Correctness proofs for two protocols

Uri Abraham

February 6, 1996

Abstract

A methodology for proving the correctness of protocols is presented through two examples. (1) The Bloom two-writer register. (2) The Consumer/Producer buffer implementation. Each proof is divided into two levels: The programs are mentioned only at the lower level, and the specifications obtained at that level are abstractly used in the higher-level stage.

In fact, improved versions of the two protocols are given. (1) The protocol of Bloom is improved by not requiring the writers of the implemented register to read data of one another (they only read coordinating values). (2) The buffer implementation given here is more efficient than the traditional one.

The aim of this paper is to describe an approach for specifying and proving the correctness of distributed protocols. Our approach is characterized by a clear division of the correctness proofs into two or more layers: The lower layer involves the program, but the higher levels only mention abstract higher-level properties of system executions.

We expose this approach via two examples that are variants of simple and well known protocols. The first is an implementation of a two-writer multireader atomic register which uses only single writer registers (Bloom [1987]). The second is the classical implementation of a circular queue for the producer/consumer problem.

The method of proof presented here has enabled us to improve both protocols. This is a natural consequence of an approach that looks at the protocols from a higher-level view and allows for more abstract reasoning. Both protocols are improved by disassociating the control values from the data values. A data value is the useful information carried by the register or the queue, while control values are used by the processes to coordinate their activities. For example, time-stamps are coordinating control values, while a file of data put into the buffer by the writer is a data value. (We are not giving here a formal definition of this distinction which is quite intuitive.) Most often, the data and control information are mixed into a single record with two fields; the control value is seen then as a 'tag' that helps in processing the data. However, a disadvantage of such an association is that for very long data values (which may extend over huge files) a considerable slowing down of the processes may result. For example, in Bloom protocol, the writer processes must also read each other's data values which they never use (and moreover this may be undesirable if privacy is important). In our variant of this protocol, the writers only read control values.

The improvement of the queue implementation given in the second part of the paper is in showing how to implement an n place queue with only n buffers (the classical implementation needs one extra buffer). Again, this improvement is obtained as a result of thinking in abstract terms which disassociate the control from the information. The two proofs given here can be read independently of each other.

1 Introduction

When a distributed system with several concurrent communicating processes runs, the resulting entity with its set of events is modeled by a 'system execution'. The processes use different communication means such as channels, registers (shared variables), etc., and these can be implemented in diverse ways which may combine procedures and electronic devises. Thus, a single 'higher-level' statement, such as 'write v into channel c', which is elementary for the process using it, may in fact be quite complex when its execution is analyzed into its many 'lower-level' events. It is the same run of the system that can be viewed and represented at its different levels, and thus different system executions may correspond to different views of the same system run, but with different granularities. Given two such system executions, $\mathcal{H}$ and $\mathcal{L}$, we say that $\mathcal{H}$ is a 'higher-level view' of $\mathcal{L}$ (and $\mathcal{L}$ is a 'lower-level view' of $\mathcal{H}$) if $\mathcal{L}$ provides a finer description than $\mathcal{H}$: the events in $\mathcal{H}$ are abstract entities, and they correspond to *sets* of events in $\mathcal{L}$— each such set is an *implementation* (interpretation) of the higher-level event it constitutes.

Some properties of a protocol are expressed in the language of the higher-level system execution $\mathcal{H}$. For example, to state that a protocol implements an atomic register, or that it satisfies the mutual-exclusion property, is to describe its higher-level global behavior which is expressed in $\mathcal{H}$. Some other properties of a protocol, such as that it is 'wait free', refer also to its lower-view description. Of course, one must refer to the lower level and to the details of the protocol's text in order to prove its higher-level properties, but it seems desirable to minimize this lower-level reference since the complexity of the proof may be unbearable otherwise.

In the following we introduce the basic notions of a 'system execution' and the higher and lower-level views of system executions through an example: The *aimless protocol*. Its properties will be used in Section 3 to prove the correctness of Bloom protocol and its variant. More details about these notions can be found in Lamport [1986], in Pratt [1986], and in the first part of Abraham[.]. The reader of Lamport's or Pratt's work should note that some of the terms used here differ from their original definitions.

1.1 The aimless protocol: An introduction to basic notions

The heart of Bloom protocol is the following "aimless" protocol. There are two processes: P_0, the *Copier*, and P_1, the *Changer*. They communicate through two single-writer, single-reader serial registers, R_0 and R_1. These are boolean registers initially set to 0 (it is convenient to assume initial writes of 0 to the registers). P_i writes onto R_i and the other process reads it. The protocol for P_0 is to read R_1 and to copy into R_0 the value returned. The protocol for P_1 is to reverse the value found in P_0's register.

To emphasize the distinction between assignment to a local variable and the read of a register we use 'read($\mathsf{R}; x$)', and 'write($\mathsf{R}; v$)' instead of $x := \mathsf{R}$, and $\mathsf{R} := v$. The protocol of P_0 is called *COPY*, and the protocol for P_1 is called *CHANGE*. (We will be using uppercase words to name both procedures and their executions.)

COPY

1. read($\mathsf{R}_1; t$);
2. write($\mathsf{R}_0; t$).

CHANGE

1. read($\mathsf{R}_0; t$);
2. $t := 1 - t$;
3. write($\mathsf{R}_1; t$).

When *Copier* and *Changer* concurrently and repeatedly execute their protocols, *COPY* and *CHANGE*, the resulting run is modeled by a system execution. It contains a temporal precedence relation $<$, defined on a collection of events denoted E, these are all the read and write events on registers R_0 and R_1. The meaning of $x < y$ is that event x precedes event y. In this simple example we may assume that $<$ is a linear order on E, but when the events are grouped into higher-level events (*COPY* and *CHANGE* events), the linearity is lost, because an event now corresponds to an interval extended in time, and intervals may overlap. Since it is important to be able to reason about the higher-level execution of a protocol as well, the partial-order approach is taken and the notion of interval-ordering has to be introduced. An irreflexive and transitive relation R on E is called an *interval-order* iff

$$\forall a, b, c, d \in E(aRb \wedge \neg cRb \wedge cRd \Longrightarrow aRd) \tag{1}$$

Wiener [1914] proved that any interval ordering can be represented by intervals of a linear order. This is the justification for the choice of interval orderings to model the precedence relation.

Definition 1.1 (Lamport) *A* global-time model *is a structure* $\mathcal{S} = (E, <_{\mathcal{S}})$ *where* E *is a set (called the set of events) and* $<_{\mathcal{S}}$ *is a partial irreflexive order relation on* E *such that the following holds, where* $\not>_{\mathcal{S}}$ *is defined on* E *by* $a \not>_{\mathcal{S}} b$ *iff* $\neg(b <_{\mathcal{S}} a)$*:*

1. $<_{\mathcal{S}}$ *is an interval-order on* E.
2. *For any* $a \in E$, $\{x \in E \mid x \not>_{\mathcal{S}} a\}$ *is finite. (Finiteness condition)*

(It follows that a global-time model is countable, and it can be infinite.)

The following (Anger's property, Anger [1989]) can easily be seen to hold in any global-time model: $a \not>_{\mathcal{S}} b <_{\mathcal{S}} c \not>_{\mathcal{S}} d$ imply $a \not>_{\mathcal{S}} d$.

We write $x \leq_{\mathcal{S}} y$ as a shorthand for $x <_{\mathcal{S}} y$ or $x = y$.

The global-time model only gives the precedence relations among the events but not any other useful information. For example, if $r, w \in E$ correspond to the read and write of registers R_1 and R_0 respectively done in the execution of the two lines of the *COPY* protocol, then $r <_{\mathcal{S}} w$, but the model does not tell us what are the values read or written. Nor does the model describes those fixed objects which are not situated in time and do not fall under the precedence relation (such as the set of possible register values or the program text that is executed). For this reason the definition of 'system execution' must involve fixed objects, relations, and functions.

Definition 1.2 *A* system execution *is a structure $\mathcal{S} = (E, C, <_{\mathcal{S}}, F)$ such that $(E, <_{\mathcal{S}})$ is a global-time model, C is a set (of 'fixed' objects), and F is a collection of unary functions and relations on $E \cup C$. (For convenience, C is omitted when its definition is obvious.)*

To continue the exposition with our 'aimless' example, let $\mathcal{S}$ be an execution of the protocol. The 'higher-level view', $\mathcal{H}$, of $\mathcal{S}$ is obtained by defining first the higher-level events as follows. Each higher-level event is formed by grouping the events in E that belong to a single complete execution of either the *COPY* or the *CHANGE* protocol. There are thus two kinds of higher-level events in our example: a *COPY* event which is an execution of the *COPY* protocol, and a *CHANGE* event which is an execution of the *CHANGE* protocol. Let H be the set of all higher-level events thus obtained. An element X of H is hence also a subset of E, and the precedence relation $<_{\mathcal{H}}$ is defined on H in the following way:

$$A <_{\mathcal{H}} B$$

iff

$$\forall a \in A, \forall b \in B (a <_{\mathcal{S}} b).$$

We also add to $\mathcal{H}$ two events, $Initialization_0$ and $Initialization_1$, which are the singletons containing the initial writes in $\mathcal{S}$ onto R_0 and R_1 respectively. Though these *Initialization* events are in *Copier* and *Changer*, they are not *COPY* or *CHANGE* events since they are not execution of the protocol. An *Initialization* event precedes any non *Initialization* event.

The collection of these higher-level events, together with the precedence relation $<_{\mathcal{H}}$, and the functions and relations that will be added later, form the higher-level view $\mathcal{H}$ of $\mathcal{S}$.

Notation: If $X \in H$ is a higher-level event and $x \in X$, then we let $X = [x]$. (This notation is unambiguous if distinct events in H are disjoint in E).

It is convenient sometimes to relate higher and lower-level events: $X <_{\mathcal{S}} a$ for $X \in H$ and $a \in E$, for example, means that $\forall x \in X (x <_{\mathcal{S}} a)$.

We define the relation $A <_{\mathcal{H}} end(B)$ for $A, B \in H$ by

$$\forall a \in A \ (a \leq_{\mathcal{S}} b),$$

where b is the last event in B (assuming B has a last event). If $A <_{\mathcal{H}} end(B)$ then for every C such that $B <_{\mathcal{H}} C$, $A <_{\mathcal{H}} C$ holds. This last property can be taken as the definition of $A <_{\mathcal{H}} end(B)$, even when $\mathcal{H}$ is not a higher-level system execution.

1.2 Higher-level relations for the aimless protocol

Let system executions $\mathcal{S} = (E, <_{\mathcal{S}}, F)$ and $\mathcal{H} = (H, <_{\mathcal{H}}, G)$ be a lower-level and a higher-level views of an execution of the aimless protocol. The registers R_i $(i = 0, 1)$ are assumed to be serial in $\mathcal{S}$. We assume a function, ω, in F, that associates to each read event, $r \in E$ of register R_i, a write event $\omega(r)$ on the same register as follows: $\omega(r)$ is the $<_{\mathcal{S}}$ rightmost write w onto R_i such that $w <_{\mathcal{S}} r$ (recall that an initialization write which precedes all reads is assumed). It is assumed that $value(\omega(r)) = value(r)$ for all r. We say that $\omega(r)$ is the write event that affects the read r.

Except for the *Initialization* events, the events in H are of two types: *CHANGE* (in *Changer*) and *COPY* (in *Copier*), according as to whether the lower-level view of the event is an execution of the *CHANGE* or the *COPY* protocol. Thus the name of the procedure is also used to denote its execution.

Definition 1.3 *1. The function $colour(X)$ is defined on the events in H. $colour(X)$ is the value (0 or 1) of the variable t written onto register* R_0 *(if X is a COPY) or* R_1 *(if X is a CHANGE). We assumed that $colour(I) = 0$ for an Initialization event I.*

2. The function 'observe' is defined on the COPY and CHANGE events in H by $observe(U) = V$ iff for the (only one) read event $r \in U$, $V = [\omega(r)]$. When the write $\omega(r)$ is the initial write onto the register, we say that U observes the Initialization. So observe takes COPY events into Changer events, and CHANGE events into Copier events.

3. *For every CHANGE event, $V \in H$, define $Dom(V) \subset Copier$ as follows. Let $U_0 = observe(V)$. For any Copier event $U \in H$, we let $U \in Dom(V)$ iff: $U = U_0$ or the following holds*

   (a) *$U_0 <_{\mathcal{H}} U$, and*

   (b) *$colour(U) = colour(U_0)$ and*

   (c) *$observe(U) <_{\mathcal{H}} V$.*

The following lemma lists some easy to establish properties of these functions.

Lemma 1.4 *1. The function observe, and the boolean function colour, satisfy the following.*

   (a) *$observe(U) <_{\mathcal{H}} U$,*

   (b) *If $V = observe(U)$ and V' is in the same process as V (Changer or Copier) then $V <_{\mathcal{H}} V' <_{\mathcal{H}} U$ does not hold. (This property of observe is called* regularity.*)*

2. *If U is a CHANGE event then $observe(U)$ is in Copier and*

$$colour(U) \neq colour(observe(U)).$$

3. *If U is a COPY then $observe(U)$ is in Changer and*

$$colour(U) = colour(observe(U)).$$

Remark. For any events $U_1 \leq_{\mathcal{H}} U_2$ in the same process, $observe(U_1) \leq_{\mathcal{H}} observe(U_2)$. To see this, put $W_i = observe(U_i)$, for $i = 1, 2$. If $W_1 \leq_{\mathcal{H}} W_2$ does not hold, then $W_2 <_{\mathcal{H}} W_1$ by the seriality of *Changer* and *Copier*. But $W_1 <_{\mathcal{H}} U_1$ by Lemma 1.4.*1(a)*. So $W_2 <_{\mathcal{H}} W_1 <_{\mathcal{H}} U_1 \leq_{\mathcal{H}} U_2$ follows in contradiction to the regularity of *observe*(Lemma 1.4.*1(b)*).

Our aim is to prove the following theorem which will be used in the correctness proof of Bloom protocol and its variant. Let $\mathcal{H}$ be a system execution. We say that $\mathcal{H}$ is a *Changer/Copier* system execution if it has two serial processes: *Changer* and *Copier*; with events called *CHANGE* and *COPY* (respectively) and initialization events (that preced all other events), and it has functions called *observe* and *colour* so that the properties established in Lemma 1.4 hold.

Theorem 1.5 *Let $\mathcal{H}$ be a Changer/Copier system execution. For any CHANGE, V, $Dom(V)$ is convex. I.e., if U_0, U_1, U_2 are in Copier and $U_0 <_{\mathcal{H}} U_1 <_{\mathcal{H}} U_2$ and $U_0, U_2 \in Dom(V)$, then $U_1 \in Dom(V)$ as well.*

<u>Proof</u>. Given a *CHANGE*, V, the finiteness condition (Definition 1.1) implies that $\{X \mid X <_{\mathcal{H}} V\}$ is finite. Thus, if the Theorem does not hold, there is a least counter-example, V. So,

1. For some *Copier* events $U_0 <_{\mathcal{H}} U_1 <_{\mathcal{H}} U_2$, $U_i \in Dom(V)$ for $i = 0, 2$ but not for $i = 1$.

2. V is the $<_{\mathcal{H}}$ minimal *CHANGE* event satisfying 1.

Since, by definition, $observe(V)$ is the $<_{\mathcal{H}}$ least event in $Dom(V)$, there is no loss of generality in assuming that $U_0 = observe(V)$. Put

$$c = colour(U_0),$$

then $c = colour(U_2)$ (since $U_2 \in DOM(V)$).

U_1 and U_2 are *COPY* events, because the *Initialization* events are not preceded by any event. Let, for $i = 1, 2$,

$$W_i = observe(U_i).$$

Then:

1. $colour(U_i) = colour(W_i)$, for $i = 1, 2$, since U_i are *COPY* events. In particular, $c = colour(U_2) = colour(W_2)$.

2. $W_1 \leq_{\mathcal{H}} W_2$, by the Remark made after Lemma 1.4.

Since $U_2 \in Dom(V)$,

$$W_2 <_{\mathcal{H}} V.$$

Hence $W_1 <_{\mathcal{H}} V$. So U_1 satisfies (a) and (c) of the definition of $Dom(V)$, and if $c = colour(U_1)$ then $U_1 \in Dom(V)$ is the required contradiction. So assume

$$\overline{c} = 1 - c = colour(U_1).$$

This exclude $W_1 = W_2$, and we may assume now that $W_1 <_{\mathcal{H}} W_2$. (So W_2 is not the *Initialization* of *Changer*, and is hence a *CHANGE* event; we are going to see that it contradicts the minimality of V.)

Let $T = observe(W_2)$. Since W_2 is a *CHANGE*, $colour(T) \neq colour(W_2)$, but as $c = colour(W_2)$,

$$\overline{c} = colour(T).$$

Since $observe(V) = U_0$ and $W_2 <_{\mathcal{H}} V$, by our Remark above,

$$T \leq_{\mathcal{H}} U_0.$$

The case $T = U_0$ is ruled out because in such a case the equality $colour(T) = colour(U_0) = c$ is impossible since $\overline{c} = colour(T)$.

In case $T <_{\mathcal{H}} U_0$, W_2 and the three events $T <_{\mathcal{H}} U_0 <_{\mathcal{H}} U_1$ contradict the minimality of V, because $W_2 <_{\mathcal{H}} V$ and the *CHANGE* W_2 satisfies:

1. $T \in Dom(W_2)$ (as $T = observe(W_2)$),
2. $U_1 \in Dom(W_2)$ as $T <_{\mathcal{H}} U_1$, $colour(T) = colour(U_1) = \overline{c}$, and $observe(U_1) = W_1 <_{\mathcal{H}} W_2$,
3. $U_0 \notin Dom(W_2)$ (as $c = colour(U_0)$).

Corollary 1.6 *If $W_1 <_{\mathcal{H}} W_2$ are CHANGE events, then either $colour(W_1) = colour(W_2)$, or for every $X \in Dom(W_1)$, and $Y \in Dom(W_2)$, $X <_{\mathcal{H}} Y$.*

Proof. Assume that W_1, W_2, X, Y are as in the Corollary, but $Y \leq_{\mathcal{H}} X$. We will prove the equality $colour(W_1) = colour(W_2)$. Put $Y_0 = observe(W_2)$, and $X_0 = observe(W_1)$, then $X_0 \leq_{\mathcal{H}} Y_0$ by the Remark made after Lemma 1.4. And $Y_0 \leq_{\mathcal{H}} Y$ by definition of $Dom(W_2)$. The convexity of $Dom(W_1)$, and $X_0 \leq_{\mathcal{H}} Y \leq_{\mathcal{H}} X$ imply that $Y \in Dom(W_1)$. But then $Y \in Dom(W_1) \cap Dom(W_2)$ implies $colour(W_1) = colour(W_2)$, because for any *CHANGE*, W, and $Y \in Dom(W)$, $colour(W) = 1 - colour(Y)$.

2 Bloom two writer protocol

The problem of implementing an atomic two writer protocol can be described informally as follows. Suppose two processes, $Writer_0$ and $Writer_1$, that repeatedly and independently transmit messages to a set of *Readers*. The communication means are two single-writer serial registers that connect any *Writer* to the other processes. A *Reader* has to decide which message to return—the one by $Writer_0$ or the one by $Writer_1$—and this choice must be consistent with some linear ordering of all the higher-level events. This means that there is a linear ordering, $\prec$, extending $<_{\mathcal{H}}$, and such that for any read event R, the value returned by R equals the value written by the $\prec$-maximal write $W \prec R$.

Bloom [1987] was the first to solve this problem; his solution follows. Registers R_0, and R_1, are written by $Writer_0$, and $Writer_1$ respectively, and read by the other process. Each register has two fields: colour and message. The values of colour are 0 and 1, and message carries values of a determined type, **message**, which the implemented register has to transmit. The local variables $t,\ t_0,\ t_1$ are boolean ($\{0,\ 1\}$); and $v,\ v_0,\ v_1,\ v_2,\ a$ are of type **message**.

WRITE (a : **message**); (* For $Writer_0$ *)

1. read($\mathsf{R}_1; \langle t, v\rangle$);
2. write($\mathsf{R}_0; \langle t, a\rangle$);

WRITE (a : **message**); (* For $Writer_1$ *)

1. read($\mathsf{R}_0; \langle t, v\rangle$);
2. $t := 1 - t$;
3. write($\mathsf{R}_1; \langle t, a\rangle$);

READ

1. read($\mathsf{R}_0; \langle t_0, v_0\rangle$)
2. read($\mathsf{R}_1; \langle t_1, v_1\rangle$)
3. $r := \begin{cases} 0 & \text{IF } t_0 = t_1 \\ 1 & \text{IF } t_0 \neq t_1 \end{cases}$
4. read($\mathsf{R}_r; \langle t_2, v_2\rangle$)
5. RETURN v_2

We begin with some informal remarks. Observe that the *Writers* are executing the aimless protocol with only one additional feature: the message a is also written into the registers. The *Reader* does not write on any register, it only reads three times the writers' registers. At first, it reads the registers of $Writer_0$ and $Writer_1$ (in that order!) and then it computes the index r to be returned and finally it reads again the register of $Writer_r$ and returns the message found there. So the first two reads serve to choose the return index, and the third one to find the message. Let R be an execution of the *READ* protocol; the last read (the execution of line 4) is singled out and is denoted $S(R)$. The intuition behind the choice of the return index r is the following. The *READ* wants to determine who is the winner in the aimless game, the *Copier* or the *Changer*, and it is the index of the process that seems to be the winner that is returned. Both of the *Writers* are assumed to be serial, but the *Reader* is not necessarily serial, and it may consist of several independent processes.

The aimless protocol is, in a sense, a *module* of Bloom protocol, it is an ingredient which can be studied in isolation, but, unlike a procedure, it does not appear as it is in the protocol. It is rather like a sketch of a figure in a painting. Formally, the notion of a 'restriction' will be used below to show how our results on the aimless protocol can be applied to the Bloom protocol.

It is instructive to see what goes wrong when the third read is omited from the *READ* protocol, and lines 4 and 5 are replaced by RETURN v_r. There may be $W_0 <_{\mathcal{H}} W_1 <_{\mathcal{H}} R$ where W_0 and W_1 are *WRITE* events by $Writer_0$ and $Writer_1$ respectively, and R a *READ*, and R returns W_0.

Let $\mathcal{L} = (L, <_{\mathcal{L}}, F)$ be a lower-level view of an execution of the protocol (the fixed objects are omited for simplicity). We assume that the processes in $\mathcal{L}$ are $Writer_0$ and $Writer_1$ and a number of reading processes which execute the protocol. The registers, R_0 and R_1, used by the *Writer*s for writing and reading, and by the *Reader*s for reading are assumed to be serial in $\mathcal{L}$. The function ω in F associates, to every read $r \in L$ of register R_i, the $<_{\mathcal{L}}$ rightmost write w to R_i such that $w <_{\mathcal{L}} r$. For some initial message a_0, the pair $(0, a_0)$ is assumed to be initially written to R_0 and to R_1.

Let $\mathcal{H} = (H, <_{\mathcal{H}}, G)$ be the higher-level view obtained by lumping together the following sets of events in L.

1. Every single execution of the *READ* or *WRITE* protocol.

2. A singleton comprising one of the *Initialization* write events.

3. Singletons comprising a read execution of line 4 of the *READ* protocol.

The events in H are thus of type *READ* or *WRITE*, or the two special events $Initialization_0$ and $Initialization_1$, or are events of the form $S(R)$ for a *READ* event R. ($S(R)$ is the last read in R, which is an execution of line 4.) We say that X is a *Writer* event if X is in $Writer_i$ for $i = 0$ or $i = 1$; that is, X is either a *WRITE* event (an execution of the protocol) or one of the *Initialization* events. A *Reader* event is either a *READ* or of the form $S(R)$ for some *READ* event R.

We are going to define on H several more functions. To begin with set $index(W) = i$ iff W is executed by $Writer_i$.

The function *colour* associates to a *Writer* event W the value of the field colour, as written on its register. The function *observe*, from *WRITE* events in $Writer_k$ into $Writer_{1-k}$ is defined as follows. For a *WRITE* event, W, $observe(W) = V$ iff for the read event $r \in W$, $V = [\omega(r)]$ holds. As we already remarked, if we omit the messages then the restriction of $\mathcal{H}$ to $Writer_0$ (as *Copier*) and to $Writer_1$ (as *Changer*) together with the functions *colour* and *observe*, gives the aimless protocol's higher-level properties.

For any event X in H, define $message(X)$ to be of type **message**: For a *Writer* event this is the value of the parameter a (or a_0 for an *Initialization* event); and for a *Reader* event it is the value returned (in line 5 of the *READ* protocol).

The function S has been defined on the *READ* events: $S(R)$ is a singleton comprising the last read in R.

The definition of the return function $\Omega : READ \longrightarrow Writer$ is natural. For every *READ* event, R, let $r \in R$ be the read (of R_0 or R_1) executed for line 4 of the protocol. Let $w = \omega(r)$, and let W be the unique *Writer* event in H such that W includes w. Then set $W = \Omega(R)$ (that is, $\Omega(R) = [\omega(r)]$). $index(\Omega(R))$ is called "the return index of R". We will later see that there is no *WRITE* event W_1 such that $\Omega(R) <_{\mathcal{H}} W_1 <_{\mathcal{H}} R$. For now, it suffices to observe the obvious property that there is no such W_1 with $index(W_1) = index(W)$.

We need to introduce to $\mathcal{H}$ another function: $consider(R, i)$ defined for a *READ* event R and an index $i = 0, 1$. Let r_0 and r_1 be the two reads in R of registers R_0 and R_1 respectively executed for lines 1 and 2 of the *READ* protocol. Then define:

$$consider(R, 0) = [\omega(r_0)],\ consider(R, 1) = [\omega(r_1)].$$

2.1 The higher-level specifications of the Bloom protocol

This section lists abstracts properties of a system execution $\mathcal{H}$. These hold whenever $\mathcal{H}$ is a higher-level execution of the Bloom protocol, but they may hold for other protocols as well (as we will see in Section 3). In Section 2.2 it will be proved that atomicity is a consequence of these specifications.

In $\mathcal{H} = (H, <_{\mathcal{H}}, G)$, the events are partitioned into the following processes:

1. The *Reader* process, which is not necessarily serial. There are two types of *Reader*'s events: *READ* events, and for each *READ* event, R, an event $S(R)$ is associated.

2. Two *Writer* processes, $Writer_0$ and $Writer_1$, each one serial. $Writer_i$ contains the event $Initialization_i$, and its other events are called *WRITE* events. $Initialization_i <_{\mathcal{H}} X$ for any *WRITE* or *READ* event X.

   We let $index(X) = i$ if X is in $Writer_i$.

G also contains the following functions (with properties as defined below). $consider(R, i)$ is defined for a *READ*, R, and a *Writer*'s index $i = 0, 1$. $consider(R, i)$ is in $Writer_i$. $colour(W)$ is a $\{0, 1\}$-valued function defined on the *Writer* events. $observe(W)$ is a function defined on *WRITE* events with values in *Writer*. $\Omega(R)$ is a function defined on the *READ* events and taking in *Writer* events values. $message(X)$ is of type **message**. If R is a *READ* event, $message(R)$ is called the return value of R.
Specifications:

cons.1 $consider(R, 0)$ and $consider(R, 1)$ are regular functions defined for any *READ* event R. This means that for every $i = 0, 1$ and *READ*, R, $consider(R, i) \not>_{\mathcal{H}} R$, and there is no $Writer_i$ event W such that

$$consider(R, i) <_{\mathcal{H}} W <_{\mathcal{H}} R.$$

cons.2 : $consider(R, 0) <_{\mathcal{H}} R$.

cons.3 : The function S associates with each *READ* event R a *Reader* event, $S(R)$, such that

$$consider(R, i) <_{\mathcal{H}} S(R), \text{and } begin(R) <_{\mathcal{H}} S(R).$$

observe **and** *colour* **specifications**

obs.1 For any *WRITE* event W, $observe(W)$ satisfies:

1. $index(observe(W)) \neq index(W)$.
2. $observe(W) <_{\mathcal{H}} W$.
3. There is no *WRITE* V with $index(V) = index(observe(W))$ such that $observe(W) <_{\mathcal{H}} V <_{\mathcal{H}} W$.

obs.2 For any *WRITE*, W:

$$index(W) = 1 \textit{ iff } colour(W) \neq colour(observe(W)).$$

Ω specification

ret.1 $message(R) = message(\Omega(R))$.

ret.2 If $V_i = consider(R, i)$ for $i = 0, 1$, then $index(\Omega(R))$, the return index, equals 0 if $colour(V_0) = colour(V_1)$; the return index is 1 otherwise.

ret.3 $\Omega(R) <_{\mathcal{H}} end(R)$, and there is no *WRITE*, W, such that $index(W) = index(\Omega(R))$ and $\Omega(R) <_{\mathcal{H}} W <_{\mathcal{H}} S(R)$.

This ends the list of specifications. They are all easy to establish for any execution of the protocol of Bloom. In the following subsection it will be proved that they imply atomicity.

Let $\mathcal{H}$ be any system execution in which these specifications hold. Restrict $\mathcal{H}$ to $Writer_0$ and $Writer_1$, and let $\mathcal{H}_1$ be the resulting system execution. So $\mathcal{H}_1$ is obtained by throwing away the *Reader* events from $\mathcal{H}$. If the processes $Writer_0$ and $Writer_1$ are now called *Copier* and *Changer* respectively, then $\mathcal{H}_1$ becomes a *Changer*/*Copier* system execution, because the statements of Lemma 1.4 are part of the **obs** specifications. Hence Theorem 1.5 and Corollary 1.6 hold in $\mathcal{H}$.

2.2 The specifications imply atomicity

In this section we prove that if $\mathcal{H}$ satisfies the specifications of Section 2.1, then the implemented register is atomic. We will show that Ω satisfies in this case the following.

Peterson and Burns Axioms

1. For any *READ* event, R, $\Omega(R) = W$ is a *Writer* event in $\mathcal{H}$ and $message(W) = message(R)$.

2. (a) $\Omega(R) \not>_{\mathcal{H}} R$, and even $\Omega(R) <_{\mathcal{H}} end(R)$. (This is the 'Bloom condition')

   (b) it is never the case that $W_1 <_{\mathcal{H}} W_2 <_{\mathcal{H}} R$ where W_i are *Writer* events and R a *READ* and $W_1 = \Omega(R)$. (This property of Ω is called *regularity*.)

3. The following is possible only when $W_0 = W_1$: For $i = 0, 1$, W_i is a *Writer* event; R_i is a *READ* event

$$W_i = \Omega(R_i),$$

   and

$$W_0 <_{\mathcal{H}} R_1, \ W_1 <_{\mathcal{H}} R_0.$$

It follows from the work of Peterson and Burns [1987] that if these three properties hold, then the implemented register is atomic. Abraham [.] also contains a treatment of this issue.

Before proving that the specifications of Section 2.1 imply the three statements of Peterson and Burns, let us give the following hint to the reader. Reading a formal proof may be quite tedious, but, as in Euclidian geometry, illustrations can be of tremendous help. Since the partial ordering, of our system executions are interval orderings, the reader may represent the events by line intervals (see Abraham, Ben-David and Magidor [1990] for more details). Moreover, even though we do not mention the protocol at this stage of the formal proof, it makes sense to draw the illustrations in accordance with the protocol, just as it makes sense to draw lines and points to accompany abstract proofs in axiomatic geometry (which does not deal with physical lines and points).

The Peterson and Burns Axioms (1) and 2(a) are explicitly stated (**ret.1** and **ret.3**). Axiom 2(b) is proved in the following, and (3) will be proved afterwards.

To prove 2(b), suppose

$$W_1 <_{\mathcal{H}} W_2 <_{\mathcal{H}} R, \ W_1 = \Omega(R)$$

and we shall get a contradiction. By **ret.3**, $index(W_1) \neq index(W_2)$. Put $i = index(W_1)$, and $j = index(W_2)$. So i is the return index of R.

Claim 1: $W_1 = consider(R, i)$.

Proof: This is a simple consequence of the assumption $\Omega(R) <_{\mathcal{H}} R$. Let $V = consider(R, i)$; then V is in $Writer_i$, and $W_1 \leq_{\mathcal{H}} V$ since otherwise (by seriality of $Writer_i$) $V <_{\mathcal{H}} W_1 <_{\mathcal{H}} R$ contradicts **cons.1**. As $V = consider(R, i) <_{\mathcal{H}} S(R)$, **ret.3** shows the impossibility of $W_1 <_{\mathcal{H}} V$ which would imply $W_1 <_{\mathcal{H}} V <_{\mathcal{H}} S(R)$. Thus $W_1 = V$.

For the rest of the proof let $W = consider(R, j)$. Then $W_2 \leq_{\mathcal{H}} W$ by **cons.1**. So

$$W_1 <_{\mathcal{H}} W.$$

Thus $i \neq j$ follows from Claim 1. Also,

$$W <_{\mathcal{H}} S(R)$$

by **cons.3**.

Claim 2: $W_1 = observe(W)$.

Proof: By **obs.1**(1), $i = index(observe(W))$. By **obs.1**(3), $W_1 <_{\mathcal{H}} W$ implies $W_1 \leq_{\mathcal{H}} observe(W)$. If the Claim does not hold, then $W_1 <_{\mathcal{H}} observe(W)$. But $observe(W) <_{\mathcal{H}} W$ (by **obs.1**(2)), and $W <_{\mathcal{H}} S(R)$, imply $W_1 <_{\mathcal{H}} observe(W) <_{\mathcal{H}} S(R)$, and this contradicts **ret.3**.

To end the proof of Axiom 2(b), assume first $i = 0, j = 1$. Then $colour(W_1) \neq colour(W)$ by Claim 2 and **obs.2**. Hence the return index of R is 1 by Claim 1 and **ret.2**. But i is the return index! Now assume $i = 1, j = 0$. Then, by **obs.2**, $colour(W) = colour(W_1)$, and by **ret.2** the return index of R is 0. This contradicts $i = 1$. So 2(b) is settled.

Lemma 2.1 *Suppose R is a READ event such that $consider(R, i) = V_i$ for $i = 0, 1$. If $colour(V_1) \neq colour(V_0)$, and if $T_0 = observe(V_1)$, then $T_0 \leq_{\mathcal{H}} V_0$ implies $V_0 \in Dom(V_1)$.*

Proof. As $index(V_1) = 1$, $colour(T_0) \neq colour(V_1)$ by **obs.2**. By assumption $colour(V_1) \neq colour(V_0)$, and hence $colour(T_0) = colour(V_0)$. If we prove $observe(V_0) <_{\mathcal{H}} V_1$, then $V_0 \in Dom(V_1)$ follows the definition of Dom (Definition 1.3). Assume for the sake of a contradiction that $V_1 \leq_{\mathcal{H}} observe(V_0)$ (seriality of $Writer_1$ is used here). Equality is not possible since $colour(observe(V_0)) = colour(V_0)$. So $V_1 <_{\mathcal{H}} observe(V_0) <_{\mathcal{H}} V_0$ (by **obs.1**). But $V_0 <_{\mathcal{H}} R$ by **cons.2**. So $V_1 <_{\mathcal{H}} observe(V_0) <_{\mathcal{H}} R$, in contradiction to the regularity of $consider$ (**cons.1**). ■

Proof of Axiom 3. Assume $W_1 <_{\mathcal{H}} R_0$, $W_0 <_{\mathcal{H}} R_1$ and $W_i = \Omega(R_i)$, $i = 0, 1$. We assume $W_0 \neq W_1$, and shall get a contradiction. By Axiom 2(b), $index(W_0) \neq index(W_1)$. Because, otherwise, W_0 and W_1 are by the same process and hence $<_{\mathcal{H}}$ comparable or equal. If $W_0 <_{\mathcal{H}} W_1$ for example, then as $W_1 <_{\mathcal{H}} R_0$, a contradiction to Axiom 2(b) is obtained. By symmetry, assume $index(W_0) = 0, index(W_1) = 1$, and we shall get a contradiction. (It is here that our study of the aimless protocol pays off.)

Let V_0 and V_1 be the *Writer* events of indices 0 and 1 such that $V_i = consider(R_1, i)$, $i = 0, 1$. Since $W_1 = \Omega(R_1)$, has index 1, **ret.2** implies that

$$colour(V_1) \neq colour(V_0).$$

Put $T_0 = observe(V_1)$. Then

$$T_0 <_{\mathcal{H}} V_1.$$

$T_0 \in Dom(V_1)$ by definition, and $index(T_0) = 0$. Now let S_0 and S_1 be the *Writer* events of indices 0 and 1 such that $consider(R_0, i) = S_i$, $i = 0, 1$. By **ret.2**,

$$colour(S_0) = colour(S_1).$$

Claim 2.2 *$V_1 <_{\mathcal{H}} R_0$. And $S_0 <_{\mathcal{H}} R_1$.*

Proof. Since $consider(R_1, 1) = V_1 <_{\mathcal{H}} S(R_1)$, and $index(W_1) = 1$, $V_1 \leq_{\mathcal{H}} W_1$ by **ret.3**. By assumption, $W_1 <_{\mathcal{H}} R_0$. Hence $V_1 <_{\mathcal{H}} R_0$. $S_0 <_{\mathcal{H}} R_1$ is proved symmetrically. ∎

Claim 2.3 *$V_0 \in Dom(V_1)$.*

Proof. Assume for the sake of a contradiction that $V_0 \notin Dom(V_1)$. Since $colour(V_1) \neq colour(V_0)$, Lemma 2.1 can be applied: if $V_0 \notin Dom(V_1)$, then

$$V_0 <_{\mathcal{H}} T_0.$$

Since $T_0 = observe(V_1) <_{\mathcal{H}} V_1$, and $V_1 <_{\mathcal{H}} R_0$ (Claim 2.2), $T_0 <_{\mathcal{H}} R_0$. Hence $T_0 \leq_{\mathcal{H}} S_0$ (by **cons.1**). But $S_0 <_{\mathcal{H}} R_1$. Hence $V_0 <_{\mathcal{H}} T_0 <_{\mathcal{H}} R_1$. This, and **cons.1**, contradict $consider(R_1, 0) = V_0$. ∎

Claim 2.4 *1. $T_0 \leq_{\mathcal{H}} S_0 \leq_{\mathcal{H}} V_0$.*

2. $colour(S_0) \neq colour(V_1)$.

Proof. First, $T_0 <_{\mathcal{H}} V_1$, and $V_1 <_{\mathcal{H}} R_0$ (by Claim 2.2) imply $T_0 <_{\mathcal{H}} R_0$. Since $consider(R_0, 0) = S_0$, and $index(T_0) = index(S_0) = 0$, $T_0 \leq_{\mathcal{H}} S_0$. To prove that $S_0 \leq_{\mathcal{H}} V_0$, assume towards a contradiction that $V_0 <_{\mathcal{H}} S_0$. Recall that $S_0 <_{\mathcal{H}} R_1$. So $V_0 <_{\mathcal{H}} S_0 <_{\mathcal{H}} R_1$. Yet $index(V_0) = index(S_0) = 0$, and this contradicts $consider(R_1, 0) = V_0$.

By convexity of $Dom(V_1)$, as $T_0, V_0 \in Dom(V_1)$, $S_0 \in Dom(V_1)$ (as we argued, Theorem 1.5 can be applied here). Hence $colour(S_0) \neq colour(V_1)$. ∎

Claim 2.5 *$V_1 <_{\mathcal{H}} S_1$ and $colour(V_1) \neq colour(S_1)$.*

Proof. By Claim 2.2, $V_1 <_{\mathcal{H}} R_0$. Since $consider(R_0, 1) = S_1$,

$$V_1 \leq_{\mathcal{H}} S_1.$$

Now $colour(S_0) = colour(S_1)$, since $\Omega(R_0)$ has index 0. We proved in Claim 2.4 that $colour(S_0) \neq colour(V_1)$. Hence, $colour(S_1) \neq colour(V_1)$, and thus $S_1 \neq V_1$. ∎

Now we get the desired contradiction which proves Axiom 3. From Claim 2.5, by Corollary 1.6, $X <_{\mathcal{H}} Y$ for every $X \in Dom(V_1), Y \in Dom(S_1)$. Let $U_0 = observe(S_1)$. Then $U_0 \in Dom(S_1)$. $V_0 \in Dom(V_1)$ by Claim 2.3. So

$$V_0 <_{\mathcal{H}} U_0.$$

By **obs.1**(2), $U_0 = observe(S_1) <_{\mathcal{H}} S_1$, and by **cons.3**, as $S_1 = consider(R_0, 1)$, $S_1 <_{\mathcal{H}} S(R_0)$. Hence $U_0 <_{\mathcal{H}} S(R_0)$, and so $U_0 \leq_{\mathcal{H}} W_0$ by ret.3. But $W_0 <_{\mathcal{H}} R_1$ by assumption. So $V_0 <_{\mathcal{H}} U_0 <_{\mathcal{H}} R_1$, in contradiction to the assumption $consider(R_1, 0) = V_0$. ∎

3 An economical variant of Bloom protocol

The Writers in Bloom protocol are required to read a record that contains two fields: colour and message. The later field may be very long and the necessity to read long messages may slow down the writers. We are going to describe a variant of Bloom protocol which does not require the *Writers* to read each other's messages. We also reduce the number of messages read in a *READ* execution from 3 to 2, and the register that carry the messages need no longer be atomic, regularity suffices.

A register that only carry messages and no control information is called here a buffer. We need for this variant protocol four regular buffers, $B_i[0]$ and $B_i[1]$ which are written by $Writer_i$ and read by *Reader*, for $i = 0, 1$.

In any system execution $\mathcal{S}$ of the protocol, we assume a return function ω defined on the lower-level reads r in $\mathcal{S}$ as before. Only that now, for reads of the buffers $B_i[l]$, which are regular, we only assume regularity of ω. This means that for any read r

1. $Value(\omega(r)) = Value(r)$.
2. $w(r) \not>_{\mathcal{S}} r$.
3. There is no write event, w, onto $B_i[l]$ (which r is reading) such that $\omega(r) <_{\mathcal{S}} w <_{\mathcal{S}} r$.

For each i, $Writer_i$ alternates the use of buffers $B_i[0]$ and $B_i[1]$: in the kth execution of the *WRITE*, $Writer_i$, writes its message onto $B_i[p]$, where $p = k \pmod 2$.

The atomic registers R_0 and R_1 carry control information only. This is a record containing a pair $\langle t, p \rangle$ of boolean values: p is called the parity, and t the colour. The initial value of R_i is the pair $\langle 0, 1 \rangle$. The initial value of the buffers is some initial message. There is a local boolean variable, *parity*, initialized to 1.

WRITE(a : **message**); (* For $Writer_0$ *)

1. read(R_1; $\langle t, p \rangle$);
2. write($B_0[parity]$; a);
3. write(R_0; $\langle t, parity \rangle$);
4. $parity := 1 - parity$.

WRITE(a : **message**); (* For $Writer_1$ *)

1. read(R_0; $\langle t, p \rangle$);

2. $t := 1 - t$;

3. write$(B_1[parity], a)$;

4. write$(\mathsf{R}_1; \langle t, parity\rangle)$;

5. $parity := 1 - parity$.

READ

1. read$(\mathsf{R}_0; \langle t_0, p_0\rangle)$

2. read$(\mathsf{R}_1; \langle t_1, p_1\rangle)$

3. $r := \begin{cases} 0 & \text{IF } t_0 = t_1 \\ 1 & \text{IF } t_0 \neq t_1 \end{cases}$

4. read$(\mathsf{R}_r; \langle t, p\rangle)$

5. read$(B_r[p], v_0)$;

6. read$(\mathsf{R}_r; \langle t', p'\rangle)$

7. IF $p = p'$ THEN return v_0;
   ELSE

   (a) read$(B_r[p'], v_1)$;
   (b) read$(\mathsf{R}_r; \langle t'', p''\rangle)$
   (c) IF $p'' = p'$ THEN return v_1
       ELSE return v_0. (* necessarily $p'' = p$ in this case *)

As in Bloom protocol, $Writer_0$ is copying the colour observed in $Writer_1$, and $Writer_1$ is reversing the colour observed in $Writer_0$. The first three lines of the *READ* protocol resemble the corresponding lines in Bloom protocol: the reader first considers $Writer_0$ and then $Writer_1$. The determination of the return index, r, is made in line 3 as before, but now only control variables are read (in lines 1 and 2). Then either the read of one or two buffers is needed to determine the return value (as opposed to three reads in Bloom protocol). The value returned is obtained in reading $B_r[x]$ where $x = 0, 1$ is so chosen that there are two reads of register R_r, one preceding and one following the read of $B_r[x]$, so that the parity of both reads of R_r is x.

This procedure employed by the *READ* to decide which value to return is quite involved (Haldar and Vidyasankar [1992] use a somewhat similar mechanism in a different context, Construction MC4), and yet the correctness proof is easy because half of the job has already been done. We need only show that the higher level specifications of Section 2.1 hold for the variant.

3.1 The higher-level properties hold

Given a lower-level execution of the variant protocol, the higher-level view $\mathcal{H}$ is defined similarly as for the Bloom protocol by collecting lower-level read/write events. (Thus the last lower-level event in a *WRITE* is a write onto R_i and not the local change of the parity.) The modifications are the following. For a *READ*, R, the higher-level event $S(R)$ consists now of an execution of lines 4-7.

The function $colour(W)$ for a *WRITE* event is the value of variable t written on the colour field of register R_i. The *parity* of W is the parity field value as written on R_i. So, if l is the parity of W, then, in line 3, W writes onto its buffer that has index l.

The return function Ω is defined by looking, for a given *READ* R, at the "return read", s, in $S(R)$ and setting $\Omega(R) = [\omega(s)]$. This return read is determined as follows. In case $p = p'$, or $p = p''$, the return read is the read of $B_r[p]$ (in line 5). Otherwise ($p \neq p' = p''$) the return read is the read of $B_r[p']$ in line 7. The other functions are defined as before, and the only non trivial part in showing that the higher-level specifications of Section 2.1 hold is the proof that $\Omega(R) <_{\mathcal{H}} end(R)$.

To see this, let us look again at the *READ* protocol. Let R be any *READ* event, and suppose that r is the return index of R, l is the parity of $\Omega(R)$, and $s \in S(R)$ is the read of $B_r[l]$, and r_1 is the last read of register R_r in R. Then

1. $[\omega(s)] = \Omega(R)$,

2. $s <_{\mathcal{S}} r_1$, and

3. l is the parity field value of $\omega(r_1)$.

Now let w_1 be the write in $\Omega(R)$ on register R_r. We claim that $w_1 <_{\mathcal{S}} r_1$, and this clearly implies $\Omega(R) <_{\mathcal{H}} end(R)$. If this is not the case, then $r_1 <_{\mathcal{S}} w_1$ by the seriality of R_r. The parity of w_1 is l which is the parity of $\Omega(R)$. Let w_0 be the write on R_r just preceding w_1 (it cannot be the case that w_1 is the initialization write since $w_1 <_{\mathcal{S}} r_1$ in this case). So $w_0 <_{\mathcal{H}} \Omega(R)$. Then the parity of w_0 is $1 - l$. But $w_0 <_{\mathcal{S}} \omega(s) \not>_{\mathcal{S}} s <_{\mathcal{S}} r_1$ implies that $w_0 <_{\mathcal{S}} r_1$ and hence r_1 is between two successive writes w_0 and w_1 onto R_r. Hence $w_0 = \omega(r_1)$ and its parity is $1 - l$ in contradiction to 3 above.

4 Buffer protocols

As described in any Operating System course, the *Producer/Consumer* problem has a standard solution that implements a circular queue with only the most elementary means—shared variables that can be accessed for write by only one process, their owner. A particularly intriguing point in this solution is that n buffer cells are needed to creat a queue that may contain at most $n - 1$ items (when full). Not that anyone really cares about one more or one less buffer cell, but the elegance of this charming protocol is hampered by the feeling that n cells should suffice for implementing an n-place queue. Of course, semaphores and other more complex shared variables may remedy this deficiency, but we want the most primitive means. The protocol given here is an implementation of a circular queue of capacity n with exactly n buffer cells (and primitive control variables). The key to the solution is to disassociate the control component from the information produced and consumed by the two processes.

Before describing this solution, let us recall the classical *Producer/Consumer* protocol (as given by the textbook of Peterson and Silberschatz [1985]). The two processes, *Producer* and *Consumer* operate in parallel and repeat forever the following procedures:

1. The *Producer* produces an item in *nextp*, and then executes $ENQUEUE(nextp)$.

2. The *Consumer* executes $DEQUEUE(nextc)$, and consumes the item in *nextc*.

Here *nextp*, and *nextc* are local variables, in *Producer* and *Consumer*, of a specified type called 'Item'. Though it is interesting to investigate this problem under the assumption that both of the processes are allowed to stop after finitely many executions, one usually adopts the simplifying assumption of never ending executions of these procedures. This ensures that the queue will never permanently stay full or permanently stay empty.

The *ENQUEUE*, and *DEQUEUE* procedures use a shared variable, called buffer, which is an array $[0..N]$ of Item, and two shared variables in, and out which carry indices in $0..N$ (this is a Pascal-type notation for the set of integers $\{0, \ldots, N\}$, which can also be denoted with $[0, \ldots, N]$). The *Producer* writes on in, and on $\mathsf{buffer}[i]$, and the *Consumer* reads these variables. the *Consumer* writes on out, which the *Producer* reads. Intuitively, the variable in points to the next free buffer cell onto which the *Producer* is loading its item, while out points to the next cell from which the *Consumer* unloads. The pool is empty when $in = out$, and is full when $in + 1 \pmod{N+1} = out$. In this last case, buffer cell $\mathsf{buffer}[in]$ is unused, and thus N is the maximal capacity of the pool, even though $N+1$ cells are used. Initially, $in = out = 0$.

ENQUEUE (x : **Item**)

1. WHILE $out = in + 1 \pmod{N+1}$ DO skip;
2. $\mathsf{buffer}[in] := x$;
3. $in := in + 1 \pmod{N+1}$;

DEQUEUE (*VAR* x : **Item**)

1. WHILE $in = out$ DO skip;
2. $x := \mathsf{buffer}[out]$;
3. $out := out + 1 \pmod{N+1}$;

The following Subsection presents the variant protocol which uses the buffer more efficiently.

4.1 Implementation of a circular queue—the variant

There are two serial processes: *Consumer* and *Producer* which use two single-writer single-reader sequential registers: In (from the *Producer* to the *Consumer*) and Out (from the *Consumer* to the *Producer*); these registers carry values in $[0, \ldots, N]$. Initially, In and Out have value 0. The role of these registers is only to coordinate the two processes, not to point to the place of the data. We use read/write commands to access the registers. For example, write(In; in), is the command to write into register In the value of variable in.

buffer is an N-place array, which carries data values of type **Item**, and its cells are loaded by the *Producer* and unloaded by the *Consumer*.

The local variables, in, and out are in $0..N$, and are initially 0. The local variable p has values in $0..N-1$, and is initially 1. (Any initial value works, but starting with $p = 1$ simplifies some of the notations used in the proof.) The value of a local variable does not change from the end one call of the protocol to the beginning of the next. (Locality means of course that these are distinct variables when they belong to different protocols, even if they have the same name.)

ENQUEUE (x : **Item**)

1. WHILE $out = in + 1 \pmod{N+1}$ DO
   read(Out; out);
2. buffer$[p] := x$;
3. $p := p + 1 \pmod{N}$;
4. $in := in + 1 \pmod{N+1}$;
5. write(In; in)

DEQUEUE (*VAR* x : **Item**)

1. WHILE $in = out$ DO
   read(In; in);
2. $x :=$ buffer$[p]$;
3. $p := p + 1 \pmod{N}$;
4. $out := out + 1 \pmod{N+1}$;
5. write(Out, out)

Before proceeding with the formal details, let us compare this variant with the classical version. The main idea is this: The classical protocol (for a queue of capacity N) ensures that at each moment there are at least as many load events as unload events, and the excess of the loads over the unloads never exceeds N. This property is not dependent on the ways or the order that the buffer cells are approached. So, we use the classical protocol, but with a buffer of just N cells (instead of $N+1$ cells) to obtain the variant. Observe first the separation of the control and data values in the variant protocol. While in the classical protocol the same variable *in*, or *out*, is used for both controlling the WHILE loop and for denoting which data cell to access, in our version the role of these variables is restricted to coordination, and another variable, p, points to the data. While p changes in a cyclic order of length N, the values of *in*, and *out* are determined in a cycle of length $N+1$; thus very soon a discrepancy is created showing that the roles of these variables is really distinct.

The second difference between the two protocols is that the variant makes a distinction between the local variable *in* and the register In, and similarly a distinction between *out* and Out. Thus instead of line 1 in the classical protocol

$$\text{WHILE } out = in + 1 \pmod{N+1} \text{ DO skip;}$$

we write the explicit instruction

$$\text{WHILE } out = in + 1 \pmod{N+1} \text{ DO}$$
$$\text{read}(\mathsf{Out}; out);$$

This results in a more economical protocol, since there is not necessarily a need to always read the shared variable.

The difference between our variant and the classical protocol is most clearly seen in case $N = 1$, that is, when the implemented queue contains at most one item and the role of the protocol is to ensure mutual-exclusion. While two buffer cells are alternatingly written in the classical protocol, a single buffer cell suffices for the variant.

Let us call an execution of the statement

$$\mathsf{buffer}[i] := x$$

a 'load' event, and an execution of

$$x := \mathsf{buffer}[i],$$

an 'unload' event. We define the *value* of the load/unload event to be the value of the item, x, loaded or unloaded.

Both the *ENQUEUE* and the *DEQUEUE* protocols can be decomposed into three parts: (1) the *wait* part, consisting of line 1, which may be non-terminating, (2) a load/unload event, and (3) the remainder.

An execution of the protocol is called 'successful' if it passed the WHILE loop of line 1 and arrived to line 2 (and to the end of the protocol); it is 'non successful' otherwise. It can be shown that, if the processes are active forever, then all the events are successful—the WHILE loop is always finite and may last 0, 1 or any finite number of read rounds. However, it is convenient at this stage of the proof to speak about possible unsuccessful events (this approach is necessary for a more general result that does not assume that the processes are active forever).

If X is a successful *ENQUEUE* event, let us denote with $load(X)$ the load event in X. Similarly, $unload(X)$ is defined for a successful *DEQUEUE* event X.

We shall prove the correctness of the variant protocol. First, we have to specify what "correctness" means here, and we must describe the assumptions made; for example, what is demanded of the buffer cells?

Let $\mathcal{L} = (L, <_{\mathcal{L}}, F)$ and $\mathcal{H} = (H, <_{\mathcal{H}}, G)$ be a lower-level and a higher-level executions resulting from an application of the variant protocol.

The events in L are the read/write events on the registers, and the load/unload events on the buffer cells, $\mathsf{buffer}[i]$, for $0 \le i < N$. The events in H are obtained by grouping events in L as will be described below. For example, $X \in H$ in an *ENQUEUE* event if X constitute in $\mathcal{L}$ an execution of the *ENQUEUE* protocol (for some parameter x of type **Item**).

As described in the first part of this paper, for $X, Y \in H$ we set $X <_{\mathcal{H}} Y$ iff for every $x \in X$ and $y \in Y$, $x <_{\mathcal{L}} y$.

It is convenient to assume that $\mathcal{L}$ contains initial writes w_{In} and w_{Out} of value 0 onto registers In and Out, and that $Initialization_P = \{w_{\mathsf{In}}\}$, $Initialization_C = \{w_{\mathsf{Out}}\}$ are the corresponding higher-level *Initialization* events.

It is convenient to regard the lower-level load/unload events as higher-level events as well. Formally, for each lower-level load/unload event x, we put $\{x\}$ in H, but informally, we may treat the load/unload event itself as a bona fide higher-level event. Other than the initialization events, we have in H

1. the higher-level executions of the protocols *ENQUEUE* and *DEQUEUE*, and

2. the higher-level *load/unload* events (i.e., singletons $\{x\}$ where x is a *load/unload* event).

Let us enumerate the initial events and the successful higher-level *ENQUEUE* and *DEQUEUE* events (in *Producer* and *Consumer*) as $\langle E_i : i \in I_0 \rangle$, and $\langle D_j : j \in J_0 \rangle$, respectively, where I_0, J_0 can be a finite initial segment or the complete set of the natural numbers. So $E_0 = \mathit{Initialization}_P$, and $D_0 = \mathit{Initialization}_C$; and $E_i <_{\mathcal{H}} E_{i+1}$, $D_i <_{\mathcal{H}} D_{i+1}$. For $i > 0$, E_i and D_i are successful executions of the *ENQUEUE* and *DEQUEUE* protocols.

What are the properties of $\mathcal{L}$ and $\mathcal{H}$ that we want to prove in order to be convinced of the protocol's correctness? These are of two categories: safety and liveness. The safety property says that the n-th *DEQUEUE* execution, D_n, removes an item that was deposited by the n-th *ENQUEUE* execution, E_n,

Safety Property: $\mathit{load}(E_n) <_{\mathcal{H}} \mathit{unload}(D_n)$ and both have the same value.

The liveness property says that E_n and D_n exist for every index n, i.e., that all the WHILE loops in the protocol are successful (there are stronger liveness properties but we choose the simplest one). We shall leave the proof of the liveness property to the interested reader.

The Safety Property is a consequence of the three conditions which are described below, the cyclic order, the safeness of the buffer cells, and the conflict freedom.

The cyclic $\pmod N$ approach to the buffer cells is the following property: *The load events of the buffer cells are linearly ordered by* $<_{\mathcal{H}}$ *and, beginning with* $\mathsf{buffer}[0]$, $\mathsf{buffer}[j]$, *for* $0 \le j < N$, *are approached cyclicly* $\pmod N$, *and similarly the unload events are serially ordered and approach cyclicly the cells* $\mathsf{buffer}[k]$, *beginning with* $\mathsf{buffer}[0]$.

Let us restate this property in formal terms. Let L_i and U_i be the i-th *load* and i-th *unload* events respectively. That is $L_i = \mathit{load}(E_i)$, and $U_i = \mathit{unload}(D_i)$. The cyclic order is thus the requirement that

1. L_i is loading $\mathsf{buffer}[k]$, and U_i is unloading $\mathsf{buffer}[k]$, where $k = i \pmod N$.

2. The processes are serial: $L_1 <_{\mathcal{H}} L_2 \ldots$, and $U_1 <_{\mathcal{H}} U_2 \ldots$

The cyclic order is an obvious property of the protocol which increases variable p by 1 $\pmod N$ in each execution.

The *safeness* of a buffer cell, $\mathsf{buffer}[i]$, is the following property of the system S (a collection of system executions): *For every system execution $\mathcal{S} \in S$, if $\mathsf{buffer}[i]$ is conflict free in $\mathcal{S}$ then the value of the k th unload event on $\mathsf{buffer}[i]$ equals the value of the k-th unload event on that buffer.* (Thus no claim against the manufacturer of the buffer cell can be made if the malfunctioning occurs in a system execution in which the buffer cell is not conflict free.) The safeness assumption made here for each $\mathsf{buffer}[i]$ is the 'hardware' assumption needed to ensure the correctness of the protocol.

The last property needed for safety is the following. Let us say that the buffer array buffer is *conflict free* in system execution $\mathcal{S}$ iff for each $0 \leq l < N$, The load and unload operation executions on $\mathsf{buffer}[l]$ are serially ordered by $<_{\mathcal{S}}$, alternating load with unload and beginning with a load event. (Let us emphasize that this does not exclude the possibility that for different indices a load of $\mathsf{buffer}[l]$ can be executed concurrently with an unload of some $\mathsf{buffer}[j]$.)

In fact, if the cyclic ($\pmod N$) property is assumed, then the conflict freedom is equivalent to the following:

1. For every index i for which U_i is defined, L_i is defined and $L_i <_{\mathcal{H}} U_i$.
2. For every index, i, for which L_{i+N} is defined, U_i is defined and $U_i <_{\mathcal{H}} L_{i+N}$.

It is easy to see that these three properties, the cyclic order of approaching the buffer cells by each of the processes, the safeness of the buffer cells, and the conflict freedom of the buffer suffice to derive the Safety Property formulated above. Indeed, the cyclic order implies that L_n is the l-th load event of $\mathsf{buffer}[k]$, where $n = N(l-1) + k$. Similarly, U_n is the l-th unload event of $\mathsf{buffer}[k]$ for the same l and k. As $\mathsf{buffer}[k]$ is assumed to be conflict free, the l-th load event always precedes the l-th unload event and their values are the same.

Since the cyclic order is obvious and the safeness assumption is explicitly made, what remains to be seen is the conflict freedom of the buffer. It is by coordinating the load/unload operations that the protocol ensures the conflict freedom property as we shall prove in the following two subsections. For that, we shall first define the function *activate* on the successful *DEQUEUE* and *ENQUEUE* events in $\mathcal{H}$ and study its properties.

4.2 The function *activate* and its Basic Specifications

Recall that $\mathcal{L} = (L, <_{\mathcal{L}}, F)$ and $\mathcal{H} = (H, <_{\mathcal{H}}, G)$ are a lower-level and a higher-level executions resulting from an application of the variant protocol.

Let $\omega \in F$ be the function that assigns to every read, r, of In or Out the corresponding write $\omega(r)$ which is the $<_{\mathcal{L}}$ rightmost such that $\omega(r) <_{\mathcal{L}} r$. Seriality of these registers implies that the value of r equals the value of $\omega(r)$.

Definition 4.1 *1. For any event $X \in \mathcal{H}$, and variable v local to the process executing X, let us denote with $v(X)$ the value of v at the end of X. We assume that there is a function in G that associates X with $v(X)$.*

2. The data value, Value(E), of a successful event E, is the value $x(E)$ of the Item loaded or unloaded in the buffer *(line 2).*

3. Let $X \in H$ be a successful DEQUEUE or ENQUEUE event. If the WHILE *loop of X is non empty, then we call the last read in the loop, of register* In *or* Out*, the 'successful read'. The successful read is thus the read in line* 1 *which ends the busy WHILE wait of the successful event. (There are successful X's for which the condition in the WHILE loop does not hold, and no read is necessary for their completion. This is a slight advantage of our protocol over the classical protocol which requires that the in and out shared variables are accessed at each DEQUEUE and ENQUEUE operation.)*

Definition 4.2 *1. The function activate(X) is inductively defined on the successful events in H as follows. For the Initialization events set: activate$(E_0) = D_0$ and activate$(D_0) = E_0$. For the first ENQUEUE event: activate$(E_1) = D_0$.*

Now, if X is a successful ENQUEUE or DEQUEUE, and X contains a successful read, r, let activates$(X) = [\omega(r)]$. (As explained in Section 2.1, $[x]$ denotes the (unique) higher-level event that includes x.)

If the successful event X contains no read of In *or* Out*, and X is not an Initialization event, then $X = E_i$ or $X = D_i$ for $i > 0$, and correspondingly we define in this case*

$$activate(X) = activate E_{i-1}, \text{ or } activate(X) = activate(D_{i-1}).$$

The higher-level *load*/*unload* events, and the functions with these same names were already defined in the previous Subsection. We will deal with these events only in the following Subsection, and there we will use the following obvious properties:

Lemma 4.3

1. *If $E_m = activate(D_i)$, then $load(E_m) <_{\mathcal{H}} unload(D_i)$.*
2. *If $D_n = activate(E_k)$, and $n \neq 0$, then $unload(D_n) <_{\mathcal{H}} load(E_k)$.*

Notation 4.4 *For any integer k let $[k]_{N+1}$ be that integer, j, in $[0, \ldots, N]$ such that $j = k \pmod{N+1}$.*

We will be using some obvious arithmetics mod $N+1$. For example, $[x+1]_{N+1} = ([x]_{N+1}) + 1$, if $[x]_{N+1} \neq N$ (or equivalently, if $[x+1]_{N+1} \neq 0$).

Lemma 4.5 1. *$in(E_0) = 0$; $in(D_0) = 0$. And for any E_i*

$$in(E_i) = i \pmod{N+1}.$$

For any D_i

$$out(D_j) = j \pmod{N+1}.$$

2. *Let D and E be a DEQUEUE and ENQUEUE successful events:*

$$\text{If } E = activate(D), \text{ then } in(E) = in(D).$$

$$\text{If } D = activate(E), \text{ then } out(E) = out(D).$$

3. *If D is a successful DEQUEUE, then*

$$in(D) \neq out(D) - 1 \pmod{N+1}.$$

If E is a successful ENQUEUE, then

$$out(E) \neq in(E) \pmod{N+1}.$$

The proof is quite easy and is not given in details. It is proved by induction. If, for example, $E = activate(D)$, then $in(E) = in(D)$ is proved as follows. If D contains a read of In, then it contains a last read of In, and then the value obtained in this read is the value of $in(D)$ which is thus the value written by E, namely $in(E)$. If D contains no read, then $D = D_i$ and $E = activate(D_{i-1})$ by definition. By the inductive assumption, $in(E) = in(D_{i-1})$. But $in(D_i) = in(D_{i-1})$ since the variable in is changed only by reading register In.

Claim 4.6 *[Basic Specifications] The function activate satisfies the following.*

1. *For $i > 0$, if $D_j = activate(E_i)$, then $[i-j]_{N+1} \neq 0$.*

2. *If $E_i = activate(D_j)$, then $[i-j]_{N+1} \neq N$*

3. *$D_0 = activate(E_0)$. $E_0 = activate(D_0)$. But for $j > 0$, $activate(activate(D_j))$ is some D_n for $n < j$. Similarly, $activate(activate(E_j))$ is some E_n for $n < j$.*

4. *activate is monotonic: If $i < i'$ then $activate(E_i) \leq_{\mathcal{H}} activate(E_{i'})$, and $activate(D_i) \leq_{\mathcal{H}} activate(D_{i'})$.*

5. *(The jump:)*

   - *If $E_k = activate(D_{j-1}) <_{\mathcal{H}} activate(D_j)$, then*
   $$k = j - 1 \pmod{N+1}.$$

   - *If $D_k = activate(E_{i-1}) <_{\mathcal{H}} activate(E_i)$, then*
   $$k = i \pmod{N+1}.$$

The proof of this Claim is simple and involves no deep issues of concurrency. To prove *1*, use Lemma 4.5(2) to deduce that $out(E_i) = out(D_j)$. Now, as E_i is a successful *ENQUEUE* event, $out(E_i) \neq in(E_i) \pmod{N+1}$. Thus $in(E_i) \neq out(D_j) \pmod{N+1}$), and by Lemma 4.5(1) $[i-j]_{N+1} \neq 0$ follows, because $out(D_j) = j$ and $in(E_i) = i \pmod{N+1}$.

Item *2* is similarly proved (show that $in(E_i) = out(D_{j-1})$).

Items *3* and *4* are easy (because the write on the registers In and Out is the last event in any execution of the protocol). Let us prove the first item in *5*. Suppose $E_k = activate(D_{j-1}) <_{\mathcal{H}} activate(D_j)$. Since $activate(D_j) \neq activate(D_{j-1})$, it must be that $activate(D_j)$ was defined to be $[\omega(r)]$ for the successful read r in D_j. The existence of r implies of course that the (last) WHILE loop was entered with condition ($in = out$) being true in D_j. But in entering this loop, the value of the variables in and out is the value determined at the previous *DEQUEUE* event, namely in D_{j-1}, and hence $in(D_{j-1}) = out(D_{j-1})$. From Lemma 4.5(2), $in(E_k) = in(D_{j-1})$, and hence

$$out(D_{j-1}) = in(E_k).$$

Now $out(D_{j-1}) = [j-1]_{N+1}$, and $in(E_k) = [k]_{N+1}$. Hence $k = j - 1 \pmod{N+1}$.

Lemma 4.7 (Main Lemma) *Assume the Basic Specifications for a function called activate. If $E_i = activate(D_j)$ or $D_j = activate(E_i)$ then*

$$0 \leq i - j \leq N.$$

Proof. We emphasize that the only assumptions used in this Lemma are that the increasing sequences E_i and D_i satisfy the specifications of Claim 4.6 which are assumed to hold in $\mathcal{H}$. For the proof, it does not matter whether $\mathcal{H}$ is or is not obtained by executing the protocol.

The proof of the Lemma is by induction on the ordered pair $\langle i, j\rangle$ (in the lexicographical order). By item 3, $activate(E_0) = D_0$ and $activate(D_0) = E_0$. There are two cases to consider:

Case 1: $E_i = activate(D_j)$.
In this case (by Basic Specifications 4.6 (*2*))

$$[i-j]_{N+1} \neq N. \tag{0}$$

As said above, if $j = 0$ then $i = 0$ and we are done, so assume $j > 0$. By monotonicity of *activate*, $activate(D_{j-1}) \leq_{\mathcal{H}} activate(D_j)$. So there are two subcases: equality, and strict precedence. Assume first equality:
Subcase 1.1: $activate(D_{j-1}) = E_i$. Then, by the inductive assumption

$$0 \leq i - (j-1) \leq N. \tag{1}$$

$0 = i - j + 1$ is not possible, since in this case $[i-j]_{N+1} = N$ contradicts equation (0). Hence $0 \leq i - j < N$ follows from (1) as required.
Subcase 1.2: $activate(D_{j-1}) <_{\mathcal{H}} activate(D_j)$. Put $activate(D_{j-1}) = E_k$. By monotonicity of *activate*, as $activate(D_j) = E_i$,

$$k < i$$

and there is a jump. Hence by our basic specification (*5*),

$$k = j - 1 \pmod{N+1}. \tag{2}$$

The inductive assumption, applied to $\langle k, j-1 \rangle$, implies $0 \leq k - (j-1) \leq N$. Together with Equation (2) this gives that

$$k - (j-1) = 0. \tag{3}$$

Let $l \geq 1$ be such that $i = k + l$. We Claim that $l \leq N$.
Proof: Assume $l > N$. Then $k+N+1 \leq i$ and $E = E_{k+N+1}$ exists. Let $D_m = activate(E)$. Then $m \leq j-1$ is proved as follows. By item *3*, $activate(E_i) = D_n$ for some $n < j$. But by monotonicity $activate(E) \leq_{\mathcal{H}} activate(E_i))$. By the inductive assumption applied to $\langle k+N+1, m \rangle$,

$$(k + N + 1) - m \leq N. \tag{4}$$

Yet, as $m \leq j - 1$, $k - m \geq k - (j-1) = 0$, in contradiction to equation (4) which implies $k - m \leq -1$. So we have proven the claim:

$$1 \leq l \leq N. \tag{5}$$

We now have

$$i - j = k + l - j = k - (j-1) + l - 1 = l - 1.$$

Since $0 \leq l - 1 < N$, by equation (5), $0 \leq i - j < N$ as required.

Case 2: $D_j = activate(E_i)$. The proof is so similar to **Case 1** that we only outline its main steps. The interesting situation, corresponding to **Subcase 1.2** is that

$$D_k = activates(E_{i-1}) <_{\mathcal{H}} activates(E_i).$$

Then $0 \leq i-1-k \leq N$ by the inductive assumption, and $k = i \pmod{N+1}$, since there is a jump. Hence

$$i - k = N + 1.$$

Put $k + l = j$, then (as $k < j$) $1 \leq l$, and we must show that $l \leq N + 1$ in order to deduce the desired inequality: $0 \leq i - j \leq N$.

Assume $l > N+1$, and then $k+N+1 < j$. Then $activate(D_{k+N+1}) = E_{i_0}$ for $i_0 < i$. Apply induction to get that

$$0 \leq i_0 - (k + N + 1).$$

But $i_0 - (k + N + 1) < i - k - (N + 1) = 0$.

In the following Section we use the Main Lemma and show how it implies the conflict freedom required to prove the safety property of the implemented queue.

4.3 Proof of the safety property

Recall that in order to prove the safety property it suffices to ensure the conflict freedom properties, as the cyclic order is obvious and the safeness of the buffer is assumed (see Section 4.1).

Let D_i be any *DEQUEUE* event, we will prove that $load(E_i) <_{\mathcal{H}} unload(E_i)$. Let $E_m = activate(D_i)$. Then $0 \leq m - i$, by the Main Lemma. Thus $i \leq m$, and $E_i \leq_{\mathcal{H}} E_m$. But, as we observed in Lemma 4.3, $load(E_m) <_{\mathcal{H}} unload(D_i)$, and hence $load(E_i) <_{\mathcal{H}} unload(D_i)$ follows.

It remains to show that if E_{i+N} is defined (i.e., $i + N \in I_0$), then D_i is defined and $unload(D_i) <_{\mathcal{H}} load(E_{i+N})$. For this, set $D_n = activate(E_{i+N})$. Then (by the Main Lemma) $i + N - n \leq N$, and hence $i \leq n$. So $D_i \leq_{\mathcal{H}} D_n$. But (by Lemma 4.3) $unload(D_n) <_{\mathcal{H}} load(E_{i+N})$, and hence $unload(D_i) <_{\mathcal{H}} load(E_{i+N})$.

5 Conclusion

The aim of this note was to present a new approach to proving the correctness of protocols. In this modular approach, system executions at different levels are used to organize the proof in a way that confines the protocol to one end and the useful properties that one wishes to prove to the other end (the lower and higher-level views).

The 'behavioral' approach (and ours could probably be characterize as such) has been (justly) criticized for being prone to mistakes. For example, L. Lamport and F. B. Schneider write ([1985] p. 203):

"Most computer scientists find it natural to reason about a concurrent program in terms of its behavior—the sequence of events generated by its execution. Experience has taught us that such reasoning is not reliable; we have seen too many convincing proofs of incorrect algorithms. This has led to assertional proof methods, in which one reasons about the program's state instead of its behavior. Unlike behavioral reasoning, assertional proofs can be formalized—i.e., reduced to a series of precise steps that can, in principle, be machine-verified."

We agree that it is natural to reason in terms of behavior: this type of reasoning can also use diagrams, and is easier to communicate. However, we think that, though there is no guarantee against mistakes, natural *and* reliable reasoning about concurrent programs can be established in terms of an analysis of behavior. We believe that behavioral reasoning may be formalized in a way that does not forsake intuition, and we propose our approach as a step in this direction.

References

U. Abraham

. *On interprocess communication and the implementation of a multi-writer atomic register* unpublished.

U. Abraham and S. Ben-David and M. Magidor

1990 *On Global-Time and Inter-process Communication.* Semantics for Concurrency, Proceedings of the International BCS-FACS Workshop, pp. 311 - 323.

Anger

1989 *On Lamport's interprocess communication.* ACM Transactions on Programming Languages and Systems, Vol. 11 No. 3, pp. 404 - 417.

B. Bloom

1987 *Constructing two-writer atomic registers.* 6th ACM Symp. on Principles of Distributed Computing, pp. 249 - 259.

S. Haldar and K. Vidyasankar
1992 *Simple extensions of 1-writer atomic variable constructions to multi-writer ones.* Tech. Report 9201 (Memorial University of Newfoundland).

L. Lamport
1986 *On Interprocess Communication, Part I: Basic formalism, Part II: Algorithms.* Distributed Computing, Vol. 1, pp. 77 - 101.

L. Lamport(Section 2.2) L. Lamport and F. B. Schneider(Chapter 5)In:
1985 *Distributed systems.* M. Paul and H. J. Siegert (eds) Lecture Notes in Computer Science 190, Springer.

G.L Peterson and J.E. Burns
1987 *Concurrent reading while writing* **II** *: the multiwriter case.* FOCS-87.

J. L. Peterson and A. Silberschatz
1985 *Operating System Concepts* Addison-Wesley, second edition.

V. Pratt
1986 *Modeling Concurrency with Partial Orders.* International J. of Parallel Programming, Vol.15, pp. 33 - 71.

N. Wiener
1914 *A contribution to the theory of relative position.* Proc. Camb. Philos. Soc. 17, pp.441-449.