The Sliding-Window Protocol in CSP

by

K. Paliwoda

and

J.W. Sanders

Oxford University Computing Laboratory

Programming Research Group

The Sliding-Window Protocol in CSP

by

K. Paliwoda

and

J.W. Sanders

Technical Monograph PRG-66
ISBN 0-902928-48-1
March 1988

Oxford University Computing Laboratory
Programming Research Group
8-11 Keble Road
Oxford OX1 3QD
England

Oxford University Computing Laboratory
Programming Research Group
8-11 Keble Road
Oxford OX1 3QD
England

The Sliding-Window Protocol in CSP

K. Paliwoda and J.W. Sanders
Oxford University Computing Laboratory
Programming Research Group
8-11 Keble Road
Oxford, OX1 3QD

Abstract

A formal specification and proof of correctness is given of the sliding-window protocol using the notation of Communicating Sequential Processes. First the stop-and-wait protocol is defined; its correctness, that it forms a 1-place buffer, is almost evident. Next the alternating-bit protocol is defined and described in terms of the stop-and-wait protocol, and its correctness deduced. Finally the sliding-window protocol is described in terms of the alternating-bit protocol and its correctness deduced accordingly. The protocols are refined, and implemented in occam.

The paper has two thrusts: that modularity of a specification helps to structure proofs about it (in this case, proofs that the protocols implement buffers); and that refinement in CSP leads to structured, correct implementations in occam.

1 Introduction

There is, by now, a substantial repertoire of high-level specifications ranging from editors to transaction-processing systems. Some of these have even been implemented! But there appears to remain a need for intermediate-level specifications which are clear, which can be seen easily to meet their more high-level specifications, and which admit straightforward refinements. The difficulty involved in such examples lies in the trade-off between the description of a design and the statement of properties it is required to have.

In this paper we describe the sliding-window protocol (SWP), prove that it meets the higher-level specification of a buffer, and implement it in occam. The description, and hence the proof of correctness, is structured in terms of the simpler alternating-bit protocol (ABP) which in turn is expressed in terms of the stop-and-wait protocol (SAWP). We have aimed to structure the specifications to meet the criteria imposed in the previous paragraph. Thus once the protocols have been proved correct, the correctness of their occam implementations can be asserted almost mechanically.

The outline of the paper is as follows. Sections 2 contains notation and elementary properties of it. Section 3 contains the SAWP: the description and proof that it forms a 1-place buffer; and the ABP: description, proof (structured in terms of that for the SAWP) that it forms a 1-place buffer, and implementation in occam. Section 4 contains the specification of the SWP (structured in terms of the ABP), the proof that it forms a buffer, and a refinement and implementation in occam.

2 Notation

We use the notation of CSP, mostly as in [Hoare 85]. The processes we consider participate (with a single exception) in only communication events and so we describe their alphabets by depicting their channels. As usual we identify a channel name with the sequence of values which has been carried by that channel, and use the trace semantics of CSP to identify a process with the strongest predicate which holds between its channel names. In doing this we write $P[b/a]$ for predicate P with b substituted for free variable a. When process P refines Q (either a process or a predicate) we write

$$P \text{ sat } Q.$$

The process combinators of CSP are as in [Hoare 85] except that we write

$$P \lhd A \rhd Q$$

for the conditional statement which is the process P if predicate A is true and otherwise equals process Q. Recall that $\setminus$ denotes abstraction and that $*$ denotes the while construct.

We write square brackets (instead of angled brackets) in sequence comprehension: $[\,]$, $[x]$ and $[x]^\frown t$ denotes repectively the empty sequence, the singleton sequence containing x, and the sequence with head x and tail t; in the latter case we also write $first([x]^\frown t) = x$. All sequences are finite and the length of t is denoted $\#t$.

The function, *squash*, on sequences which compresses adjacent duplicates is

$$\begin{aligned} squash[\,] &= [\,] \\ squash[x] &= [x] \\ squash([x]^\frown t) &= squash\ t \quad \text{if } x = first\ t \\ squash([x]^\frown t) &= [x]^\frown (squash\ t) \quad \text{if } x \neq first\ t \quad \text{where} t \neq [\,]. \end{aligned}$$

We use the following relations on sequences:

$s \leq t$ iff s is a prefix of t
$s \leq^n t$ iff s is a prefix of t at most n elements shorter
$s \trianglelefteq t$ iff s is a (not necessarily contiguous) subsequence of t.

Finally $\ominus$ denotes difference modulo 2, and $\oplus_w$ denotes addition modulo the natural number w.

3 The stop-and-wait and alternating-bit protocols

3.1 Buffers

A buffer is a process with input channel *in* and output channel *out*, which accepts data on *in* and later delivers it on *out*.

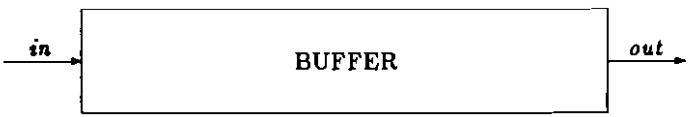

Figure 1: A buffer.

It does not corrupt, create nor lose data, but imposes only a delay during its transmission. In reality a buffer is almost certainly not just a wire, but some lower-level protocol which detects and discards corrupted messages. The way it does this (using headers, checkbits and so on) is of no concern at our present level of abstraction. For the purpose of describing a buffer, its internal workings are completely immaterial; we specify it by the way it relates the input sequence to the output sequence. Since the output sequence is a prefix of the input sequence, we write

$$Buffer(in, out) \;\widehat{=}\; out \leq in.$$

For a natural number n, an n-place buffer is a buffer with capacity n:

$$nBuffer(in, out) \;\widehat{=}\; out \leq^{n} in.$$

We shall see that the three protocols to be studied in this paper are all correct in the sense that they form buffers of finite capacity.

3.2 The stop-and-wait protocol

A buffer may be implemented using a sender (S) and a receiver (R) at either end of an intervening medium. If the sender transmits faster than the receiver is able to accept, data accumulates in the medium, and some sort of flow control is necessary. The simplest protocol guaranteeing such flow control is the stop-and-wait protocol (SAWP).

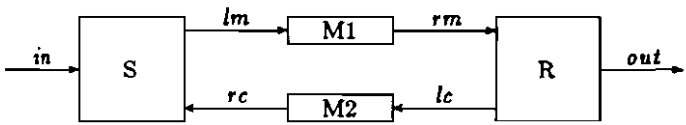

Figure 2: The stop-and-wait protocol.

The sender accepts input, relays it to the medium on channel lm and awaits acknowledgement on channel rc; when the acknowledgement arrives the sender repeats this cycle. The receiver accepts input from the medium on channel rm, outputs it, and acknowledges on channel lc. It seems evident, and we shall prove it in the corollary to follow, that the value used for acknowledgement is immaterial; for convenience we suppose that the acknowledgement consists of the piece of data which is being acknowledged. Thus in CSP we have

$$\begin{aligned} \mathrm{S} &= in?x \rightarrow lm!x \rightarrow rc?x \rightarrow \mathrm{S} \\ \mathrm{R} &= rm?x \rightarrow out!x \rightarrow lc!x \rightarrow \mathrm{R}. \end{aligned}$$

The medium is represented by two processes, M1 and M2. This is necessary because messages and acknowledgements flow in opposite directions and therefore have to be communicated via different logical channels. However we suppose that M1 and M2 are buffers:

$$\begin{aligned} &\mathrm{M1} \ \ \mathbf{sat} \ \ \mathit{Buffer}[lm/in, rm/out] \\ &\mathrm{M2} \ \ \mathbf{sat} \ \ \mathit{Buffer}[lc/in, rc/out]. \end{aligned}$$

The SAWP is the parallel combination of these four processes with their four intermediate channels concealed

$$\mathrm{SAWP} \ \widehat{=} \ (\mathrm{S} \,||\, \mathrm{M1} \,||\, \mathrm{M2} \,||\, \mathrm{R}) \backslash \{lm, rm, lc, rc\}.$$

The statement of its correctness is:

Theorem 1. SAWP **sat** *1Buffer*(in, out).

Proof. From the definitions (where stacked predicates are conjoined) we have

$$\begin{array}{lll} \mathrm{S} & \textbf{sat} & lm \leq^1 in \\ & & rc \leq^1 lm \\ & & rc \leq^1 in, \end{array}$$

$$\begin{array}{lll} \mathrm{R} & \textbf{sat} & out \leq^1 rm \\ & & lc \leq^1 out \\ & & lc \leq^1 rm, \end{array}$$

$$\mathrm{M1} \quad \textbf{sat} \quad rm \leq lm,$$

$$\mathrm{M2} \quad \textbf{sat} \quad rc \leq lc.$$

Thus

$$\begin{array}{lll} \mathrm{SAWP} & \textbf{sat} & rc \leq lc \leq^1 out \leq^1 rm \leq lm \leq^1 in \\ & & rc \leq^1 in \end{array}$$

and so $out \leq^1 in$. □

The proof of the theorem can be strengthened to permit any acknowledgement value in the definition above; indeed it is this more general form which we shall need in subsequent sections. This is the content of Corollary 1.

From now on we shall omit the free *Buffer* variables when no confusion can arise.

Corollary 1. The SAWP with arbitrary acknowledgements satisfies *1Buffer*.

Proof. The more liberal sender and receiver are

$$\begin{array}{lcl} \mathrm{S}_{\heartsuit} & = & in?x \rightarrow lm!x \rightarrow rc?x \rightarrow \mathrm{S}_{\heartsuit} \\ \mathrm{R}_{\heartsuit} & = & rm?x \rightarrow out!x \rightarrow lc!\heartsuit \rightarrow \sqcap_{\heartsuit} \mathrm{R}_{\heartsuit}, \end{array}$$

where the nondeterministic choice is over all acknowledgement values $\heartsuit$.

Then, as in the theorem but writing $k \leq^1 l$ for $0 \leq l - k \leq 1$,

$$\begin{array}{lll} \mathrm{S}_\triangledown & \textbf{sat} & lm \leq^1 in \\ & & \#rc \leq^1 \#lm \\ & & \#rc \leq^1 \#in, \end{array}$$

$$\begin{array}{lll} \mathrm{R}_\triangledown & \textbf{sat} & out \leq^1 rm \\ & & \#lc \leq^1 \#out \\ & & \#lc \leq^1 \#rm. \end{array}$$

So defining $\mathrm{SAWP}_\triangledown$,

$$\mathrm{SAWP}_\triangledown \;\widehat{=}\; (\mathrm{S}_\triangledown \,||\, \mathrm{M1} \,||\, \mathrm{M2} \,||\, \mathrm{R}_\triangledown) \backslash \{lm, rm, lc, rc\},$$

we have

$$\begin{array}{lll} \mathrm{SAWP}_\triangledown & \textbf{sat} & out \leq^1 rm \leq lm \leq^1 in \\ & & \#rc \leq^1 \#in \\ & & \#rc \leq \#lc \leq^1 \#out \leq \#rm \leq \#lm \leq^1 \#in. \end{array}$$

Thus

$$\begin{array}{l} out \leq in \\ \#out \leq^1 \#in, \end{array}$$

and the result follows. □

3.3 The alternating-bit protocol

The alternating bit protocol (ABP) ensures safe transmission of messages via a medium which sometimes loses but never corrupts them. Again it consists of a sender and a receiver working together as follows: having sent a message, the sender awaits an acknowledgement. If the message gets through to the receiver it will be output and acknowledged. If this acknowledgement reaches the sender then the next message is sent; however if either the message or the acknowledgement is lost then the sender times out and sends a duplicate. Messages are tagged by alternating bits to ensure that the receiver can distinguish two consecutive but identical messages from a message and its retransmitted duplicate.

If we ignore timeout (for the moment!), the choice between retransmitting and awaiting an acknowledgement is a nondeterministic one. In CSP this nondeterminism is expressed:

$$
\begin{array}{rcl}
\mathrm{S} & = & \mathrm{S}_0 \\
\mathrm{S}_b & = & in?x \rightarrow lm!x.b \rightarrow \mathrm{S}_{x.b} \qquad \text{where } b \in \{0,1\} \\
\mathrm{S}_{x.b} & = & rc?a \rightarrow (\mathrm{S}_{1\ominus b} \lhd a = b \rhd \mathrm{S}_{x.b}) \\
 & & \sqcap\ lm!x.b \rightarrow \mathrm{S}_{x.b}
\end{array}
$$

$$
\begin{array}{rcl}
\mathrm{R} & = & \mathrm{R}_1 \\
\mathrm{R}_b & = & rm?x.c \rightarrow\ (out!x \rightarrow lc!c \rightarrow \mathrm{R}_c \\
 & & \qquad \lhd c \neq b \rhd \\
 & & \qquad lc!b \rightarrow \mathrm{R}_b) \qquad \text{where } b \in \{0,1\}.
\end{array}
$$

This time we suppose that the media M1 and M2 are worse than for the SAWP: they may not only delay data, but possibly lose some. Thus the sequence of data on their output channel is a subsequence of the data on their input channel

$$
\begin{array}{rcl}
rm & \trianglelefteq & lm \\
rc & \trianglelefteq & lc.
\end{array}
$$

As before, we now define

$$\mathrm{ABP} \ \widehat{=}\ (\mathrm{S} \,\|\, \mathrm{M1} \,\|\, \mathrm{M2} \,\|\, \mathrm{R}) \backslash \{lm, rm, lc, rc\}$$

and summarize its correctness in

Theorem 2. ABP sat *1Buffer*.

Proof. We show that the ABP, with some changes of variable, forms a SAWP; since the processes in Theorem 1 are determined (in the trace semantics) by the predicates given there, the result follows. To define the changes of variable we use the following two further pieces of notation. A sequence of bits is alternating if it is empty or starts with 0 and adjacent elements are distinct

$$
\begin{array}{rcl}
alternating(s) & \widehat{=} & s \neq [\,] \ \Rightarrow\ (first\ s = 0 \\
 & & \qquad\qquad i, i+1 \in \mathrm{dom}\, s \Rightarrow s(i) \neq s(i+1)).
\end{array}
$$

The projection functions $projection_1$ and $projection_2$ map an augmented datum $x.b$ to its first and second coordinates, x and b, respectively. These are lifted to sequences of augmented data to give functions p_1 and p_2, defined (for $j = 1, 2$)

$$p_j(s) \ \widehat{=}\ s \fatsemi projection_j.$$

We show, applying the replacement and squash notation from section 2 to the trace semantics of the processes, that

$$
\begin{array}{l}
(\ S[p_1(squash(lm))/lm] \\
\|\ R[p_1(squash(rm))/rm, squash(lc)/lc] \\
\|\ M1[squash(lm)/lm] \\
\|\ M2[squash(rc)/rc] \\
)\ \backslash\{lm, rm, lc, rc\}
\end{array}
$$

satisfies $\mathrm{SAWP}_{\triangledown}$.

Straight from the process definitions (and again using stacking for conjunction),

$$
\begin{array}{lll}
\mathrm{S} & \mathbf{sat} & p_1(squash(lm)) \leq^1 in \\
 & & alternating(squash(p_2(lm))) \\
 & & \#rc \leq^1 \#squash(p_2(lm)) \\
 & & \#rc \leq^1 \#in,
\end{array}
$$

$$
\begin{array}{lll}
\mathrm{R} & \mathbf{sat} & out \leq^1 p_1(squash(rm)) \\
 & & alternating(squash(p_2(rm))) \\
 & & \#squash(lc) \leq^1 \#out \\
 & & squash(lc) \leq^1 squash(p_2(rm)),
\end{array}
$$

$$
\mathrm{M1}\ \ \mathbf{sat}\ \ rm \trianglelefteq lm,
$$

$$
\mathrm{M2}\ \ \mathbf{sat}\ \ rc \trianglelefteq lc.
$$

Evidently the conjuncts involving predicate *alternating* are true so

$$
\begin{array}{rll}
\mathrm{S}[p_1(squash(lm))/lm] & \mathbf{sat} & \mathrm{S}_{\triangledown} \\
\mathrm{R}[p_1(squash(rm))/rm] & \mathbf{sat} & \mathrm{R}_{\triangledown},
\end{array}
$$

so it remains to show

$$
\begin{array}{l}
squash(rm) \leq squash(lm) \\
squash(rc) \leq squash(lc).
\end{array}
$$

These are similar so we prove only the first. None of the four processes changes x without changing b, so it suffices to show

$$
squash(p_2(rm)) \leq squash(p_2(lm)).
$$

But from the definition of M1

$$\#squash(rm) \leq \#squash(lm),$$

and from S and R

$$alternating(squash(p_2(lm))) \wedge alternating(squash(p_2(rm))).$$

The desired inequality follows. □

Notes. If the sender resolves always to await an acknowledgement then the ABP deadlocks; this unsatisfactory state of affairs leads us to the refinement in tbe next section.

If either medium resolves never to pass on data then the ABP (or any other protocol for that matter) fails; however the previous result can be interpreted as showing that this is not the fault of the ABP which interposes a delay of at most one between input and output. It thus follows that if the sequence of inputs tends to infinity in length then so too does the sequence of outputs.

3.4 Refinement of the ABP

In order to implement the ABP we must refine it to remove nondeterminism in the sender and the media; the receiver is deterministic and need be refined no further.

Consider first the sender. At first glance the occurrence of a timeout in the sender seems to require the use of timed CSP . However we will be able to prove the correctness of the protocol using untimed CSP by viewing the length of the timeout to affect the efficiency rather than the correctness of the protocol.

We let ◯ denote a time-out event. It belongs to the alphabet of the sender alone and may happen only as long as an acknowledgement is expected but has not yet been received. (This makes ◯ quite different from the interrupt events described in [Hoare 85], pp.180 ff. which also belong to the alphabet of the environment and may occur at any time irrespective of the occurrence of any events in the process which is to be interrupted.)

The sender is refined, once the new event ◯ is concealed, by the deterministic process

$$\begin{array}{lcl} S2 & = & S_0 \\ S_b & = & in?x \rightarrow lm!x.b \rightarrow S_{x.b} \\ S_{x.b} & = & rc?a \rightarrow (S_{1\ominus b} \lhd a = b \rhd S_{x.b}) \\ & & \mid \bigcirc \rightarrow lm!x.b \rightarrow S_{x.b}. \end{array}$$

Proof that S2 \{○} refines the previous sender S is immediate from the law L9 [Hoare 85], p.113.

In order to simulate operation of the ABP in occam, we also refine the medium. Let us first refine the specification of M1 (and similarly M2) by a process which, after having accepted a message on its left channel, either outputs this message on its right channel or waits for another message on its left channel. It may ignore up to k-1 messages in this way, but the k^{th} message must be output on its right channel

$$\begin{aligned} \mathrm{M} &= \mathit{left}?x \rightarrow \mathrm{M}_{2,x} \\ \mathrm{M}_{n,x} &= (\mathit{right}!x \rightarrow \mathrm{M}) \sqcap (\mathit{left}?y \rightarrow \mathrm{M}_{n+1,y}) \quad \text{for } 1 < n < k \\ \mathrm{M}_{k,x} &= \mathit{right}!x \rightarrow \mathrm{M}. \end{aligned}$$

Evidently M satisfies, as we supposed in the previous section,

$$\mathit{right} \trianglelefteq \mathit{left}.$$

We now set

$$\begin{aligned} \mathrm{M1} &= \mathrm{M}[\mathit{lm}/\mathit{left}, \mathit{rm}/\mathit{right}] \\ \mathrm{M2} &= \mathrm{M}[\mathit{lc}/\mathit{left}, \mathit{rc}/\mathit{right}]. \end{aligned}$$

Since the media are still nondeterministic we shall refine them one more step in the next section.

3.5 Implementation in occam

For an introduction to occam see [INMOS 84] or [Jones 87]. The main structure of the occam program we give for the ABP is an exact copy of its description, in the previous section, in CSP: it consists of four processes running in parallel and connected as in Figure 2:

```
-- declaration part
CHAN in,out:
CHAN lm,rm,lc,rc:
DEF milli.second = 1000:

PROC mpass1 (CHAN left,right) =
  ... define message-passing medium

PROC mpass2 (CHAN left,right) =
```

```
  ... define medium for acknowledgements

PROC sender (CHAN lm,rc)=
  ... define sending part

PROC receiver(CHAN rm,lc) =
  ... define receiving part

--  process body
PAR
  mpass1(lm,rm)
  mpass2(lc,rc)
  sender(lm,rc)
  receiver(rm,lc)
```

It remains to exhibit the subprocesses defined in the declaration part of the program and to show that they refine their counterparts in CSP. Since we are now dealing with sequential processes this is straightforward and we shall omit detailed proofs. The main points are that in occam, while-loops replace recursion, and variables have to be explicitly assigned all the state information which in CSP is contained in subscripts. Bearing this in mind we can translate the CSP specifications into occam as follows. The receiver is

```
PROC receiver(CHAN rm,lc) =
  VAR compare,mess,bit:
  SEQ
    compare := 1
    WHILE TRUE
      SEQ
        rm? bit;mess
        IF
          bit <> compare
            SEQ
              out!mess
              compare := bit
          TRUE
            SKIP
```

```
      lc! bit:
```

The sender is slightly more complicated, firstly because of the mutual recursion between S_b and $S_{\bar{x}.b}$ and secondly because of the timeout. The mutual recursion is translated into a nested while-loop, whereas the time-out involves two separate commands: `TIME?start`, which sets the value of `start` to the current reading on the system's clock, and `TIME? AFTER start+milli.second`, which becomes true when the difference between the current time and `start` is more than `milli.second` (these two commands differ from the implementation described in [INMOS 84]; for an explanation see [Jones 87], p.36).

```
PROC sender (CHAN lm,rc)=
  VAR ack,bit,mess,start,waitforack:
  SEQ
    bit:= 0
    WHILE TRUE
      SEQ
        in? mess
        lm! bit;mess
        waitforack:= TRUE
        WHILE waitforack
          SEQ
            TIME? start
            ALT
              rc? ack
                IF
                  bit = ack
                    SEQ
                      waitforack:=FALSE
                      bit:=1-bit
                  bit <> ack
                    SKIP
              TIME? AFTER start + milli.second
                lm! bit;mess:
```

To simulate non-deterministic choice in the media we use a random-number generator written by Geraint Jones. The numbers generated are

in the range of 1 to 511, and the chance of a message getting through the medium is about 50%—much worse than in acceptable transmission lines!

```
PROC mpass1 (CHAN left,right) =
  --  declaration part
  CHAN r:
  DEF k=100:
  DEF mask = NOT ((NOT 0) << 9):
  --  random number generator
  PROC shift (VAR state) =
    SEQ i = [1 FOR 9]
      state:= ((state<<1) /\ mask) \/ (((state>>4) >< (state>>8))
/\ 1):
  PROC random (CHAN c) =
    VAR state:
    SEQ
      TIME? state
      state:= state \/ 1
      WHILE TRUE
        SEQ
          shift(state)
          c ! state :
  --  body
  PAR
    random(r)
    VAR xrnd,mess,bit,count:
    SEQ
      count:= 1
      WHILE TRUE
        IF
          count = 1
            PAR
              left? bit;mess
              count:=2
          (1< count) AND (count < k)
            SEQ
              r? xrnd
              IF
```

```
        xrnd < 250
          PAR
            right! bit;mess
            count := 1
        TRUE
          PAR
            left?bit;mess
            count := count +1
count=k
  PAR
    right!bit;mess
    count:=1:
```

PROC mpasa2 is identical to PROC mpasa1 except for the fact that it handles only bits instead of both bits and messages.

This completes the implementation.

4 The sliding-window protocol

4.1 Specification in CSP

Like the alternating-bit protocol, the sliding-window protocol (SWP) is designed to ensure safe transmission of data through a medium that sometimes loses them; however it is able to deal with several messages outstanding at the same time. For a detailed informal description of the SWP see [Tbaum 81], p.148 ff.; it can be summarized as follows.

Each message is tagged with a sequence number, ranging from 0 up to some maximum. The sender is permitted to dispatch several messages with consecutive tags whilst awaiting their acknowledgements. These messages are said to fall within the *sender's window*. At the other end, the receiver maintains a *receiver's window* consisting of a list of message tags which it is prepared to accept. The sender's window and the receiver's window need not have the same upper and lower limits, or even the same size; we suppose that the size of the sender's window is w and that of the receiver is v.

It is, of course, possible to describe the SWP from first principles (see, for example, [Duke 87]) just as we have done for the ABP. However both the specification and the proof that it forms a buffer are then more difficult. Instead we choose to specify the SWP in terms of the ABP, and to deduce its correctness accordingly.

We deal first with the case $v = w$ of equal window sizes. The basic idea is to use w-many ABP's working in parallel. Then if each message in the sender's window is being dealt with by a separate ABP, an array of these ABP's will be able to deal with all the messages concurrently and, after some slight modifications, to mimic the behaviour of the SWP. To distinguish the sender and receiver of the SWP from the senders and receivers of the ABP's, we refer to the former as *global* and the latter as *local* (except in terms such as "sender's window" and "receiver's window" which are obviously global).

In specifying the SWP we need not only the w-many ABP's (numbered 0 to $w-1$), but two very simple processes: a distributor DIS and a collector COL, connected to the ABP's as shown in Figure 3.

DIS takes the input stream of data and sends one message to each ABP in turn:

$$\begin{aligned} \mathrm{DIS} &= \mathrm{D}_0 \\ \mathrm{D}_h &= in?x \rightarrow h.send!x \rightarrow \mathrm{D}_{h \oplus_w 1} \quad \text{where } 0 \leq h < w. \end{aligned}$$

At the other end, messages are collected from the local receivers by the

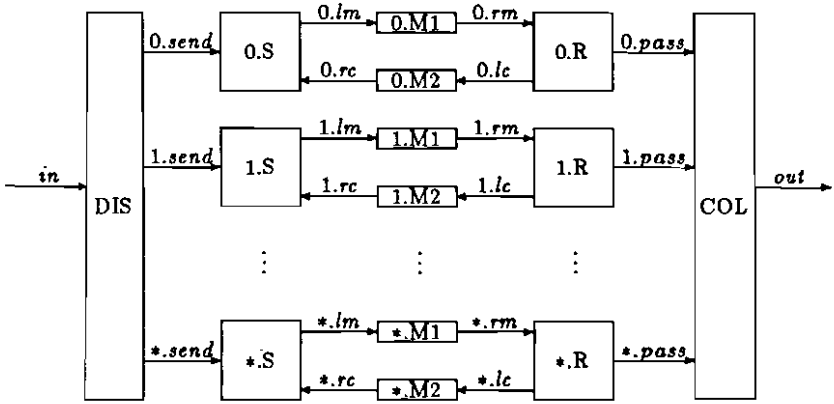

Figure 3: The SWP in terms of parallel ABP's.

process COL:

$$\begin{aligned} \mathrm{COL} &= \mathrm{C}_0 \\ \mathrm{C}_l &= l.pass?x \rightarrow out!x \rightarrow \mathrm{C}_{l \oplus_w 1} \quad \text{where } 0 \leq l < w. \end{aligned}$$

Informally we see that the processes behave as follows.

- Once the first w messages have been distributed by the process DIS the ABP's will refuse to accept further messages until the ones they are currently dealing with have been acknowledged. DIS therefore has to wait until the 0^{th} ABP receives an acknowledgement for the 0^{th} message before it can pass on the w^{th} message, and so on. Hence there can never be more than w outstanding messages in the system.

- The local receivers, together with COL, behave like a global receiver which accepts messages in any order up to w ahead of the last message output, stores them until they form a consecutive sequence, and outputs the whole sequence in correct order. In short, they behave like a global receiver with window size $v = w$.

Since these points are important in our understanding of the SWP it is hardly surprising that their proof is central to its verification. But first we must formally define the SWP!

The media remain as for the ABP and we set, for $0 \leq i < w$, (observe that we must rename the input and output channels)

$$i.\mathrm{ABP} \;\widehat{=}\; (\, i.\mathrm{S} \,\|\, i.\mathrm{M1} \,\|\, i.\mathrm{M2} \,\|\, i.\mathrm{R}\,)[i.send/in, i.pass/out] \setminus \{i.lm, i.rm, i.lc, i.rc\}.$$

Then the SWP with equal window sizes is

$$\mathrm{SWP} \;\widehat{=}\; (\,\mathrm{DIS} \,\|_{0 \leq i < w}\, i.\mathrm{ABP} \,\|\, \mathrm{COL}\,) \setminus \mathrm{Chan}$$

where

$$\mathrm{Chan} \;\widehat{=}\; \bigcup\{i.send \mid 0 \leq i < w\} \cup \bigcup\{i.pass \mid 0 \leq i < w\}.$$

4.2 Proof of Correctness

The correctness of the SWP with equal window sizes is summarized in

Theorem 3. When $v = w$, SWP **sat** $(w+2)Buffer$.

Proof. To handle the set of channels $\{i.send \mid 0 \leq i < w\}$ we write $send$ for the sequence of events $i.x$ (which written in full are events $i.send.x$) in which DIS engages. Thus $p_1(send)$ is the sequence of index numbers of ABP's with which DIS communicates, and $p_2(send)$ is the sequence of values communicated to them.

We call a sequence *cyclic* if, for some n, it is a prefix of the sequence $[0, 1, \cdots, w-1]$ concatenated with itself n times:

$$cyclic(s) \;\widehat{=}\; \exists\, n : \mathbb{N} \cdot s \leq [0, 1, \cdots, w-1]^n.$$

From the definitions of DIS and COL,

$$\mathrm{DIS} \;\; \textbf{sat} \;\; p_2(send) \leq^1 in \tag{1}$$

$$cyclic(p_1(send)), \tag{2}$$

$$\mathrm{COL} \;\; \textbf{sat} \;\; out \leq^1 p_2(pass) \tag{3}$$

$$cyclic(p_1(pass)). \tag{4}$$

From Theorem 2

$$\forall\, i : 0 \,..\, (w-1) \cdot i.pass \leq^1 i.send \tag{5}$$

hence

$$\#pass \leq^{w} \#send. \tag{6}$$

From (2), (4) and (5) we deduce

$$pass \leq send$$

which from (6) gives

$$p_2(pass) \leq^{w} p_2(send). \tag{7}$$

Finally (3), (7) and (1) give

$$out \leq^{w+2} in$$

as required. □

We have considered the SWP with window size $v = w$. The case $v > w$ can be ignored as it means that the receiver is prepared to accept messages further than w ahead of the one last received, even though the sender can never transmit such messages.

The remaining case $1 \leq v < w$ is treated by inserting a governor, GOV, as shown in Figure 4.

GOV records the trailing edge t of the receiver's window and the set K of indices in it which have been acknowledged; it ignores messages transmitted more than v ahead of t and increments the trailing edge when a sequence of consecutive acknowledgements warrants it. Of course all these calculations are modulo w, so we have to be careful in specifying intervals in the circle $0 \,..\, w-1$; we do it as follows. To say that index i lies more than v ahead of the trailing edge t is to assert

$$\begin{aligned} A \quad \triangleq \quad & (t \oplus_w v < t) \wedge (t \oplus_w v < i \leq t) \\ & \vee \\ & (t < t \oplus_w v) \wedge (t \oplus_w v < i < w \vee 0 \leq i \leq t). \end{aligned}$$

To say that k is in excess of u by no more than v is to assert

$$\begin{aligned} B \quad \triangleq \quad & (u \oplus_w v < u) \wedge (0 \leq k < u \oplus_w v \vee u < k < w) \\ & \vee \\ & (u < u \oplus_w v) \wedge (u < k \leq u \oplus_w v). \end{aligned}$$

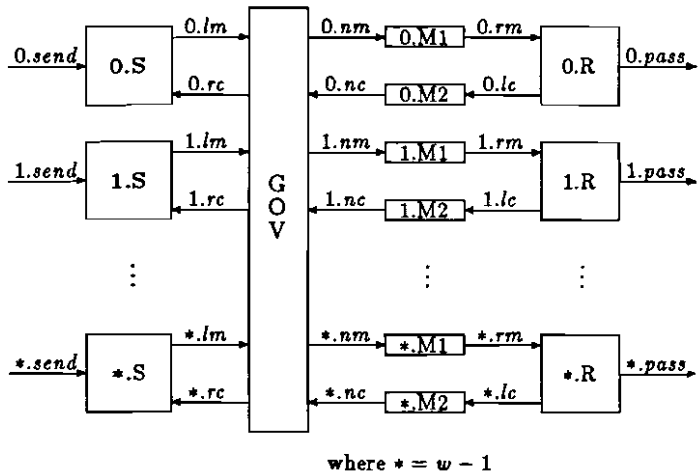

Figure 4: Configuring the receiver's window.

Now we can express the governor

$$\mathrm{GOV} \;\widehat{=}\; \mathrm{G}_{w-1,\{\}}$$

$$\mathrm{G}_{t,K} \;\widehat{=}\; \Big[\!\Big]_{0\le i<w}\, i.nc?a \to i.rc!a \to \mathrm{G}_{u,L} \;\Big[\!\Big]\; \Big[\!\Big]_{0\le i<w}\, i.lm?a \to (\mathrm{G}_{t,K} \lhd A \rhd i.nm!a \to \mathrm{G}_{t,K})$$

where $0 \le t < w$; $K \in \mathrm{P}(0\,..\,w-1)$; $L = \{n \in K \mid B\}$ is the set of indices above u which have been acknowledged; and u is the maximum value in K below which there are no gaps in acknowledgement—it is computed, using the $*$ notation for while loops from [Hoare 85], p.186, by

$$(t \oplus_w 1 \in K) \;*\; (u := u \oplus_w 1).$$

To include GOV we must relabel the channels of the media i.M1 and i.M2 (which were originally $\{i.lm,\ i.rm\}$ and $\{i.lc,\ i.rc\}$ respectively): for

$0 \leq i < w$ we set

$$i.\mathrm{M1}' \triangleq i.\mathrm{M1}[i.nm/i.lm],$$
$$i.\mathrm{M2}' \triangleq i.\mathrm{M2}[i.nc/i.rc]$$

and we also let

$$\mathrm{A}_i \triangleq \{i.lm, i.nm.i.rm, i.rc, i.nc, i.lc, i.pass, i.send\}.$$

The SWP with receiver's window size $1 \leq v < w$ is defined by

$$\mathrm{SWP_G} \triangleq \mathrm{DIS} \,\|\, \mathrm{GOV} \,\|\, \|_{0 \leq i < w} (\, i.\mathrm{S} \,\|\, i.\mathrm{M1}' \,\|\, i.\mathrm{M2}' \,\|\, i.\mathrm{R}\,) \,\|\, \mathrm{COL}$$
$$\setminus \bigcup\{A_i \mid 0 \leq i < w\}$$

We must now show that, since GOV either copies or ignores messages, its behaviour in parallel with the media is indistinguishable from that of the media alone.

Theorem 4. $\mathrm{SWP_G}$ **sat** $(w+2)$*Buffer*.

Proof. From the definitions of GOV and $i.\mathrm{M1}'$ we see

$$\mathrm{GOV} \quad \textbf{sat} \quad i.nm \trianglelefteq i.lm$$

$$i.\mathrm{M1}' \quad \textbf{sat} \quad i.rm \trianglelefteq i.nm,$$

hence

$$\mathrm{GOV} \,\|\, i.\mathrm{M1}' \quad \textbf{sat} \quad i.rm \trianglelefteq i.lm.$$

Using this as the medium in Theorem 2 we deduce

$$\forall i : 0 \,..\, (w-1) \cdot i.pass \leq^1 i.send$$

hence

$$pass \leq^w send$$

and so

$$p_2(pass) \leq^w p_2(send).$$

From the definitions of COL and DIS we have

$$out \leq^1 p_2(pass)$$
$$p_2(send) \leq^1 in.$$

We conclude that

$$out \leq^{1+w+1} in. \quad \square$$

Note. The usual way of explaining the SWP employs tags which are natural numbers (not just bits); indeed this is where the terms *sender's window* and *receiver's window* come from. We have not needed to employ such tags, but can always reconstruct them from our description (where their ghost appears in the indices of the ABP's and their alternating tags), or even rephrase it in terms of them. However we think the present approach is simpler.

4.3 Refinement

The description of the SWP given above relies solely on the use of local information. The following refinement increases its efficiency by making use of global information about the sequence of acknowledgements.

Since messages are output and acknowledged strictly in order, forward jumps in the sequence of acknowledgements received by S can arise only if some acknowledgements have been lost; since COL guarantees that the intermediate messages must have been output, the sender's window can be advanced accordingly. The design described in the previous section, however, waits until duplicate acknowledgements have been generated (by the sender timing out) to do so.

This unnecessary delay can be avoided by modifying process GOV to give a new process FILL. Recall that GOV relays acknowledgements from $i.nc$ to $i.rc$, incrementing the trailing edge and updating K. We now wish FILL, whilst still doing that, to fill in gaps between a new acknowledgement and the highest received so far.

The set of indices which have received an acknowledgement, now contain no gaps and so the set K in GOV can be suppressed. However FILL must relay acknowledgements of the correct parity: if $k < i$ then the parity of the bits being filled in coincides with the parity of acknowledgement i; otherwise acknowledgements of the parity opposite to the i^{th} must be filled in until $w - 1$ when the original parity must be used until i. Thus (using the same

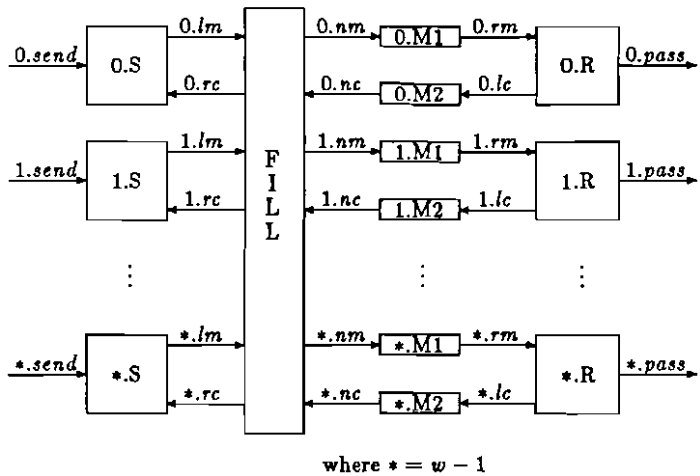

Figure 5: Filling gaps in acknowledgement.

predicate A as in the definition of GOV)

$$\mathrm{FILL} \;\widehat{=}\; \mathrm{F}_{w-1}$$

$$\mathrm{F}_k \;\widehat{=}\; \prod_{0\le i<w} i.nc?a \to X_a \;\Big[\!\Big]\; \prod_{0\le i<w} i.lm?a \to (\mathrm{F}_k \lhd A \rhd i.nm!a \to \mathrm{F}_k)$$

where

$$X_a \;\widehat{=}\; (k \neq i) * (k := k \oplus_w 1 \to k.rc!a \to SKIP) ; \mathrm{F}_i \lhd k < i \rhd (k \neq w) * (k := k \oplus_w 1 \to k.rc!1 \ominus a \to SKIP) ; (k \neq i) * (k := k \oplus_w 1 \to k.rc!a \to SKIP) ; \mathrm{F}_i.$$

The refined version, $\mathrm{SWP_F}$, of the SWP is defined

$$\mathrm{SWP_F} \;\hat{=}\; \mathrm{DIS} \,\|\, \mathrm{FILL} \,\|\, \|_{0 \le i < w} (i.\mathrm{S} \,\|\, i.\mathrm{M1}' \,\|\, i.\mathrm{M2}' \,\|\, i.\mathrm{R}) \,\|\, \mathrm{COL} \setminus \bigcup\{\mathrm{A}_i \mid 0 \le i < w\},$$

where again

$$\mathrm{A}_i \;\hat{=}\; \{i.lm, i.nm.i.rm, i.rc, i.nc, i.lc, i.pass, i.send\}.$$

At first glance the insertion of FILL may seem to destroy the independence of the ABP's and with it the validity of the proof of correctness of $\mathrm{SWP_F}$. The reason for this illusion is perhaps the fact that (cf. GOV)

$$(\mathrm{FILL} \,\|\, i.\mathrm{M2}') \setminus \{i.nc\}$$

does not refine

$$i.\mathrm{M2}$$

(since the former can fill in outputs which the latter lost). But we should not expect it to—only in the context of the rest of the protocol do we envisage replacing the latter with the former. Indeed

Theorem 5. $\mathrm{SWP_F}$ sat SWP.

Proof. Recall, for $0 \le i < w$, the definitions of $i.\mathrm{M1}'$ and $i.\mathrm{M2}'$, and let

$$\mathrm{C}_i \;\hat{=}\; \{i.lm, i.nm, i.rm, i.lc, i.nc, i.rc\}.$$

From its definition

$$\begin{array}{lll} \mathrm{FILL} & \mathbf{sat} & cyclic(p_1(rc)) \\ & & nm \trianglelefteq lm \\ & & (nc \le^1 rc) \vee (rc \le^1 nc). \end{array}$$

Thus from the definition of $i.\mathrm{M1}$,

$$(\mathrm{FILL} \,\|\, i.\mathrm{M1}') \setminus \{i.nm\} \quad \mathbf{sat} \quad i.\mathrm{M1}.$$

Now by the proof of Corollary 1,

$$(\mathrm{FILL} \,\|\, i.\mathrm{S} \| i.\mathrm{M1}' \| i.\mathrm{M2}' \| i.\mathrm{R}) \setminus (\mathrm{C}_i \cup \{x : \mathrm{Chan} \mid p_1(x) \neq i\})$$
$$\mathbf{sat}$$
$$i.pass \le^1 i.send.$$

In other words FILL combined with a single ABP is as good as the ABP on its own:

$$(\mathrm{FILL} \,\|\, i.\mathrm{S} \| i.\mathrm{M1}' \| i.\mathrm{M2}' \| i.\mathrm{R}) \setminus (\mathrm{C}_i \cup \{x : \mathrm{Chan} \mid p_1(x) \neq i\})$$
sat
$i.ABP$.

Using the standard laws of CSP and the fact that FILL, DIS and COL are all deterministic,

$$\mathrm{SWP_F}$$
$$=$$
$$(\mathrm{DIS} \,\|\, \mathrm{FILL} \,\|\, \|_{0 \leq i < w} (i.\mathrm{S} \,\|\, i.\mathrm{M1}' \,\|\, i.\mathrm{M2}' \,\|\, i.\mathrm{R}) \,\|\, \mathrm{COL}) \setminus \bigcup\{\mathrm{A}_i \mid 0 \leq i < w\}$$
sat
$$(\mathrm{DIS} \,\|\, \|_{0 \leq i < w} (\mathrm{FILL} \,\|\, i.\mathrm{S} \,\|\, i.\mathrm{M1}' \,\|\, i.\mathrm{M2}' \,\|\, i.\mathrm{R}) \setminus \bigcup\{\mathrm{C}_i \mid 0 \leq i < w\} \,\|\, \mathrm{COL}) \setminus \mathrm{Chan}$$
sat
$$(\mathrm{DIS} \,\|\, \|_{0 \leq i < w} i.\mathrm{ABP} \,\|\, \mathrm{COL}) \setminus \mathrm{Chan}$$
$$=$$
$$\mathrm{SWP},$$

and this completes the proof. □

4.4 Implementation in occam

We choose to implement the SWP without the benefit of the modifications of the previous section; indeed the following program implements the SWP with equal window sizes. As before, the main structure of the occam implementation is identical to that of the CSP description.

```
PROGRAM swp

-- declaration part
DEF w=4:  -- w stands for sender's window size
CHAN in:
CHAN out:
CHAN send[w],pass[w]:

PROC dis =
```

```
  ... define distributing process

PROC abp(VALUE i,CHAN send,pass) =
  ... include alternating bit protocol

PROC col =
  ... define collecting process

-- process body
PAR
  dis
  PAR i=[0 FOR w-1]
    abp(i,send[i],pass[i])
  col
```

The processes dis and col are both very simple:

```
PROC dis =
  --  distributes messages to individual abp's
  VAR mess:
  SEQ
    WHILE TRUE
      SEQ h= [0 FOR w-1]
        SEQ
          in? mess
          send[h]! mess:


PROC col =
  --  collects messages from abp's
  VAR mess:
  SEQ
    WHILE TRUE
      SEQ l= [0 FOR w-1]
        SEQ
          pass[l]? mess
          out! mess:
```

Apart from a minor change to the header our sliding-window protocol is ready to run!

5 Acknowledgements

The idea that the SWP might he viewed as several ABP's communicating and acknowledging in parallel was communicated to us by Carroll Morgan and we gratefully acknowledge this. Geraint Jones, Tony Hoare and Jeremy Jacob made invaluable suggestions after reading drafts and Andrew Kay helped with occam code.

Part of the research was supported by the Science and Engineering Research Council of Great Britain.

References

[Duke 87] R. Duke, I. Hayes and G. Rose, *Verification of a cyclic retransmission protocol*, preprint, 1987.

[Hoare 85] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall International, 1985.

[INMOS 84] INMOS Ltd., *occam Programming Manual*, Prentice-Hall International, 1984.

[Jones 87] G. Jones, *Programming in 'occam'*, Prentice-Hall International, 1987.

[Tbaum 81] A.S. Tanenbaum, *Computer Networks*, Prentice-Hall International, 1981.

[Hayes 87] I. Hayes (editor), *Specification Case Studies*, Prentice-Hall International, 1987.