formula $\phi(T)$ described later.

$$\exists T \phi(T) \longrightarrow \exists T[\phi(T) \wedge \forall S\ (F(S) \prec \mathit{end}(F(T)) \longrightarrow \neg\phi(S))].$$

One way to see that this axiom holds in any execution, is to recall that $end((X))$ refers to the last lower-level event in X, and these events are assumed to be serially ordered in order-type of the natural numbers in the history. A more abstract alternative is to use Lamport finiteness property to refute the existence of an infinite sequence of Read events T_i such that $F(T_{i+1}) \prec end(F(T_i))$.

Lemma 4.2 *If V is a* Write *and R a* Read *such that $begin(G(V)) \prec R$ and $F(R) \prec end(V)$, then $V = \Omega_0(R)$.*

Proof. $S_6(1)$ implies that $V \preceq \Omega_0(R)$, and if equality does not hold then $V \prec \Omega_0(R)$. But then $F(R) \prec \Omega_0(R)$ (follows from $F(R) \prec end(V)$) and this contradicts $S_1(4)$. ∎

Lemma 4.3 *Suppose that* Read *R is in flag mode and $V = \Omega_0(R)$. Let T be a* Read *such that $R \prec T$ and $F(T) \prec end(V)$. Then $V = \Omega_0(T)$ and T is also in flag mode.*

Proof. Assume that R in *flag* mode, $V = \Omega_0(R)$, and T are as in the lemma. We first prove that $V = \Omega_0(T)$. Since $V = \Omega_0(R)$, $G(V) \not\twoheadrightarrow R$ by $S_3(2)$. Hence

$$begin(G(V)) \prec T$$

follows from $R \prec T$. Now the previous lemma implies that

$$V = \Omega_0(T).$$

Now we prove that T is in *flag* mode. Let $\phi(T)$ be the statement:

> *T is a Read operation such that $R \prec T$ and $F(T) \prec end(V)$, but T is not in flag mode.*

We want to prove that there is no T such that $\phi(T)$, and assume otherwise for a contradiction. Then (by the minimality axiom) for some T $\phi(T)$ holds but for every Read operation S such that $F(S) \prec end(F(T))$, $\phi(S)$ no longer holds.

Since $\phi(T)$ holds, T is not in *flag* mode. We have argued that $R \prec T$ implies $begin(G(V)) \prec T$, and so Lemma 4.2 implies that $V = \Omega_0(T)$. Also, S_7 can be applied, and if k is the index of R, then $S = \Omega_k(T)$ is defined and

$$F(S) \prec end(F(T)).$$

Since $R \prec T$, $R \preceq S$ (by the regularity of Ω_k and seriality of Reader k). We claim that

$$V = \Omega_0(S), \text{ and } S \text{ is in } \textit{flag} \text{ mode.}$$

Once this is proved, then S_5 implies that T is in *flag* mode too, which is a contradiction. The claim is obvious if $S = R$, because $V = \Omega_0(R)$, and R is in *flag* mode.

So assume $R \prec S$. Then $begin(G(V)) \prec S$. Since $F(S) \prec end(F(T))$, $F(S) \prec end(V)$ follows from $F(T) \prec end(V)$. Thus $V = \Omega_0(S)$ by Lemma 4.2. Since $F(S) \prec end(F(T))$, $\phi(S)$ does not hold. Yet $R \prec S$, and $F(S) \prec end(V)$. So S is in *flag* mode (or else $\phi(S)$ holds)! ∎

Lemma 4.4 Ω *is regular.*

Proof. For every Read operation R, if $W = \Omega(R)$ then

$$W \not\succ R$$

follows from S_1, S_3, and S_4: In case R is in *flag* mode, then $W \preceq \Omega_0(R)$ (by $S_3(1)$), and then the regularity of Ω_0 implies that $\Omega_0(R) \not\succ R$ and hence $W \not\succ R$. In case R is not in the *flag* mode, then $G(W) \prec end(R)$ follows from S_4, and implies that $W \not\succ R$, because otherwise $R \prec W$ implies $G(W) \prec W$, and then $begin(W) \prec G(W)$ implies $G(W) \prec G(W)$! (Use A_2.)

Now the existence of a Writer operation V such that $\Omega(R) \prec V \prec R$ is ruled out because if $V \prec R$, then $G(V) \prec R$ follows from $G(V) \prec end(V)$, and then $V \preceq \Omega(R)$ by $S_2(2)$.

Lemma 4.5 *If* $R \prec S$ *are* Read *operations, then* $\Omega(R) \preceq \Omega(S)$.

Proof. The proof is split in two cases, according as to whether R is or is not in the *flag* mode.

Case 1: R is not in the *flag* mode. Then $G(\Omega(R)) \prec end(R)$ by S_4. So $G(\Omega(R)) \prec S$ (since $R \prec S$). Hence $\Omega(R) \preceq \Omega(S)$ follows from $S_2(2)$.

Case 2: R is in the *flag* mode. Let $V = \Omega_0(R)$. The following is a consequence of S_3 and Lemma 4.3:

1. $\Omega(R) \preceq V$.
2. $G(V) \not\succ R$.

3. If $R \preceq T$, where T is a Read operation such that $F(T) \prec end(V)$ then $V = \Omega_0(T)$ and T is also in *flag* mode.

The seriality of Writer implies that either $V \preceq \Omega(S)$ or else $\Omega(S) \prec V$. In the first case, item 1 above implies that $\Omega(R) \preceq \Omega(S)$ and then the lemma is proved. So assume that

$$\Omega(S) \prec V.$$

Since $R \prec S$,

$$begin(G(V)) \prec S$$

follows from item 2 and axiom A_2. Let k be the index of R (so that R is a Read k operation). The last displayed formula show that S_6 can be applied, and so

- $V \preceq \Omega_0(S)$ and if S is in *flag* mode then $V \preceq \Omega(S)$. Hence, by our assumption, S is not in *flag* mode. Thus $V = \Omega_0(S)$, or otherwise $\Omega(S) \prec V \prec \Omega_0(S)$ contradicts $S_2(3)$.
- $T = \Omega_k(S)$ is defined and $F(T) \prec end(V)$.

Since $R \prec S$, $R \preceq T = \Omega_k(S)$ by the regularity of Ω_k and seriality of Reader k. Item 3 above says that $V = \Omega_0(T)$, and T is in *flag* mode (since $F(T) \prec end(V)$). But then S_5 and $V = \Omega_0(S)$ imply that S *is* in the *flag* mode!

5 A two buffer protocol

Motivated by the protocol of Haldar and Vidyasankar [1991], we present in this section a protocol for implementing an atomic single-writer multi-reader register from single-writer, single-reader regular registers and two buffers. A buffer is a register that carries only data values. There is some elegance in using buffers, because of the strict separation between the data and the coordination parts. It seems that it is harder to devise buffer protocols because one cannot rely on tags to identify the data. Yet, Vidyasankar [1991] and Haldar and Vidyasankar [1991] have shown that buffer protocols can be simple and efficient. The structure of our protocol resembles that of Haldar and Vidyasankar [1991], it contains less operations, but its registers are much heavier.

The registers used are the following:

1. Register WR is regular, written by Writer and read by Reader k. It carries a pair $(colour, phase)$, where $colour$ is in **colour**$= [0, \ldots, 2M+1]$, and $phase$ is in $\{1, 2\}$.

2. The registers $RW[k]$ are regular, written by Reader k and read by the Writer. Their values are unordered pairs or singletons from **colour**.

3. Registers $RR[k]$ are regular, written by Reader k and read by all the Readers. They carry either the special "masking" value $*$, or else a **colour**. The initial value of these registers is $*$.

4. The buffers a and b are regular, written by Writer and read by the Readers. They carry messages of *valtype*.

program Writer(*val*: *valtype*)
begin

1: **forall** $(1 \leq k \leq M)$ **concurrently do read** $r[k] := RW[k]$ **od**;
find $d \in$ **colour** $\setminus \{d\} \cup \bigcup\{r[k] \mid 1 \leq k \leq M\}$ (* so d is a new value in $[0, \ldots, 2M+1]$ not in any $r[k]$ *)

2: **write** $WR := (d, 1)$;

3: **write** $a := val$;

4: **write** $WR := (d, 2)$;

5: **write** $b := val$;
end.

---

program Reader$(i:\ 1..M)$ returns *valtype*;
begin

1: **read** $v := a$;

2: **read** $x := WR$;

3: **write** $RW[i] := \{x.colour, oldx\}$; $oldx := x.colour$;

4: **read** $y := WR$;

5: **if** $x.colour \neq y.colour$ **then** *flag*:= **false**
elseif $y.phase = 2$ **then** *flag* := **true**; **else**

1. (* Now we are in case when $y.phase = 1$ and $x.colour = y.colour$, and we will say that R is *sensitive* in such a case *)
   forall $(1 \leq k \leq M)$ **concurrently do read** $s[k] := RR[k]$;
2. **if** $\exists k\ s[k] = y.colour$ **then** *flag*:= **true else** *flag*:= **false**;

6: **if not** *flag* **then read** $v := b$ **fi**;

7: **if** *flag* **then write** $RR[i] := y.colour$ **else write**$RR[i] := *$ **fi**;
return v ;
end.

The G part of a Write operation W consists of the second write on WR. That is $G(W)$ is the higher-level event that contains the write $W : 4$. The F part of a Read operation R consists of the execution of the first 6 lines. That is, $F(R) = \{(R : 1), \ldots, (R : 6)\}$. As before, we define higher-level relations and functions and prove some basic properties which show that the specification list holds for any execution, and thus that the protocol implements an atomic register.

5.1 Higher-level relations and properties

Suppose a history H and the resulting higher-level view system execution. The regular function ω gives, for any read r of a regular register or of a buffer, a write event $\omega(r)$ that determines r.

Definition 5.1 *1. For any Read operation R we say that R is sensitive iff*

$$(R!y.colour = x.colour) \text{ and } (R!y.phase = 1).$$

If $(R!y.colour = x.colour)$ but $(R!y.phase = 2)$, then R is said to be in the direct mode.

If R is sensitive then R contains M (concurrent) reads of registers $RR[k]$, and if for some k

$$R!s[k] = y.colour \tag{12}$$

then R is said to be cued. If R is in the direct mode or in the cued mode then R is said to be in the flag mode (then $R!flag$ holds). (Remark that when R is in the flag mode it is the value read in buffer a that is returned; when R is not in flag mode, the value of buffer b is returned)

2. *The function Ω_0 is defined on Read operations R as follows: $\Omega_0(R)$ is the Write operation containing $\omega(R:4)$.*

   The function Ω_k is defined only on Reads R that are sensitive, and then $\Omega_k(R)$ is the Reader k operation containing $\omega(R:5(1).k)$. If equation (12) holds and $S = \Omega_k(R)$, then we say that R is cued by S.

3. *The definition of the return function $\Omega(R)$ is done accordingly as to which buffer is returned by R. In case R returns buffer a, let $\Omega(R)$ be the Write operation containing $\omega(R:1)$. If R returns buffer b, then $\Omega(R)$ is defined accordingly to be the Write operation containing $\omega(R:6)$.*

   In any case, it is obvious that if W is a Write operation such that $W \prec R$ then $W \preceq \Omega(R)$ (because W finished writing the two buffers before R began its execution). Similarly, if $(W:3) \prec R$ and R is in flag mode, then $W \preceq \Omega(R)$ follows.

The proof of the higher-level specification list will be somewhat less formal and easier parts will be left to the readers. We only prove $S_2(3)$, S_3, and S_6 which are more delicate.

Lemma 5.2 *If* Read *R is not in flag mode, then there is no* Writer *event W such that $\Omega(R) \prec W \prec \Omega_0(R)$ (this is $S_2(3)$).*

Proof. Assume that for some Writerevent W, $W \prec \Omega_0(R) = V$ holds. Then $\omega(R:4)$ is in V by definition of Ω_0. So $W \prec V \not\succ (R:4) \prec (R:6)$, and hence $W \prec (R:6)$. Since $(R:6)$ is the read of buffer b and R is not in *flag* mode, $W \preceq \Omega(R)$ as required. ∎

Lemma 5.3 *If Read R is in flag mode, then $\Omega(R) \preceq \Omega_0(R)$.*

Proof. Let $W = \Omega(R)$. Then $(W:3) \not\succ (R:1)$ (for otherwise, the read in R of buffer a precedes the write in W). Thus $(W:2) \prec (W:3) \not\succ (R:1) \prec (R:4)$, and $(W:2) \prec (W:4)$ follows. Hence $W \preceq \Omega_0(R)$. ∎

Lemma 5.4 *S_6 holds.*

Proof. Assume $begin(G(V)) \prec S$. Then $(V:2) \prec S$. Hence $V \preceq \Omega_0(S)$ and if S is in *flag* mode then $V \preceq \Omega(S)$ as well.

To prove $S_6(2)$, assume that $begin(G(V)) \prec S$ and $\Omega(S) \prec V$. Then S is not in *flag* mode by the first part of our lemma. Thus S returns the value of buffer b. Hence $(S:6) \not\vdash (V:5)$ (or else in reading buffer b, S gets the value of V or a later value). This implies that the reads $(S:2)$ and $(S:4)$ get the values written by V, and hence S is *sensitive*. Thus $T = \Omega_k(S)$ is defined.

$F(T) \prec (S:6)$ now follows from $F(T) \prec (T:7) \not\vdash (S:5) \prec (S:6)$. But $(S:6) \not\vdash (V:5)$, and thus $F(T) \prec end(V:5)$ (by Lemma 2.5). ∎

We now turn to a lemma that leads to $S_3(2)$.

Lemma 5.5 *If R and S are Read operations, and R is cued by $S = \Omega_k(R)$, then*

1. $F(S) \prec end(F(R))$.
2. $\Omega_0(R) \preceq \Omega_0(S)$.

Proof. Item 1 is easy to derive from $(S:7) \not\vdash (R:5)$. To prove 2, assume on the contrary that $V = \Omega_0(S) \prec \Omega_0(R) = W$. Since R is *cued* by S, S is in *flag* mode and

$$R!y.colour = S!y.colour. \tag{13}$$

Since $(V!d) \neq (V^+!d)$, $V^+ = W$ is not possible, and hence $V^+ \prec W$. Since $W = \Omega_0(R)$, $\omega(R:4)$ is either $W:2$ or $W:4$. Hence

$$W:2 \not\vdash R:4.$$

Now let $u = S:2$ be the write onto $RW[k]$, Then $u \prec S:4 \not\vdash V^+ \prec W$. (The relation $S:4 \not\vdash V^+$ follows from $V = \Omega_0(S)$ and the definition of Ω_0.) Thus $u \prec W$. Let $r = W:1.k$ be the read of $RW[k]$. Then

$$u \prec r.$$

On the other hand, $r \prec (W:2) \not\vdash (R:4) \prec end(S^+)$ (if S^+ exists) and so $r \prec end(S^+)$ (the reason $R:4 \prec end(S^+)$ is that, otherwise, $S^+ \prec R:5$ and then $S^+ \preceq \Omega_k(R)$ follows). Thus, $\omega(r)$ is either u or u^+, where $u^+ = (S^+:3)$. In both cases, the value of $\omega(r)$ contains $S!x.colour$, because of the way the writes at $S:3$ and $S^+:3$ are determined (namely the usage of *oldx*). The value of $\omega(r)$ is $(W!r[k])$ which does not contain $(W!d)$, the colour written in $W:2$ and $W:4$. Thus $(S!x.colour) \neq (W!d)$. But $(W!d) = (R!y.colour)$, and $(S!x.colour = y.colour)$, because S is in *flag* mode. Hence $R!y.colour \neq S!y.colour$, in contradiction to equation (13) above. ∎

Corollary 5.6 *If Read operation R is in the flag mode, and $V = \Omega_0(R)$, then $G(V) \not\vdash R$.*

Proof. If R is in *direct* mode (and then $R!y.phase = 2$), then $\omega(R:4)$ is necessarily the second write onto *WR* in V (that is $V:4$) and hence $G(V) \not\vdash R$.

If R is not in the direct mode, then R is *cued* by some $\Omega_k(R)$, and we may define a sequence of Read operations, $R = R_0, \ldots, R_m$ such that R_i is *cued* by R_{i+1}, and R_m is in the *direct* mode. We have proved in the last lemma that

- $F(R_{i+1}) \prec end(F(R_i))$, for all i, and hence $F(R_m) \prec end(F(R))$.
- $\Omega_0(R_i) \preceq \Omega_0(R_{i+1})$ for all i.

(Lamport finiteness property may be used to prove that there is no infinite sequence such that R_i is *cued* by R_{i+1}.) So

$$V = \Omega_0(R_0) \preceq \Omega_0(R_m).$$

But since R_m is in the *direct* mode, $G(\Omega_0(R_m)) \not\vdash F(R_m)$ (by the first case considered in the proof). So $G(V) \preceq G(\Omega_0(R_m)) \not\vdash F(R_m) \prec end(F(R))$. Hence $G(V) \not\vdash F(R)$. ∎

6 Conclusion

The problem of implementing multiple-reader atomic registers from simpler registers has attracted many writers, and the reader is referred to the recently published work [SAG] of Singh, Anderson, and Gouda for references and history of the problem. We will add to this reference list only the following two items: Vidyasankar [1991], and Haldar and Vidyasankar [1991].

construction	type of registers	max value of registers	time complexity of reads	time complexity of writes
SAG	atomic, mixed	$3^M \cdot 4$	$M+3$	$3M$
S3	regular, mixed	$5^{2M} \cdot 6$	6	3
HV1	regular, and buffers	3	8	5
S5	regular, and buffers	order M^2	5	3

Table 1: M is the number of Readers. Only operations on coordinating registers are counted.

Vidyasankar [1991] gives an amazingly simple protocol which uses a single-writer, multiple-reader *atomic boolean* register and regular buffers to obtain an atomic multiple-reader register. The protocol HV1 of Haldar and Vidyasankar [1991] replaces the atomic register of Vidyasankar [1991] with regular registers (using an ingenious forwarding technique), but tries to retain the same basic ideas and structure.

The two protocols presented in this paper implement single-writer, multiple-reader atomic registers from regular registers. The first protocol uses mixed registers (which carry both coordinating and data values) and the second uses control registers and buffers. Our first protocol (S3 in Table 1) is a variant of the protocol of [SAG] (SAG in that table), and the second (S5) is a variant of the protocol of Haldar and Vidyasankar [1991] (HV1 in the table). The advantage of our protocols is their parallel-time efficiency. The parallel-time measure of a protocol is such that concurrent executions of read or write statements are counted as one.

Having a better parallel time measure is of theoretical and practical interest, but it must not be taken as an efficiency measure of the protocol since neither the size nor the type (atomic, regular, or safe) of the register is taken into account. It is clear that if all registers are implemented with safe boolean registers, then HV1 is better than the other protocols.

Despite their differences, the two protocol S3 and S5 share a characteristic "forwarding" approach, which is not so easy to describe formally. It is not even clear what should be the format of such a description. The answer attempted in this paper is to describe a list of higher-level specifications that both protocols satisfy. The functions Ω_k for example reflect the idea of Readers giving a cue which can cause some Reads to be in flag mode. It remains to be seen in which class the other protocols fall, and whether a natural classification of atomic register protocols can be obtained.

The proof of [SAG] is a paradigm for a certain approach to correctness proofs of protocols, and the main motive of our paper is to suggest a different approach, one which can deal with regular registers and concurrent executions naturally. Relying on and expanding the concept of system execution (Lamport [1986]), this approach has a distinctive character in the separation of the proof into a higher–level and lower–level parts. The interested reader can find in the references more examples of our approach and a deeper analysis of system executions and their relations with "global-time" and the notion of a state.

Acknowledgement: Many thanks to S. Haldar and K. Vidyasankar who proved that the help and sharp eyes of colleagues are still the best verification methods. The idea to investigate parallel-time measures of protocols was suggested to me by J. Tromp. Vidyasankar improved an earlier version of the protocol of Section 5.

References

Abraham, U. 1994. On interprocess communication and the implementation of multi-writer atomic registers. 1994 preprint, to be published by *Theoretical Computer Science.*

Abraham, U. 1993. What is a State of a System (an outline). In M. Droste and Y. Gurevich eds, *Semantics of Programming Languages and Model Theory*, Volume 5 of Algebra, Logic and Applications, pp. 213–243, Gordon and Breach.

Abraham, U. 1993. On system executions and states. *J. Applied Intelligence* 3, pages 311–323.

Abraham, U. 1994. Bakery Algorithms. In H-D. Burkhard, L. Czaja, and P. Starke, eds, *Proceedings of the Concurrency, Specification and Programming Workshop, Nieborow 1993*, pp. 7–40, Wydawnictwa Uniwersytetu Warszawskiego, Warszawa.

Abraham, U., Ben-David, S., and Magidor, M. 1990. On Global-Time and Inter-process Communication. In *Semantics for Concurrency, Proceed-*

ings of the International BCS-FACS Workshop, Leicester (edited by Kwiatkowska et al, Springer-Verlag in collaboration with the British Computer Society), pp. 311 – 323.

Ben-David, S. 1988. The global-time assumption and semantics for distributed systems. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing*, pp. 223 – 232.

Fishburn, P. C. 1986. *Interval orders and interval graphs.* Wiley (Wiley-Interscience series in discrete mathematics).

Haldar, S., and Vidyasankar, K. 1991. A simple construction of 1-writer multireader multivalued atomic variable from regular variables. Tech. Report #9108. Memorial Univ. Newfoundland, St. John's, NF, Canada. To be published in JACM as "Constructing 1-Writer Multireader Multivalued Atomic Variables from Regular Variables".

Lamport, L. 1986. On Interprocess Communication, Part I: Basic formalism, Part II: Algorithms. *Dist. Comput.*, *1*, 77 – 101.

Singh, A. K., Anderson, J. H., and Gouda, M. G. 1994. The elusive atomic register. *J. ACM, 41*, 311–339.

Vidyasankar, K. 1991. A very simple construction of 1-writer multireader multivalued atomic variable. *Inf. Proc. Lett., 37*, 323–326.

Appendix: Making the proof formal

The proofs in Section 3.2, in which the list of basic properties is established, are conducted in the traditional mathematical style, rather than in strict first order theory. My aim here is to explicate the language and axioms that are needed in order to relate the protocol to system executions and to transform these proofs into formal first-order proofs.

A second aim of this note is to show how state variables can define regular registers.

I have no intention to incorporate this appendix in the paper since it is rather ad hoc; I am far from giving any general approach to semantics. My style is that of a lecture rather than a textbook, so instead of first presenting the language and then the axioms, I will combine the two in order to facilitate understanding of my motives. The protocol is related to the events in the system execution in a direct way: Each command is connected with an event in every execution of the protocol, each event is an execution of some command, and the axioms describe the semantics of the execution.

A_1 The language includes the precedence relation $\prec$, and the axioms state that this is an irreflexive transitive ordering that satisfy the Russell–Wiener axiom. Since Lamport's finiteness property speaks about "finiteness" which is not a first-order property, we rather have an axiom schema: For every statement ϕ there is an axiom saying that if $\exists x \phi(x)$, then for some x it is the case that $\phi(x)$ holds but for all $x' \prec x \; \neg\phi(x')$. The axioms in A_1 are called the "precedence" axioms.

A_2 We think about and want to formalize system executions with both higher-level and lower-level events. We note that higher-level events can themselves be grouped into even higher events. The relation $In(x, X)$ indicates that the higher-level event X contains x (which may itself consist of further lower-level events). We say that X is a 'higher-level' event in case for some x, $In(x, X)$. An axiom says for all higher-level X and event Y that $X \prec Y$ *iff* $\forall\, x(In(x, X) \to x \prec Y)$. A similar axiom says that $X \prec Y$ iff $\forall y(In(y, Y) \to X \prec y)$. These are the "higher-level precedence" axioms.

A_3 Events of lowest level are communication read/write events on registers. For each register R the predicates read_R and write_R are in the language, as well as the function symbols ω and *value*. (The intuitive meaning of $\mathrm{read}_R(r)$ is that r is a read event, $value(r)$ is then the value returned by r, while $\omega(r)$ is the write on R event that determined r.) Of course this is formally specified by the axioms, and there is an axiom saying that if the writes on R are serially ordered, then ω is a regular return function (see Definition 2.1). This group is called "register axioms".

Just to give an example, the following in one of the register axioms:

$$\forall r[\mathrm{read}_R(r) \to \mathrm{write}_R(\omega(r)) \wedge value(r) = value(\omega(r))].$$

A_4 We assume predicate "Writer" and M predicates "Reader$_i$" for $1 \leq i \leq M$. Here M is not a variable, but a fixed number. When we say "for some i, $1 \leq i \leq M \ldots$, this is not a quantification but rather a disjunction of M sentences. Reader(R) means that $\bigvee_i$ Reader$_i(R)$. Write(W) means that W is an execution of the Write protocol (Writer(W) also holds for the first Writer event, the initialization).

The fact that our protocols contain no loops or recursive calls allows a very simple semantics of ordering of execution events. For every type of operation (such as Read/Write execution of the protocol) there is a set of "locations". Usually, these are the line numbers of the protocol, but in cases when two instructions are written on the same line we add indices. For example, location 2_2 in the Write protocol refers to the assignment instruction $oldseq := seq$. There is a fixed partial ordering on these locations which indicates the ordering of executions. For example, line 1 is executed before line 2, and this gives the following axiom (a "seriality axiom"):

$$\forall R(\mathrm{Read}(R) \rightarrow (R:1) \prec (R:2)).$$

This simple minded approach suffices here. Formally, $(\ :1)$ is a function, and $(R:1)$ is its value at R. The lower-level events are described by axioms such as:

$$\forall R(\mathrm{Read}_i(R) \rightarrow [In((R:1),R) \wedge \mathrm{write}_{RW[i]}(R:1)]).$$

In general, for each type of operation and location, l, there is an axiom which says that if X is an operation of the appropriate type, then $(X:l)$ is in X of a certain character (read, write, assignment etc.).

If x is any lower-level event, then for some higher-level Reader of Writer Event X, $In(x,X)$. That is, there are no lower-level events except as those resulting from the protocol's executions (or the initialization events). The axioms should tell us whether X is a Reader or Writer Event, depending on the character of x. For example, $\mathrm{write}_{WR}(r) \rightarrow \exists W(\mathrm{Writer}(W) \wedge In(w,W)$.

If Write(W) and $In(x,W)$, then x is either the start or the conclude event in W or else x has the form $(W:l)$ where l is the location index. Similar axioms hold for Read Events. The axioms should locate the lower-level events accurately. For example,

$$\mathrm{Write}(W) \wedge In(x,W) \wedge \mathrm{write}_{WR}(x) \rightarrow x=(W:2) \vee x=(W:3).$$

A_5 For each instruction there is a set of variables such that it is meaningful to ask for the values of these variables after the execution of the instruction. For example, after execution of a single read of $RW[k]$ (in line 1 of the Writer protocol) only the value of $seq[k]$ is determined and it is meaningless to ask for other values in this array.

For each Event X and location l, if v is a meaningful variable for l, then $(X : l)!v$ denotes the value of this variable (after the execution of $(X : l)$. That is, ! is a function symbol in our language; it relates events and variables with values. For example, if W is a Write event, then $(W : 1)!seq$ denotes the (new) value of seq as determined by the execution $(W : 1)$. An axiom states that this value is determined by the values $(W : 1.k)!seq[k]$, and another axiom states that $(W : 1.k)!seq[k] = value(W : 1.k)$ (or in words, the value of $seq[k]$ in W is the value returned by the read of register $RW[k]$).

In general, if event r is an execution of an instruction **read** $x := R$, where x is some local variable and R is a register, then $r!x = value(r)$. (The value of $value(r)$ itself is determined by the register axioms).

If st is the *start* event of W, for example, then $(st!alt)$ is the value determined by the previous Writer event. Similar axioms hold for all other variables that keep their values from one operation to the following one.

A_6 Executions of lower-level **write** instructions do not change any local variables. That is, if w is any write event and it is the successor of event e in any higher-level operation X, then $w!x = e!x$ for any variable (meaningful for w or e). Thus, for example, If Write(W) then $(W : 2_1)!x = (W : 1)!x$ for any variable x.

Write instructions however determine the value of the write event. If w and e are as above (both belong to some higher-level operation X and w follows immediately e in X), and if $w = (X : l)$ is an execution of **write**$R := x$, then $value(w) = (X : l)!x = e!x$. To put it simply in words, the values of write events is the value of the appropriate variables at the preceding event. The write instructions at line 2 and 3 of the Writer protocol contain constants (**true** and **false**) and their semantics is obviously modified. For example, for a Write event W, if $w = W : 3$ then $value(w) = (W : 3)!(old, new, oldseq, seq, alt, \mathbf{true})$. The axioms tell us, for example, that $(W:3)!oldseq = (W:2_2)!oldseq = (W:2_1)!seq$ (because of the assignment instruction). Then the axioms can be applied to conclude that $(W:2_1)!seq = (W:1)!seq$ (writes do not change values of variables). Finally, the value of the array *seq* is determined by the axiom stating that $(W:1)!seq[k] = (W:1.k)!seq[k] = value(W:1.k)$. In the paper itself we omit all this chain of arguments and simply say that the value written by $(W:3)$ in field $oldseq[k]$ is the value of the read event $(W:1.k)$, namely the return value of the read of register $RW[k]$ in W.

A_7 The seriality axioms, and the assumption that each process is serial, imply that for each register the writes on that register are serially ordered.

I had no intention of giving a complete list of axioms here, just to exemplify what I intend to put in such axioms. Clearly, such a list can have a value only in a general framework which I don't claim to provide. While the register axioms reflect the fact that we deal with concurrency, all other axioms are the same as for sequential processes.

As an example, let us see how the first lemma, Lemma 3.5, can be stated and proved formally.

The lemma's assumptions are:

1. R_1 and R_2 are Read operations. This means that for some i and j, $\text{Read}_i(R_1)$ and $\text{Read}_j(R_2)$. (So these are not the initial events.) The axioms describing Read executions can be applied.

2. $R_2 = \Omega_k(R_1)$. Going back to the definition of Ω_k, this assumption means that $In(\omega(R_1:3.k), R_2)$. (That is, if w is the read event of register $RR[k]$ executed for line 3, then the corresponding write $\omega(r)$ is in R_2.)

The lemma's conclusion is that $(R_2:4) \prec (R_1:4)$

Since $r = (R_1:3.k)$ is a read of register $RR[k]$, $\omega(r) = w$ is a write on this same register. Hence, as $In(w, R_2)$ is assumed, $w = (R_2:5)$ (the axioms say that this is the only write on this register). Now,

$$w \not\succ\!\!- r \tag{14}$$

follows from the regularity of ω. The seriality axioms (as in A_3) imply that

$$(R_2 : 4) \prec (R_2 : 5), \ (R_1 : 3.k) \prec (R_1 : 4). \tag{15}$$

Gathering what we proved so far

$$(R_2 : 4) \prec (R_2 : 5) \not\succ (R_1 : 3.k) \prec (R_1 : 4)$$

and applying the Russell–Wiener axiom, $(R : 4) \prec (R_1 : 4)$ is concluded as required.

This example was particularly simple, because it didn't involve the values of the read/write events. Let us give another example, a proof of the main claim in Lemma 3.7(2).

Lemma 6.1 *Suppose that* Write(V)*,* Read$_i$(R)*,* $(V : 2) \prec R$*, and* $V = \Omega_0(R)$*. Then* R *is sensitive.*

Proof. By definition, $V = \Omega_0(R)$ means that $In(\omega(R : 4), V)$. Put $r = (R : 4)$, and $w = \omega(r)$. It follows from the axioms that $\mathrm{read}_{WR}(r)$, and $\mathrm{write}_{WR}(w)$, and $a = value(r) = value(w)$. In order to prove that $(R!sensitive)$ holds, we go to the definition of the boolean statement sensitive and see that it is a conjunction of two statements. We will only prove the first one as an example:

$$R!y.seq[i] \in \{x_0.oldseq[i], x_0.seq[i]\}. \tag{16}$$

The axioms (in A_5) connecting the values of variables assigned by **read** instructions tell us that $R!y = (R : 4)!y = value(R : 4) = a$. On the other hand, since $a = value(w)$ and $In(w, V)$, it can be seen that

$$a.seq = V!seq \tag{17}$$

(that is, the value of field seq is the same in both writes in V, and thus $a.seq$ is the value of variable seq in V).

Let $v = (R!x_0)$. We will prove that either $a.seq[i] = v.seq[i]$ or else that $a.seq[i] = v.oldseq[i]$, therby proving (16).

According to the axioms in A_5, the value of x_0 is determined by the read at line 0 (more precisely by the second instruction there, referred to as 0_2). Let $r_0 = (R : 0_2)$. Then $v = value(r_0)$.

In the assumptions of the lemma and in the following discussion, $(V : 2)$ refers to the write event on register WR rather than to the assignment instruction in that line, that is to $(V:2_1)$. (It would have been better to write each instruction in the protocol on a separate line perhaps.) Since $(V:2) \prec R$, and $In(r_0, R)$, $(V:2) \prec r_0$ follows from the higher-level precedence axioms. Now, since $(V:2)$ is a write on *WR* register, and r_0 a read of that register,

$$(V:2) \preceq \omega(r_0) \tag{18}$$

follows from the *regularity* and *seriality* axioms.

Claim 6.2 *Either* $(V:2) = \omega(r_0)$*, or* $(V:3) = \omega(r_0)$*, or the successor* Write*,* V^+*, exists and* $(V^+:2) = \omega(r_0)$*.*

Assuming for a moment this claim, we must prove the lemma in each of its three cases, but we only do the third case, since it is the most interesting. Assume $(V^+:2) = \omega(r_0)$. It follows from this assumption that

$$value(V^+:2) = value(r_0) = v. \tag{19}$$

We will prove that $a.seq[i] = v.oldseq[i]$ which concludes the lemma.

Now $value(V^+:2).oldseq$ is the value of variable *oldseq* at the start event of V^+, and there is an axiom to say that this is the same value of that variable at V. But $(V!oldseq) = (V!seq)$ (a formal deduction of this follows from the semantics of the assignment instruction $oldseq := seq$ at line 2). Hence

$$value(V^+:2).oldseq = V!seq. \tag{20}$$

Thus

$$v.oldseq = V!seq = a.seq.$$

(See equations (19), (20), and (17).)

Return to the proof of the claim. It follows from the axioms at the end of A_4 that for some Writer event X, $In(\omega(r_0), X)$. Since (as we saw in equation (18)) $(V:2) \preceq \omega(r_0)$,

$$V \preceq X$$

follows (or else $X \prec V$ by seriality of Writer, and then $In(\omega(r_0), X)$ implies $\omega(r_0) \prec V$, and thus $\omega(r_0) \prec (V:2)$ which brings the contradiction $\omega(r_0) \prec \omega(r_0)$).

There are thus two cases: $V = X$ and $V \prec X$. In the first, $In(\omega(r_0), V)$ and the lemma follows since V contains only two writes on *WR*, $(V:2)$ and $(V:3)$ (this should follows from the axioms.)

In the second case $V \prec X$ and hence V^+ exists and $V^+ \preceq X$ (this follows from seriality and the finiteness scheme). Hence $(V^+ : 2) \preceq \omega(r_0)$ follows from the fact that (V^+) is the first write in V^+ and from the higher-level precedence axioms. We must show that

$$(V^+ : 2) = \omega(r_0).$$

If this is not the case then $(V^+ : 2) \prec \omega(r_0)$ and hence $(V^+ : 3) \preceq \omega(r_0)$ follows from the fact that $(V^+ : 3)$ is the successor of $(V^+ : 2)$ in the set of writes on *WR*. Now $(V^+ : 2) \prec (V^+ : 3) \mid\!\!\succ r_0 \prec r$ follows, and the Russell–Wiener axiom implies that $(V^+ : 2) \prec r$ in contradiction to the regularity of ω (since $\omega(r) \prec V^+$).

All the other lemmas can be proved in a similar fashion, and thus the proof of the specification list can be written as a first-oreder proof. (The higher-level part of the proof (Section 4) is already written as a first-order deduction.)

Appendix: How to deal with regular registers in the context of histories

Consider a structure $A = (N; \leq, S, P_1, \ldots, P_n, 0)$ where N is the set of natural numbers (the universe of A), $\leq$ is the usual ordering relation, S is the successor function, and P_i's are diverse one-place predicates (monadic predicates). The numbers are suppose to represent states of some computation, 0 is the initial state, $S(x)$ is the state following x, and the predicates describe properties of the states.

It is very convenient to allow, in addition to the unary predicates, unary functions from N into some fixed finite domain D. Since predicates can encode such functions, this expansion is not essential. It demands however that we add D to A and thus make A into a two-sorted structure. These functions are also called "state variables".

Lower-level events can be defined in A: We say that the pair $(x, S(x))$ is an event, and we let $(e_i \mid i \in N)$ be an enumeration of these events. That is, e_0 is the event connecting the initial state with state number one, etc.

Now suppose that E is a collection of pairwise disjoint, finite sets of events. That is, if $X \in E$ then $X \subset \{e_i \mid i \in N\}$ is finite, and $X \cup Y = \emptyset$ for

$X \neq Y$ in E. Let $E = W \cup R$ be a partition of E into two sets called the Write and Read Events respectively. That is each $e \in W$ is a finite set of events representing some compound Write higher-level Event, and each set in R represents some Read Event. We want to model the assumption that these Read/Write Events represent a regular register. In our paper, we only took the start and conclude events and called them compound events, and the reader who wishes to do so may assume that each set in E consists of only two events. An ordering $\prec$ is defined on E as before. We assume that some $w \in W$ called the initial Write, $\prec$-precedes all other events in E. We also assume that both R and W are linearly ordered by $\prec$.

Let *value* be a function defined on E (taking some values from a fixed finite domain), and let ω be a regular function from R into W. That is

1. For all $r \in R$, $value(r) = value(\omega(r))$, and
2. $\omega(r) \not\succ r$, and there is no $w \in W$ such that $\omega(r) \prec w \prec r$.

I wish to describe here how this regularity condition on ω can be transported to the structure A. The problem is that A does not contain functions from N to N, only from N to D. (If we allow arbitrary functions, we step out of the realm of states and we loose decidability.) The solution is to observe that ω is a function of "bounded depth" (See Definition 2.6 in "On system executions and states" J. Applied Intelligence 3, 17–30 (1993); or "What is a state of a system (an outline)" in Droste and Gurevich editors, Semantics of Programming Languages and Model Theory, Gordon and Breach 1993.)

To model regular registers with histories, we keep the following state variables:

LastValue: keeps the value of the last Write that has finished.

OngoingValue: keeps the value of the ongoing Write if there is one, and is undefined otherwise. After the conclude event, *OngoingValue* becomes the *LastValue*.

OngoingRead: is a boolean variable that tells whether there is a Read Event such that we are in a state in between the initial and final states of that Read; this variable becomes true at any start event of a Read, and turns false at the corresponding conclude event.

ReturnValue: keeps the return value of the Read: It may be undefined all along the Read, but it must get some value at the conclude event. The value of *ReturnValue* can be determined (only once) at any stage in which the Read is *ongoing*, and its value then is either the *LastValue* or the *OngoingValue*.

I believe that a combination of assertional methods with the definition of regular registers as given above will yield a proof of the axioms that I stated in the preceding section, and will thus give a general formal basis for my rather ad hoc approach.