Protocol Specification, Testing and Verification, XII
R.J. Linn, Jr. and M.Ü. Üyar
Elsevier Science Publishers B.V. (North-Holland)

The Two-Dimensional Window Protocol

Mohamed G. Gouda

Department of Computer Sciences, University of Texas at Austin, Austin, Texas 78712-1188, U. S. A.

This work is supported in part by a grant from the Texas Advanced Technology Program: 1992-1994.

Abstract

We develop a generalization of the window protocol where data messages are sent in independent streams. Each stream has its own window, but the window sizes of different streams can be changed at runtime under the restriction that their sum remains constant. This restriction is needed to establish an upper bound on the required resources for the protocol. Our development of this protocol is "gentle" as it is divided into three "easy" steps. First, a regular window protocol is presented and formally verified. Second, this window protocol is extended to allow multiple data streams with fixed window sizes. The extended protocol is verified by extending the verification of the window protocol in a straightforward manner. Third, the multiple stream protocol is modified to allow the window sizes of different streams to change at runtime. The modification is slight and verification of the resulting protocol is derived from the verification of the multiple stream protocol.

1. INTRODUCTION

The window protocol is the most widely-used protocol for controlling the transmission of data messages from a sender process to a receiver process over a medium that may lose, corrupt, reorder, or duplicate transmitted messages. This protocol and its many variations have been studied extensively; see for instance [1], [2], [3], [4], [6], [7], [8] and [9]. However, all variations of the window protocol are based on the assumption that the data messages to be transmitted are totally ordered, and that they should be delivered according to that total order. Hence, the sender process in the window protocol sends the data messages according to that total order, and the receiver process receives the messages and delivers them according to that same order. If the receiver happens to receive the messages out of order, due to some transmission fault, then the receiver keeps the received messages until it can deliver them according to their total order.

The problem with this assumption is two-fold. First, in many situations, the data messages to be transmitted are not totally ordered and there is no requirement to deliver them according to a total order. Second, the requirement to deliver the transmitted data messages according to a predefined total order is expensive to satisfy. We discuss this problems in some detail next.

In many situations, the data messages to be transmitted are ordered according to some lax partial order, rather than a rigid total order. Consider, for example, a situation where a sender is to send two files A and B to a receiver. To send a file, the sender first divides the file into a number of small messages then proceeds to send these messages, one by one, according to their order in the file. Thus, the sender sends the messages of file A according to some total order and the receiver delivers these messages according to that same order. Similarly, the sender sends the messages of file B according to some total order and the receiver delivers these messages according to that same order. There is no requirement to order the messages of file A with respect to those of file B. Sending and delivering the messages of file A should proceed independently of sending and delivering the messages of file B.

However, if the sender is to use a window protocol to send all messages, those that belong to file A and those that belong to file B, then the sender has to impose an arbitrary total order on all the messages and send them according to that order. At the other side, the receiver has to abide by that (arbitrary) total order and deliver the messages according to it. This can be costly. For example, if a message from file A is lost while being transmitted from the sender to the receiver, then the receiver cannot deliver any message of file B, that was sent after the lost message, until the lost message is resent and delivered. Note that in this case the message delivery of file B is delayed even though no message of file B is lost.

The situation is even worse if the sender is to send a number of unrelated messages to the receiver using a window protocol. In this case, the sender has to impose an arbitrary total order over the unrelated messages and send them according to that total order. The receiver also has to deliver these unrelated messages according to that same total order.

To solve this problem, we develop in this paper a generalization of the window protocol where the data messages to be transmitted form a number of independent streams (partial order), rather than a single stream (total order). Each stream has its own window. However, the window sizes of different streams can change at runtime under the restriction that their sum remains constant in order to guarantee an upper bound on the required resources for the protocol.

Our development of this new protocol, which we call the two-dimensional window protocol, consists of three steps. First, we present a window protocol with a single stream (Section 3). Second, we extend this protocol to allow multiple streams with independent, fixed-size windows (Section 4). Third, we modify the second protocol to allow the window sizes of different streams to change at runtime while keeping their sum constant (Section 5). The resulting protocol is our two-dimensional window protocol.

For convenience, we give, in the next section, a brief presentation of the notation that we use to define communication protocols, and the method that we apply to verify their correctness. (A detailed presentation of both the notation and the verification method can be found in [5].)

2. ON PROTOCOL DEFINITION AND VERIFICATION

In this paper, we consider protocols that consist of two processes p and q. Each process is defined by a set of local variables and a set of actions.

process <process name>

var <Pascal-like declarations of local variables>

begin
<action>
[] <action>
...
[] <action>

end

Each action is of the form:

<label> : <guard> → <statement>

The label is a number that uniquely identifies the action. The guard is either a boolean expression or a receive statement of the form: **rcv** g, for some message g. The statement is defined recursively as one of the following:

a **skip** statement,

an assignment statement,

a send statement of the form: **send** g,

a sequence of statements of the form: <statement> ; <statement>,

a conditional statement of the form: **if** <condition> **then** <statement> **fi**, or

an iterative statement of the form: **while** <condition> **do** <statement> **od**.

There are two one-directional channels between processes p and q. At each instant during protocol execution, each channel has a sequence of messages. The message sequence in the channel from p to q contains all data messages that have been sent by p and have not yet been received by q. The message sequence in the channel from q to p contains all acknowledgement messages that have been sent by q and have not yet been received by p. The sending of a message from p to q (or from q to p) consists of adding the message at the tail of the message sequence in the channel from p to q (or from q to p, respectively). The receiving of a message

from p to q (or from q to p) consists of removing the message at the head of the message sequence in the channel from p to q (or from q to p, respectively).

The message sequence in a channel is updated by executing send or receive statements. It is also updated by the occurrence of the following types of faults.

i. *Message Reorder:*
Two messages in the message sequence are interchanged.

ii. *Message Loss:*
One message disappears from the message sequence.

iii. *Message Corruption:*
One message in the message sequence is replaced by the special message error.

iv. *Message Duplication:*
One message in the message sequence is duplicated, and the duplicate message is placed in any position in the message sequence.

A protocol state is defined by a value for each local variable in process p or q, a sequence of data or error messages in the channel from p to q, and a sequence of acknowledgement or error messages in the channel from q to p.

An action in process p (or q) is enabled for execution at a protocol state s if the guard of the action is a boolean expression that evaluates to true at s, or if the guard is a receive statement **rcv** g, and g is the head message in the message sequence in the channel from q to p (or from p to q, respectively) at state s.

Execution of an enabled action in process p (or q) depends on whether the guard of the action is a boolean expression or a receive statement. If the guard is a boolean expression, the action is executed by executing the statement of the action. If the guard is a receive statement, the action is executed by first receiving the head message from the message sequence in the channel from q to p (or from p to q, respectively), then executing the statement of the action.

A protocol computation is a maximal sequence of the form:

State.0 ; action.0 ; state.1 ; action.1 ; state.2 ; ...

where each state.i is a protocol state, and each action.i is an action in p or q that is enabled for execution at state.i. Moreover, executing action.i starting at state.i yields state.(i+1). The maximality of the sequence means that the sequence is infinite, or it is finite but no action is enabled for execution at its last state.

Correctness of any such protocol can be established by showing that the protocol satisfies three logical properties: safety, fault-tolerance, and progress. These three properties can be expressed in terms of some predicate C that assigns a value, true or false, to every state of the protocol.

i. *Safety:*
This property states that if the protocol is at any state where C holds, then executing any action in p or q yields a state where C holds; i. e. C is closed under protocol execution.

ii. *Fault-Tolerance:*
This property states that if the protocol is at any state where C holds, then each occurrence of message reorder, loss, corruption, or duplication in one of the two channels between p and q yields a state where C holds; i.e. C is closed under fault occurrence.

iii. *Progress:*
This property states that for every protocol computation that starts with a state where C holds, each of the variables, where newly sent data messages in p or newly received data messages in q are counted, is incremented infinitely often along the computation.

A predicate C that satisfies these three properties defines a closed domain for protocol execution. (This domain is closed under both protocol execution and fault occurrence.) In this closed domain, safety and fault-tolerance are always guaranteed, and progress is guaranteed infinitely often. Such a predicate is referred to as a protocol closure [5].

We are now ready to start our development of the two-dimensional window protocol. As mentioned earlier, our first step is to define and verify a window protocol.

3. A WINDOW PROTOCOL

Consider a protocol consisting of two processes p and q. Process p sends data messages to process q, and q replies by sending back an acknowledgement message for every data message it receives from p. Each data message is of the form dt(i), where i is a natural number called the sequence number of the message. Each acknowledgement message is of the form ak(i), where i is the sequence number of the data message whose reception by q is acknowledged by ak(i).

Let C denote the protocol closure that describes the execution domain of the protocol. C is defined and verified (i. e. shown to satisfy the three properties of safety, fault-tolerance, and progress described above) below.

Process p has two local variables na and ns whose values range over the natural numbers. Variable na stores the sequence number of the next data message whose acknowledgement is yet to be received by p. Variable ns stores the sequence number of the next data message to be sent by p. The following relation holds at every protocol state where C holds (i. e. at every protocol state that is encountered during protocol execution).

$$na \leq ns$$

By design, there is a positive integer ws such that the value of ns never exceeds the value of na+ws at every protocol state where C holds.

$$ns \leq na+ws$$

Constant ws is called the window size of the protocol.

Process q has a local variable nr whose value ranges over the natural numbers. Variable nr stores the sequence number of the next data message to be received by q. Hence, the following relation holds at every protocol state where C holds.

$$nr \leq ns$$

Process q sends an acknowledgement message for every data message it receives; the sent acknowledgement message has the same sequence number as that of the received data message. Thus, the value of nr can be viewed as the sequence number of the next acknowledgement message to be sent by q. Hence, the following relation holds at every protocol state where C holds.

$$na \leq nr$$

In summary, four relations hold at every protocol state where C holds. These four relations can be expressed collectively as follows.

$$na \leq nr \leq ns \leq na+ws \qquad (1)$$

Because process p can receive the acknowledgement messages out of order (in which they were sent), p has an infinite boolean array ackd such that

ackd[i] = true if p has received message ak(i)
false if p has not received message ak(i)

Hence, all elements of array ackd are false initially.

Similarly, because process q can receive the data messages out of order (in which they were sent), q has an infinite boolean array rcvd such that

rcvd[i] = true if q has received message dt(i)
false if q has not received message dt(i)

Hence, all elements of array rcvd are false initially.

Process p has four actions: one action for sending the next data message, one action for receiving an acknowledgement message, one timeout action for resending an old data message (when this message or its acknowledgement is lost), and one action for receiving and discarding a corrupted message. We describe each of these actions in order.

The sequence number of the next data message to be sent is ns. Thus, sending this message would cause ns to be incremented by one. In order to maintain the relation ns ≤ na+ws, this message can be sent only when ns < na+ws. Thus, the action for sending the next data message can be written as follows.

1: ns < na+ws → **send** dt(ns) ; ns := ns+1

On receiving an ak(i) message the ith element in array ackd is set to true and na is incremented one by one until it points to the next element in array ackd whose value is false. The action for receiving an acknowledgement message can be written as follows.

2: **rcv** ak(i) → ackd[i] := true ;
while ackd[na] **do** na := na+1 **od**

The timeout action in process p is enabled for execution at any protocol state where the following three conditions hold: (na<ns), ~(dt(na) in ch.p.q), and ~(ak(na) in ch.q.p). The fist condition implies that p has sent message dt(na) and has not yet received its acknowledgement ak(na). The second condition states that message dt(na) is not in the channel from p to q. The third condition states that

the acknowledgement message ak(na) is not in the channel from q to p. Hence, the conjunction of these three conditions implies that either message dt(na) is lost from the channel from p to q, or its acknowledgement ak(na) is lost from the channel from q to p.When these three conditions hold, message dt(na) is sent one more time. The timeout condition is written as follows.

3: **timeout** na< ns ∧ ~(dt(na) in ch.p.q) ∧ ~(ak(na) in ch.q.p)
→ **send** dt(na)

Process p discards every corrupted message it receives. Thus, the action for receiving a corrupted message can be written as follows.

4: **rcv** error → **skip**

Process p can now be defined as follows.

```
process p

const ws        :   natural number    { ws > 0}

var   na, ns, i :   natural number  ,
      ackd      :   array natural number] of boolean

begin

   1: ns < na+ws →  send dt(ns) ; ns := ns+1
[] 2: rcv ak(i)  →  ackd[i] := true ;
                    while ackd[na] do na := na+1 od
[] 3: timeout na< ns ∧ ~(dt(na) in ch.p.q) ∧ ~(ak(na) in ch.q.p)
                 →  send dt(na)
[] 4: rcv error  →  skip

end
```

Process q has two actions: one action for receiving a data message and sending back its acknowledgement, and one action for receiving then discarding a corrupted message. We discuss the first action in detail. (The second action is similar to action 4 in process p.)

On receiving a dt(j) message, process q sends the ak(j) message, then compares the message sequence number j with the value of nr. (Recall that nr stores the sequence number of the next data message to be received by q.) If j is at least nr, then the received dt(j) message is new. In this case, the jth element of array rcvd is set to true, and nr is incremented one by one until it points to the next element in array rcvd whose value is false. The action for receiving a data message in process q can be written as follows.

```
5: rcv dt(j)  →  send ak(j) ;
                 if j ≥ nr
                 then  rcvd[j] := true ;
                       while rcvd[nr] do nr := nr+1 od
                 fi
```

Process q can be defined as follows.

```
process q

var   nr, j      :   natural number ;
      rcvd       :   array [natural number] of boolean ;

begin

   5: rcv dt(j)   →   send ak(j) ;
                      if j ≥ nr
                      then  rcvd[j] := true ;
                            while rcvd[nr] do nr := nr+1 od
                      fi
[] 6: rcv error   →   skip

end
```

A closure C for the above protocol is as follows. (We still have to prove that it is indeed a closure for the protocol.)

C = c1 ∧ c2 ∧ c3 ∧ c4 ∧ c5 ∧ c6 ∧ c7

where

c1 = na ≤ nr ≤ ns ≤ na+ws

c2 = (∀ k less than na : ackd[k]) ∧ ~ackd[na]

c3 = (∀ k less than nr : rcvd[k]) ∧ ~rcvd[nr]

c4 = (∀ dt(k) message in ch.p.q : k < ns)

c5 = (∀ k : rcvd[k] ⇒ k < ns)

c6 = (∀ ak(k) message in ch.q.p : rcvd[k])

c7 = (∀ k : ackd[k] ⇒ rcvd[k])

Predicate C consists of the seven conjucts c1 to c7. Conjunct c1 is simply relation (1) that is discussed earlier: this guarantees that relation (1) holds at every protocol state where C holds as required. Conjunct c2 states that every sequence number less than na has been acknowledged in p, but na itself has not yet been acknowledged in p. Conjunct c3 states that every sequence number less than nr has been received in q, but nr itself has not yet been received in q. Conjunct c4 states that the sequence number of every data message in the channel from p to q

is less than ns. Conjunct c5 states that every sequence number that has been received in q is less than ns. Conjunct c6 states that the sequence number of every acknowledgement message in the channel from q to p has been received in q. Finally, Conjunct c7 states that every sequence number that has been acknowledged in p has been received in q.

In order to prove that C is a closure for the protocol, we need to show that C satisfies the three conditions of safety, fault-tolerance, and progress discussed earlier in Section 2. This is done next.

3.1. Proving Safety

We need to prove that if any action in p or q is executed at a state where C holds, then C holds at the resulting state. We carry the proof for action 2 in process p. (Proofs for other actions are similar.)

Recall that action 2 is as follows.

```
2: rcv ak(i)   →   ackd[i] := true ;
                   while ackd[na] do na := na+1 od
```

This action assigns ackd[i] the value true and increments na one by one until ~ackd[na]. The four conjuncts c3, c4, c5, and c6 are not affected by this action and so they remain true. Because both c6 and c7 are true before the execution of action 2, c7 is true after the execution. It remains now to show that the two conjuncts c1 and c2 remain true after the execution of action 2.

If the value of na is not changed by the execution of action 2, then both c1 and c2 remain true. Assume that the value of na is changed from u to v by the execution of action 2. From the fact that the value of na has changed from u to v, and because c2 holds before the execution, the following condition holds before the execution.

(∀ k less than u : ackd[k]) ∧
(~ackd[u]) ∧
(∀ k larger than u and less than v : ackd[k]) ∧
(the sequence number i of the ak message received in action 2 is u)

From this condition and because both c6 and c7 hold before the execution, the condition (∀ k less than v : rcvd[k]) holds before the execution. Therefore the relation v ≤ nr holds before the execution of action 2, and remains true after the execution. Hence, na ≤ nr and c1 remain true after the execution.

After executing action 2, we have the following condition.

(na = v) ∧
(∀ k less than v : ackd[k]) ∧
(~ackd[v])

Hence, c2 is true after the execution.

3.2. Proving Fault-Tolerance

None of the conjuncts of C depends on the order, or number of messages in the channels, or number of error or duplicate messages in the channels. Hence, any fault occurrence at a state where C holds yields a state where C holds.

3.3. Proving Progress

We need to prove that each of the variables na, ns, and nr is incremented infinitely often along any protocol computation that starts with a state where C holds. First, we show that every protocol computation is infinite. Second, we show that along every infinite computation that starts with a state where C holds, each of the variables na, ns, and nr is incremented infinitely often.

To prove the first part, consider a protocol state s where both channels are empty (which implies that none of the actions 2, 4, 5, and 6 is enabled for execution at s), and where ns =na+ws (which implies that action 1 is not enabled for execution at s). In this case, action 3 is enabled for execution at state s. Therefore, at least one action is enabled for execution at every protocol state, and every protocol computation is infinite.

To prove the second part, let c be an arbitrary infinite computation of the protocol, and assume that c starts with a state where C holds. From the safety property, C holds at every state in computation c. Therefore, the relation $na \le nr \le ns \le na+ws$ holds at every state in c. Hence, it is enough to show that at least one of the variables na, ns, or nr is incremented infinitely often along c. In other words, it is enough to show that at least one of the actions 1, 2 with i=na, or 5 with j=nr is executed infinitely often along c. Because c is infinite, process p has to send messages infinitely often along c. If action 1 is not executed infinitely often along c, then action 3 has to be executed infinitely often along c. These infinitely many executions of action 3 cause action 5 with j=na to be executed infinitely often along c, and action 2 with i=na to be executed infinitely often along c. Therefore, na is incremented infinitely often along c, and each of ns and nr is incremented infinitely often along c.

4. A MULTIPLE WINDOW PROTOCOL

The data messages in the above protocol can be viewed as forming one stream from p to q. In this section, we develop a window protocol with r independent streams from p to q. Each stream is identified by a unique identifier in the range 1...r.

The streams are independent because each of them has its own messages, its own variables, and its own window. Each data message (sent by process p) is of the form: dt(x,i), where x is the stream to which the message belongs, and i is the message sequence number in stream x. Similarly, each acknowledgement message (sent by process q) is of the form: ak(x,i), where x is the stream to which the message belongs, and i is the message sequence number in stream x. Each stream x has two variables na[x] and ns[x], and one array ackd[x] in process p. Also, each stream x has one variable nr[x] and one array rcvd[x] in process q.

Relation (1) is now generalized as follows.

$$na[x] \le nr[x] \le ns[x] \le na[x]+w[x] \qquad \text{for each } x=1,\ldots,r \qquad (2)$$

where w[x] is a natural number called the window size of stream x. Note that the window size of a stream may be zero; however, we require that the sum of window sizes of all streams be at least one.

Process p for the multiple window protocol can now be defined as follows.

```
process  p

const  r      :  natural number , { r > 0 }
       w      :  array [1..r] of natural number
                 { At least one w[x] is more than 0 }

var    na, ns :  array [1..r] of natural number ,
       i      :  natural number ,
       ackd   :  array [1..r, natural number] of boolean

par    x      :  1..r

begin

   1:  ns[x] < na[x]+w[x]   →   send dt(x, ns[x]) ; ns[x] := ns[x]+1
[] 2:  rcv ak(x, i)         →   ackd[x, i] := true ;
                                while ackd[x, na[x]]
                                do na[x] := na[x]+1 od
[] 3:  timeout na[x] < ns[x] ∧
            ~(dt(x, na[x]) in ch.p.q) ∧ ~(ak(x,na[x]) in ch.q.p)
                            →   send dt(x, na[x])
[] 4:  rcv error            →   skip

end
```

This process definition includes a parameter x whose value ranges over 1..r. Each action that is defined using parameter x is called a parameterized action. A parameterized action is in fact a set of r actions: each of which is defined by replacing x by one of its values. For example, the parameterized action

```
   1:    ns[x] < na[x]+w[x]   →   send dt(x, ns[x]) ; ns[x] := ns[x]+1
```

is a short-hand for the following set of actions.

```
   1.1:  ns[1] < na[1]+w[1]   →   send dt(1, ns[1]) ; ns[1] := ns[1]+1
[] 1.2:  ns[2] < na[2]+w[2]   →   send dt(2, ns[2]) ; ns[2] := ns[2]+1
...
[] 1.r:  ns[r] < na[r]+w[r]   →   send dt(r, ns[r]) ; ns[r] := ns[r]+1
```

Process q for the multiple window protocol can be defined as follows.

```
process q

var    nr     :  array [1..r] of natural number ,
       j      :  natural number ,
       rcvd   :  array [1..r, natural number] of boolean
```

```
par   y          :    1..r
begin
   5: rcv dt(y, j)   →   send ak(y, j) ;
                         if j ≥ nr[y]
                         then  rcvd[y, j] := true ;
                               while rcvd[y, nr[y]]
                               do nr[y] := nr[y]+1 od
                         fi
[] 6: rcv error      →   skip
end
```

This process definition includes a parameter y whose value ranges over 1..r. It also includes one parameterized action, action 5, which defines a set of r actions: each of these actions is obtained from the parameterized action by substituting one value from 1..r for parameter y.

A closure for the multiple window protocol is as follows. (We still have to prove that it is a closure for the protocol.)

D = (∀ u in the range 1..r : d.u)

where

d.u = d1.u ∧ d2.u ∧ d3.u ∧ d4 .u ∧ d5.u ∧ d6.u ∧ d7.u

d1.u = na[u] ≤ nr[u] ≤ ns[u] ≤ na[u]+w[u]

d2.u = (∀ k less than na[u] : ackd[u, k]) ∧ ~ackd[u, na[u]]

d3.u = (∀ k less than nr[u] : rcvd[u, k]) ∧ ~rcvd[u, nr[u]]

d4.u = (∀ dt(u, k) message in ch.p.q : k < ns[u])

d5.u = (∀ k : rcvd[u, k] ⇒ k < ns[u])

d6.u = (∀ ak(u, k) message in ch.q.p : rcvd[u, k])

d7.u = (∀ k : ackd[u, k] ⇒ rcvd[u, k])

It is straightforward to show that each d.u satisfies the safety and fault-tolerance properties discussed in Section 2. (The proof is similar to the above proof showing that predicate C satisfies the safety and fault-tolerance properties for the window protocol.) Moreover, each d.u satisfies the following progress property. For every protocol computation that starts with a state where d.u holds,

if w[u] > 0 and if the computation has infinite number of actions with x = u in p, then each of the variables na[u], ns[u], and nr[u] is incremented infinitely often along the computation.

Because each d.u satisfies the safety and fault-tolerance properties, D satisfies the safety and fault-tolerance properties. Because each d.u satisfies the above progress property and because the window size of at least one stream is greater than 0, D satisfies the following progress property. For every protocol computation that starts with a state where D holds, there is at least one stream u where each of the variables na[u], ns[u], and nr[u] is incremented infinitely often along the computation. Hence, predicate D is a closure for the multiple window protocol.

5. A TWO-DIMENSIONAL WINDOW PROTOCOL

The two-dimensional window protocol is obtained from the multiple window protocol in the previous section as follows. The window sizes of different streams can change at runtime under the restriction that their sum remains constant. In particular, the sum of window sizes of different streams always equals the number of streams. Let ws denote the number of streams and w[x] denote the window size of stream x. Also, let D be a closure for the two-dimensional window protocol. (We shall see to it that closure D for the multiple window protocol is a closure for the two-dimensional window protocol after r is replaced by ws.) Then, the following relation holds at every protocol state where D holds.

(sum over x in the range 1..ws : w[x]) = ws (3)

The reason for guaranteeing that relation (3) holds at every protocol state where D holds is to establish an upper bound on the required resources for the two-dimensional window protocol. In particular, the maximum number of data messages that remain in process p, after they have been sent and before they have been acknowledged, is ws. Also, the maximum number of messages that can be in each of the two channels between p and q is ws. Moreover, the maximum number of data messages that can be in process q, after they have been received and before they can be delivered, is ws-1.

In order to guarantee that relation (3) holds at every protocol state where D holds, any time the window size of a stream x is incremented by one, the window size of another stream z is decremented by one. But decrementing w[z] by one can happen only when ns[z] < na[z]+w[z] in order to ensure that the relation ns[z] ≤ na[z]+w[z] is maintained. Thus, the action for incrementing the window size of stream x in process p is as follows.

```
7: ns[x] = na[x]+w[x] ∧ ns[z] < na[z]+w[z]
                  →   send dt(x,ns[x]) ; ns[x] := ns[x]+1 ;
                      w[x] := w[x]+1 ; w[z] := w[z]-1
```

Process p for the two-dimensional window protocol can be defined as follows.

```
process  p

const  ws      :   natural number  ,  { ws > 0 }
       w       :   array [1..ws] of natural number
                   { Sum of w[1], ... , w[ws] is ws }

var    na, ns  :   array [1..ws] of natural number  ,
       i       :   natural number  ,
       ackd    :   array [1..ws, natural number] of boolean

par    x, z    :   1..ws

begin

   1:  ns[x] < na[x]+w[x]   →   send dt(x, ns[x]) ; ns[x] := ns[x]+1
[] 2:  rcv ak(x, i)         →   ackd[x, i] := true ;
                                while ackd[x, na[x]]
                                do na[x] := na[x]+1 od
[] 3:  timeout na[x] < ns[x] ∧
            ~(dt(x, na[x]) in ch.p.q) ∧ ~(ak(na[x]) in ch.q.p)
                            →   send dt(x, na[x])
[] 4:  rcv error            →   skip
[] 7:  ns[x] = na[x]+w[x] ∧ ns[z] < na[z]+w[z]
                            →   send dt(x,ns[x]) ; ns[x] := ns[x]+1 ;
                                w[x] := w[x]+1 ; w[z] := w[z]-1
end
```

Process q for the two-dimensional window protocol is exactly the same as process q for the multiple window protocol.

It is straightforward to prove that predicate D, which is a closure for the multiple window protocol, is a closure for the two dimensional window protocol after r is replaced by ws. The proof is based on the following three observations. First, D is a closure for the multiple window protocol. Second, the two-dimensional window protocol differs from the multiple window protocol only in the introduction of action 7 to process p. Third, action 7 satisfies the property: If action 7 is executed at a protocol state where D holds, then D holds at the resulting state.

6. CONCLUDING REMARKS

The original window protocol is but a special case of the two-dimensional window protocol, where the number of streams is one. As usual, the advantage of a special case over a general case is its simplicity, and the advantage of a general case over a special case is its flexibility.

The simplicity of the window protocol over the two-dimensional window protocol is demonstrated by two aspects. First, the messages in the window protocol have no stream identifiers because all messages form one stream. Second, the two processes p and q in the window protocol are simpler than those in the two-dimensional window protocol because they do not have to manage multiple streams. An argument can be made, however, that the cost of adding a number of bits per message to record the stream identifier of the message, and designing the two processes to handle multiple streams is not high.

The flexibility of the two-dimensional window protocol over the original window protocol is due to the independence of the multiple transmitted streams. When a stream suffers from a message loss and is delayed until the loss is detected and the lost message is retransmitted and acknowledged, other streams can still progress without delay.

It is straightforward to show that the same steps that we have followed in this paper can be followed to transform any version of the window protocol into a corresponding version of the two-dimensional window protocol.

7. REFERENCES

[1] K. A. Bartlett, R. A. Scantlebury, and P. Wilkinson, "A note on reliable full-duplex transmission over half-duplex links", Communications of the ACM, Vol. 12, pp. 260-261, 1969.

[2] G. M. Brown, M. G. Gouda, and R. E. Miller, "Block acknowledgement: Redesigning the window protocol", IEEE Transactions on Communications, Vol. COM-39, pp. 524-532, 1991.

[3] V. G. Cerf and R. E. Khan, "A protocol for packet network intercommunication", IEEE Transactions on Communications, Vol. COM-22, pp. 637-648, 1974.

[4] M. G. Gouda, "On a simple protocol whose proof is not: The state machine approach", IEEE Transactions on Communications, Vol. COM-33, pp. 380-382, 1985.

[5] M. G. Gouda, "Protocol Verification Made Simple," Computer Networks and ISDN, to appear 1992.

[6] B. Hailpern and S. S. Owicki, "Modular verification of computer verification protocols", IEEE Transactions on Communications, Vol. COM-31, pp. 56-68, 1983.

[7] W. C. Lynch, "Reliable full-duplex file transmission over half-duplex telephone lines", Communications of the ACM, Vol. 11, pp. 407-410, 1968.

[8] A. U. Shankar, "Verified data-transfer protocols with variable flow control", ACM Transactions on Computing Systems, Vol. 7, No. 3, pp. 281-316, 1989.

[9] N. V. Stenning, "A data transfer protocol", Computer Networks, Vol. 1, pp. 99-110, 1976.