SENDIN SEQUENCES:

A TECHNIQUE TO SPECIFY COMMUNICATION PROTOCOLS

MOHAMED G. GOUDA

TR 81-181

DEPARTMENT OF COMPUTER SCIENCES
UNIVERSITY OF TEXAS AT AUSTIN
AUSTIN, TX 78712

TR 181 Oct. 1981

Table of Contents

ABSTRACT

The communication between two processes can be specified using all possible sequences of interleaved sending operations executed by the two processes. We use this technique to specify a number of transmission and transport protocols, and discuss a possible role for such specifications in the verification, synthesis, and run-time checking of communication protocols.

Keywords: Communication protocol specification, Protocol verification, Protocol run-time checking, Protocol synthesis.

1. INTRODUCTION

Consider a system of two processes which exchange messages using a nonblocking send/blocking receive primitives, i.e., a process can resume execution immediately after sending a message without waiting for the other process to receive it. We propose to specify the communication in such a system by defining all acceptable sequences of interleaved sending operations executed by both processes. We call such an all-sending sequence a "sending sequence". Sending sequences specify the communication as seen by an outside observer who watches the sent messages travel between the two processes but cannot determine when each message is actually received, if at all, by its destination process.

In this paper we use sending sequences to specify some communication protocols. We also discuss techniques to use the resulting specification in the verification, synthesis, and run-time checking of communication protocols. Other techniques to specify communication protocols can be found in [1], [2], [3], and [4].

2. A SIMPLE EXAMPLE

A sending sequence between two processes P and Q can be constructed from the definition of both P and Q as follows. Start with an "empty" sending sequence. At each state of the sequence, use the definition of P and Q to identify <u>all</u> correct sending operations which can be executed by either P or Q at this state. Select at random one of those correct sending operations, and add it to the sequence causing its state to change to a new value. At this new state, a (possibly) different set of sending operations by either P or Q becomes correct; one of them is selected at random and added to the sequence, and the cycle repeats. The cycle continues either indefinitely yielding an infinite sending sequence, or until the sequence reaches an end state where no sending operation can be executed by either P or Q. Thus, the state of a sending sequence between P and Q can either be "empty" or in the form "send(u,m,r)" which denotes the next state of the sequence after process u (either P or Q) sends a message m when the sequence is in state r.

We propose to specify the communication protocol between two processes by the set of all sending sequences between them. To define such a set one needs to define all the correct (or equivalently all the erroneous) sending operations at each state of each sending sequence. This can be done using a specification language similar to that introduced by Guttag [5] to specify abstract data types. We illustrate this by an example.

Two processes P and Q communicate by exchanging "data" and "ack" (for positive acknowledgement) messages such that the following two conditions are always satisfied:

(i) The number of data messages sent by either process should not exceed by more than N the number of ack messages sent by the other process, where N is a predefined integer whose value is greater than or equal to one.

(ii) The number of ack messages sent by either process should not exceed the number of data messages sent by the other process.

The set of all communication sequences for this protocol can be specified as shown next followed by some explanations. (Line numbers have been added for ease of reference).

```
Send Seq S  (N: integer)
          1
    process P, Q
    msg data, ack
    init-seq-st empty
    constructor send: process x msg x seq-st --> seq-st
    state       cntd: process x seq-st --> integer
                cntk: process x seq-st --> integer

    declare  r,s: seq-st;
             u,v: distinct process; w: process;
               d: data msg; a: ack msg; m: msg;
    rules
    [1] send-by-prs-u:
      [1a] send(u,d,s) = error iff cntd(u,s)-cntk(v,s) = N
      [1b] send(u,a,s) = error iff cntd(v,s)-cntk(u,s) = 0

    [2] count-data-msgs-from-prs-u:
           cntd(u,s) = if s = empty
      [2a]                 then 0
                           elsif s = send(u,d,r)
      [2b]                    then cntd(u,r)+1
                              elsif s = send(w,m,r)
      [2c]                       then cntd(u,r)

    [3] count-ack-msgs-from-prs-u:
           cntk(u,s) = if s = empty
      [3a]                 then 0
                           elsif s = send(u,a,r)
      [3b]                    then cntk(u,k) + 1
                                 elsif s = send(w,m,r)
      [3c]                          then cntk(u,r)
end S
     1
```

For any value of N, $S_1(N)$ is the set of all sending sequences between P and W for that value of N; i.e., $S_1(N)$ defines a schema of sets rather than a single set.

The specification of $S_1(N)$ consists of a declaration section (lines 2 through 10), and a rule section (lines 11 through 28).

In the declaration section the following are specified:

- The communicating processes are P and Q (line 2)
- Exchanged messages are of types "data" and "ack" (line 3).
- The initial state of a sending sequence is denoted "empty" (line 4).
- The domain and range of each constructor and state function are also specified (lines 4, 5, and 6). Function "send" is called constructor since it can be used to construct any sending sequence state other than the initial state. State functions "cntd(u,s)" and "cntk(u,s)" give the total numbers of data and ack messages sent so far by process u at any sending sequence state s.
- Finally, the following symbols are defined (lines 8, 9, and 10).

r,s	denote two sending sequence states
u,v	denote two distinct processes. So if both u and v appear in one rule then u means process P (or Q), and v means process Q (or P respectively).
w	denotes a process.
d,a,m	denote instances of data, ack, and general messages respectively. A general message is either of type data or ack.

These rules are defined in the rule section. The first rule defines the conditions under which a sending operation executed by process u (P or Q) is erroneous. This rule uses the two state functions "cntd(u,s)" and "cntk(u,s)" which are formally defined by the second and third rules.

There are two differences between the above specification and the specification of abstract data types [5] and [6]. First, the above specification does not have "modifier" functions since sending sequences do not have receiving operations. Second, the two processes can change the current state of their sending sequence only by executing sending operations. Thus, to avoid the suspicion that a process can start a new sending sequence while in the middle of its current sequence by invoking an "empty" operation, "empty" is not defined as a function. Rather, it is defined as the initial sending sequence state before any sending takes place.

In appendix A we discuss how to use sending sequences to specify a simple transport protocol.

3. PROTOCOLS WHICH DEAL WITH TRANSMISSION ERRORS

Sending sequences can be used to specify protocols which deal with transmission errors. We consider three types of transmission errors namely message corruption, message loss, and message disorder. Message corruption causes some sent messages to be received with some errors in their contents. Message loss causes some sent messages to be lost completely and never received. Message disorder causes some sent messages to be received in a different order from the one in which they are sent.

Since sending sequences have no receiving operations, they cannot explicitly specify a situation in which a process P receives a message whose contents have been corrupted during transmission from another process Q. However, they can imply such a situation by stating that P sends a negative acknowledgement "nack" message to Q, after Q has sent a data message to P.

Similarly, sending sequences can be used to specify protocols which handle message loss. In such a specification, a process may re-send the last sent data message even after the other process has sent an ack message, thus implying that the ack has been lost during transmission. Consider the following Alternating-Bit protocol.

Two processes P and Q exchange three types of messages "data0", "data1", and "ack" such that the following three conditions are satisfied:

(i) If no message is lost, each process sends a data0 message, receives an ack, sends a data1 message, receives an ack, sends a data0 message and so on.

(ii) If a data message is lost during transmission from a process, then the process will not receive an ack, and it has to time itself out to re-send the last data message.

(iii) If an ack message is lost during its transmission from a process, then the process may receive again the last data message. However, the process can detect this redundancy by recognizing that both messages are of the same type, i.e., they are both of type data0 or type data1.

A sending sequence specification for this protocol is as follows:

```
ComSeq S  init empty
       3
   process P,Q
   msg data0, data1, ack

   constructor send: process x msg x seq_st--->seq_st

   state       lastd: processxseq_st--->msg
                ack?: processxseq_st--->boolean

   declare r,s: seq_st;
           u,v: distinct process; w: process;
         d0,e0: distinct data0 msg; d1,e1:distinct data1 msg;
             a: ack msg; m: msg;

   rules
   [1] send_by_prs_u:
       send(u,d0,s) error if lastd(u,s) = e0
                             or(lastd(u,s) = e1
                                and not ack?(u,s))

       send(u,d1,s) error if s = empty
                             or lastd(u,s) = e1
                             or(lastd(u,s) = e0
                                and not ack?(u,s))

       send(u,a,s) error if ack?(v,s)

   [2] last_data_msg_from_prs_u:
       lastd(u,s) = if s = empty or s = send(u,d1,r)
                       then d1
                       elsif s = send(u,d0,r)
                          then d0
                          elsif s = send(w,m,r)
                             thenlastd(u,r)

   [3] has_last_data_msg_from_u_been_acknowledged_?:
       ack?(u,s) = if s = empty or s = send(v,a,r)
                      then true
                      elsif s = send(u,d0,r) or s = send(u,d1,r)
                         then false
                         elsif s = send(w,m,r)
                            then ack?(u,r)
end S ;
     3
```

Sending sequences can be also used to specify protocols which handle message disorder. In one such protocol, a process sends N data messages hoping that they will be received in order by the other process. When the other process responds by sending back an ack(n) message, it indicates that only n of the N sent messages have been received in order; the remaining N-n have been received out-of-order and hence discarded. This causes the first process to resend the last N-n messages and adds n new messages and the cycle repeats. This protocol can be specified using sending sequences, we leave the details to the reader.

4. VERIFICATION

Let S be a set of sending sequences between two processes; and let P and Q be two communicating processes. P and Q are said to <u>realize</u> S if each sending sequence between P and Q is a member in S. To prove that some P and Q realize a given S, one should prove that each time P (or Q) is about to send a message, the error condition for sending, as defined by S, is false at this time. This is better illustrated by an example.

Consider the set $S_1(N)$ of sending sequences defined in section 2. The error condition for P to send a "data" message is (line 13):

$$\text{cntsdP} - \text{cntskQ} = N \qquad (1)$$

where cntsdP is the number of data messages sent so far by P, and cntskQ is the number of ack messages sent so far by Q. Thus, to prove that a given process P sends data messages according to the definition of $S_1(N)$, it is sufficient to show that the condition

$$\text{cntsdP} - \text{cntskQ} < N \qquad (2)$$

holds just before P sends any data message. Since P and Q communicate using nonblocking send/blocking receive primitives, we have

$$\text{cntskQ} \geq \text{cntrkP} \qquad (3)$$

where cntrkP is the number of ack messages received by P. From (3), condition (2) can be replaced by

$$\text{cntsdP} - \text{cntrkP} < N \qquad (4)$$

Similarly, to prove that a given P sends ack messages according to the definition of $S_1(N)$, it is sufficient to show that the condition

$$\text{cntrdP} - \text{cntskP} > 0 \qquad (5)$$

holds just before P sends any ack message, where cntrdP is the number of data messages received so far by P and cntskP is the number of ack messages sent so far by P. Below we show an annotated program for process P followed by some explanations:

```
process P;
   const N = ...;
   msg data, ack;
   var num1, num2 : integer;
       cntsdP, cntskP, cntrdP, cntrkP : integer

   begin num1 := num2 := 0;
         cntsdP := cntskP := cntrdP := cntrkP := 0;
         {R}
         *[num1<N ^ NEW(data):
            {R^S} Q!data; cntsdP := cntsdP + 1;
               num1 := num1 + 1 {R}
         []num1 > 0 ^ Q?ack:
            {R} cntrkP := cntrkP + 1;
                num1 := num1 - 1 {R}
         []num2 < N ^ NXT(data) ^ Q?data:
            {R} cntrdP := cntrdP + 1;
                num2 := num2 + 1 {R}
         []num2 > 0:
            {R^T} Q!ack; cntskP := cntskP + 1;
                  num2 := num2 - 1 {R}
         ]
end P;
```

__Notes:__ (i) Although we use a syntax similar to that of Hoare [7], the semantics of our communication primitives is different from that of his. The receiving primitive "Q?m" in a guard g_i becomes __true__ when an "m" message is at the head of P's input buffer. When guard g_i becomes __true__ and is selected so that its following executable statements s_i are to be executed, then the head "m" message must be removed from the P's input buffer before executing s_i. The sending primitive "Q!m" is executed by adding an "m" message to the tail of Q's input buffer. (ii) Nine assertions {...} have been inserted in nine places in the above program; they are based on three assertions R, S, and T defined as follows:

$$\begin{aligned} R &:: (cntsdP = cntrkP = num1) \wedge (0 \leq num1 \leq N) \wedge \\ &\quad (cntrdP = cntskP = num2) \wedge (0 \leq num2 \leq N) \\ S &:: (num1 < N) \\ T &:: (num2 > 0) \end{aligned}$$

Assertion R is the loop invariant. It is straightforward to show that each of the nine assertions holds in its place in the program. Since {R^S}, which implies cntsdP = cntrkP < N, holds just before sending any data message and since {R^T}, which implies cntrdP=cntskP>0, holds just before sending any ack message, then P sends all messages according to the specification of $S_1(N)$. Also, we can design a program for Q similar to that of P and prove that Q also sends all messages according to the specification of $S_1(N)$. (iii) The above scheme has some disadvantages and some advantages. __Disadvantages:__ The variables cntsdP, cntskP, cntrdP, cntrkP, along with their increment operations are redundant; they are only introduced for the sake of the proof and they have to be removed later. __Advantages:__ The proofs do not require formal definitions for the communication primitives. Also the proof for P proceeds independently from that for Q similar to the proof system discussed in [8].

So far we have proved that so long as P and Q continue to exchange messages, their sending sequence is a prefix of a sending sequence in $S_1(N)$. To prove that their sequence is indeed an element in $S_1(N)$, it is sufficient to show that each sending sequence between P and Q is infinite; i.e., P and Q cannot reach a state after which no more sending or receiving is possible. This requires to prove the following four assertions:

R1: Neither process can reach a state after which neither sending nor receiving is enables. [A process is said to be in a <u>sending state</u> if a sending operation is enabled in this state. It is in a <u>receiving state</u> if a receiving operation is enabled and no sending operation is enabled in this state.]

R2: <u>No Deadlocks</u>: Both processes cannot reach receiving states while their input message buffers are empty.

R3: <u>No Unspecified Receptions</u>: Neither process can reach a receiving state where no message of type t is expected while a message of type t is at the head of the input message buffer of the process.

R4: <u>No Unbounded Communication</u>: Neither process can reach a sending state while the input message buffer of the other process is full.

Next, we present formal statements for these assertions and prove them for the above processes P and Q. The proofs are by contradiction; i.e., we show that R_i = false for i = 1,...,4.

R_1 :: P is in nonsending and nonreceiving state

:: $T_1 \wedge T_2$

where T_1 :: P is in nonsending state

:: (num1 < N) ^ (num2 > 0)...From P def.

:: (num1 = N) ^ (num2 = 0) ...From inv. R

and T_2 :: P is in nonreceiving state

:: (num1 = 0) ^ (num2 = N) ...From P and R

Thus R_1 :: false

R_2 :: P and Q is in a deadlock state

:: $T_1 \wedge T_2 \wedge T_3 \wedge T_4$

where T_1 :: P is in a receiving state

:: $(num1 = N) \wedge (num2 = 0)$

:: $(cntsdP - cntrkP = N) \wedge (cntrdP - cntskP = 0)$

T_2 :: Q is in a receiving state

:: $(cntsdQ - cntskQ = N) \wedge (cntrdQ - cntskQ = 0)$

T_3 :: input message buffer of P is empty

:: $(cntsdQ - cntrdP = 0) \wedge (cntrdQ - cntskQ = 0)$

T_4 :: input message buffer of Q is empty

:: $(cntsdP - cntrdQ = 0) \wedge (cntskP - cntrkQ = 0)$

Thus R_2 :: false

R_3 :: P is in unspecified reception state

:: $(T_1 \wedge T_2(data) \wedge T_3(data)) \vee (T_1 \wedge T_2(ack) \wedge T_3(ack))$

where T_1 :: P is in a receiving state

:: $(num1 = N) \wedge (num2 = 0)$

$T_2(t)$ :: P is not expecting a message of type t

$T_2(data)$ :: $(num2 < N) = (num2 = N)$

$T_2(ack)$ :: $(num1 > 0) = (num1 = 0)$

$T_3(t)$ :: a message of type t is at the head of P's input buffer.

Thus R_3 :: false

R_4 :: the input buffer of Q can hold more than N data messages + N ack messages.

:: $T_1 \wedge T_2$

where T_1 :: P is in a sending state

:: $(num1 < N) \vee (num2 > 0)$

:: $(cntsdP - cntrkP < N) \vee (cntrdP - cntskP > 0)$

:: $(cntsdP - cntskQ < N) \vee (cntsdQ - cntskP > 0)$

T_2 :: input buffer of Q has N data messages + N ack messages

:: $(cntsdP - cntrdQ = N) \wedge (cntskP - cntrkQ = N)$

:: $(cntsdP - cntskQ \geq N) \wedge (cntskP - cntsdQ \geq 0)$

Thus R_4 :: false

This completes the proof that the above processes P and Q realize $S_1(N)$.

5. SYNTHESIS

The set of sending sequences between two processes can be used to deduce the external behaviour of both processes; this can be a first step towards their complete design. Consider the class of sending sequences which can be represented by regular expressions. In this class, the set of sending sequences between processes P and Q can be represented by a regular expression over the alphabet:

$$\{send(P,m_1), send(Q,m_1), \ldots, send(P,m_r), send(Q,m_r)\}$$

where $m_1, \ldots, m_r$ are distinct message types. An example of such regular expression is as follows:

$$(send(P,m_1).(send(Q,m_2).[send(Q,m_3)]*.send(P,M_4) + send(P,m_3)))*$$

which can be represented by the finite state machine in Figure 1a. Next, we discuss an algorithm to solve the following problem. Given a finite state machine MS for the sending sequences between two processes P and Q, construct two finite state machines MP and MQ to represent the external behaviours of P and Q (respectively) such that the following two conditions are satisfied:

(i) The communication between MP and MQ is deadlock-free, bounded, and has no unspecified receptions.

(ii) Each sending sequence between MP and MQ is identical to a sending sequence in MS with the possible exception of some additional "null" message exchanges between MP and MQ. [Null messages are added to ensure freedom of deadlocks, boundedness, etc.]

The algorithm is based on two assumptions. First, the output edges of any node in MS must have distinct labels. Second, messages transmitted between MP and MQ arrive without errors, and in the same order they are sent. The algorithm consists of four steps.

Algorithm:

Step 1: Construct a finite state machine MP_1 identical to MS, and label its edges such that if an edge is labelled send(P,m_i) or send(Q,m_i) in MS, then the corresponding edge in MP_1 is labelled snd(m_i) or rcv(m_i) respectively. Edges labelled snd(m) (or rcv(m)) in MP_1 are called sending (or receiving) edges.

Step 2: For any node x with both sending and receiving output edges in MP_1, do the following:

- Add a new node y to MP_1.
- Add a directed edge labelled rcv(null) from x to
- All the sending output edges of x become outputs of y. Thus, the outputs of node x are all receiving edges, while the outputs of node y are all sending edges.

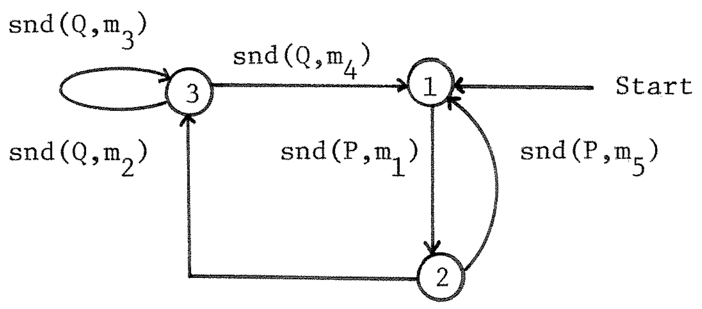
Let MP_2 denote the resulting finite state machine after all such modifications.

Step 3: For any directed cycle in MP_2 whose edges are all sending (or all receiving), add a new node with one receiving (or one sending) edge labelled rcv(null) (or snd(null)) respectively). Let MP denote the resulting finite state machine after all such modification.

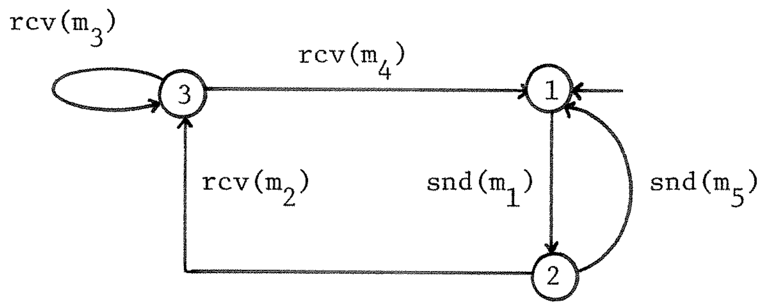
Step 4: Construct a finite state machine MQ identical to MP; and label its edges such that if an edge is labelled snd(m_i), snd(null), rcv(m_i), or rcv(null) in MP, then the corresponding edge in MQ is labelled rcv(m_i), rcv(null), snd(m_i), or snd(null) respectively.[]

Applying this algorithm to the MS in Figure 1a, we get the MP and MQ in Figures 1d and 1e respectively. Step 2 and step 4 in the algorithm guarantee that the i^{th} operation executed by MP (or MQ) is a sending operation where a message "m" is sent if the i^{th} operation of MQ (or MP respectively) is a receiving operation where "m" is received. This is sufficient to prove that the communication between MP and MQ is deadlock-free and has no unspecified receptions.

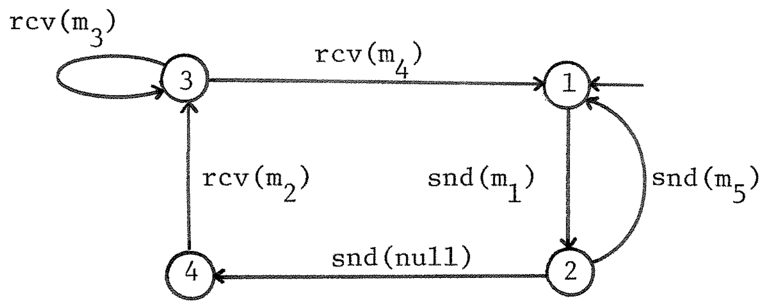
To prove boundedness, it is sufficient to find upper bounds KP and KQ on the number of messages that can exist, at any instant, in the input message buffers of MP and MQ respectively. To find KP, remove all the sending edges from MP; the resulting graph is acyclic from step 3 in the algorithm. The number of edges in the longest path in this graph is KP. Similarly, KQ can be evaluated from MQ.



(a) MS: A finite state machine for the sending sequences bet. P and Q.

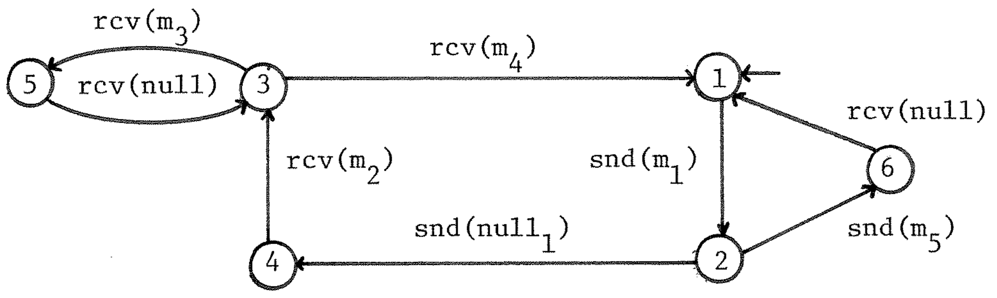


(b) MP_1: An initial finite state machine for the external behaviour of P.

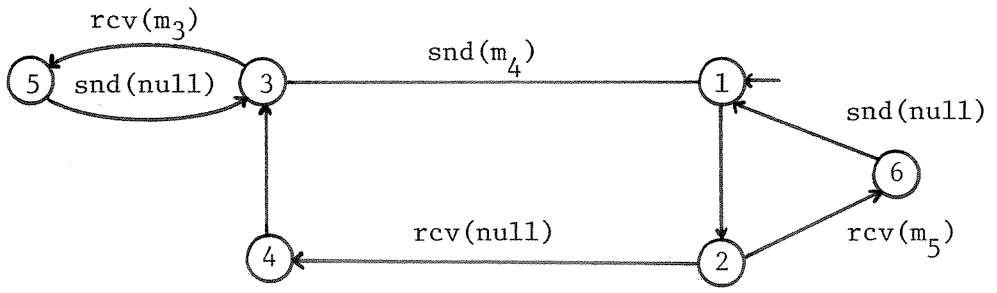


(c) MP_2: Adding "null" messages to ensure freedom of deadlocks, and absence of unspecified receptions.

FIGURE 1. An example for constructing the external behaviour of P and Q from their sending sequences.



(d) MP: Adding "null" messages to ensure boundedness.



(e) MQ

FIGURE 1 (Continued)

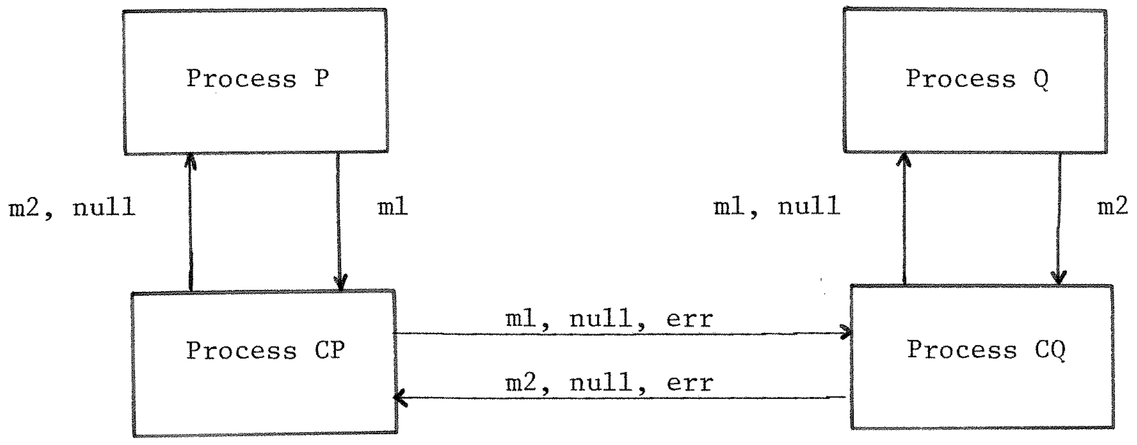


FIGURE 2. A runtime checking system for the communication between P and Q.

6. RUN-TIME CHECKING

Sending sequence specifications can be also used in checking the external behaviour of communicating processes at run-time. Let S be a set of sending sequences; and let P and Q be two communicating processes designed to realize S. To check that indeed P and Q realize S, two additional processes CP and CQ are designed, using the specification of S, and placed between P and Q as in Figure 2. Both CP and CQ store the current state of the sending sequence constructed so far between P and Q. When a message m1 is sent by P, it first goes to CP where sequencing errors are checked. If there are no errors, a "null" message is sent to P so that it can resume execution. Then, m_1 is sent to CQ which updates its current state of the sending sequence and sends back a "null" message to CP so that it also updates its current state of the sequence. Finally, m1 is sent to process Q. If CP detects a sequencing error on receiving m1, it sends an "err" message to CQ; and both CP and CQ reach an ERROR state and stop execution. A program for the CP process is as follows:

```
*[P? m1 : [   seq-err(m1): CQ! err; ERROR
          []  seq-err(m1): P! null; CQ! m1;
                    [ CQ? null: update-seq(m1)
                    []CQ?   m2: update-seq(m1); update-seq(m2)
                    []CQ?  err: ERROR
                    ]
          ]
[]CQ? m2: CQ! null; Q! m2; update-seq(m2)
[]CQ? err: ERROR
]
```

Notes: (i) After CP sends m1 to CQ, it waits to receive a "null" message from CQ; however it may receive an m_2 or a null message instead. On receiving m_2, CP updates the current state of the sending sequence using m_1 then m_2. On the other side, CQ must do the same so that both CP and CQ reach the same state of the sending sequence. (ii) The types of exchanged messages m_1 and m_2, the predicate seq-err(m_1), and the execution statements update-seq(m_1) can be all deduced from the given specification of S. As an example, we present in Appendix B a detailed definition of CP for the set of sending sequences S_1(N) defined in section 2. (iii) In the above scheme, it is assumed that the following condition is satisfied:

```
If send(P,$m_1$,s) <> error and send(Q,$m_2$,s) <> error
   then send(Q,$m_2$,send(P,$m_1$,s)) <> error
```

for any state s of a sending sequence between P and Q, and for any m_1 and m_2 exchanged between them. This condition means that if it is correct, at some stage of the communication, for P to send a message m_1 **or** for Q to send a message m_2, then it must be correct at this stage for P to send m_1 **then** for Q to send m_2. All reasonable communication protocols should satisfy this condition.

7. CONCLUDING REMARKS

We have proposed to specify the communication protocol between two processes by the set of all sending sequences between them, and suggested a simple specification language for this purpose. We have also discussed how to use such specifications in the verification, synthesis, and run-time checking of communication protocols.

In this paper, processes are assumed to communicate using nonblocking send/blocking receive primitives. This implies that receiving a message by one process occurs, if at all, at some later time after the message is sent by the other process. To avoid this potentially complex dependency between sending and receiving operations, we have decided to use sequences of sending operations only in our specification. This has contributed greatly to the simplicity of the resulting specifications.

Hoare [9], Misra and Chandy [8], and Bochmann [10] have suggested similar specification techniques when the processes communicate using blocking send/blocking receive primitives. The specification language in this paper can be also used in this case with the added understanding that each sending operation by one process implies a receiving operation by the other process.

The discussion in this paper is limited to the case of two processes. However, it is straightforward to extend the specification language and some of the verification and run-time checking techniques to more than two processes.

ACKNOWLEDGEMENTS: The author is thankful to Professors Chandy and Misra for helpful discussions during the course of this work. He is also thankful to Dr. Sunshine for his comments on an earlier draft of this paper. Thanks are also due to K. F. Carbone for her careful typing.

REFERENCES

1. P. M. Merlin, "Specification and Validation of Protocols," IEEE Trans. on Comm., Vol. COM-27, Nov. 1979, pp. 1661-1680.

2. C. A. Sunshine, "Formal Modeling of Communication Protocols," USC/Info. Sc. Inst. Tech. Report 81-89, Mar. 81.

3. G. V. Bochmann, "A General Transition Model for Protocols and Communication Services," IEEE Trans. on Comm., Vol. COM-28, No. 4, April 80, pp. 643-650.

4. D. H. Thompson, et al, "Specification and Verification of Communication Protocols in AFFIRM Using State Transition Models," USC/Infor. Sc. Inst. Tech. Report 81-88, Mar. 81.

5. J. Guttag, "Abstract Data Types and the Development of Data Structures," Comm. of the ACM, Vol. 20, June 1977, pp. 396-404.

6. L. Flon and J. Misra, "A Unified Approach to the Specification and Verification of Abstract Data Types," Proc. of the Conf. on Specif. for Reliable Software, 1979, pp. 162-169.

7. C. A. R. Hoare, "Communicating Sequential Processes," Comm. of the ACM, Vol. 21, No. 8, 1978.

8. J. Misra and K. M. Chandy, "Proofs of Networks of Processes," IEEE Trans. on Software Engineering, Vol. SE-7, No. 4, July 1981.

9. C. A. R. Hoare, "A Model for Communicating Sequential Processes," Comput. Lab., Oxford Univ., Dec. 1978.

10. P. Merlin, and G. V. Bochmann, "On the Construction of Communication Protocols and Module Specifications," Univerite De Montreal. Dept. D'informatique. Tech. Rep. 352, Jan. 80.

APPENDIX A: A SIMPLE TRANSPORT PROTOCOL

Two processes P and Q communicate according to the following transport protocol:

(i) A connection is established between the two processes only after each of them sends a "connect" message.

(ii) After establishing the connection, each process can send a "credit(k:**integer**)" message to indicate that it is willing to receive at most k "data" messages from the other user. Later, the credit may be extended by sending other credit messages.

(iii) The connection between the two processes continues until one of them sends a "disconnect" message. In this case, the other process also sends a "disconnect" message to remove the connection. There is a delay of at most $N(N \geq 1)$ message transmissions after the first process sends a disconnect message and before the second process sends a disconnect message.

(iv) After the connection removal, the two processes go to a sleep state until both of them send "connect" messages and the cycle repeats.

The following sending sequence specification for this protocol has four state functions:

lastm(u,s):	is the type of the last message sent by process u when the sending sequence is in state s. If u has not sent any message yet, then lastm(u,s) = nmg. If the last message sent by u is of type connect, credit, data, or disconnect, then the value of lastm(u,s) is con, crd, dta, or dsc respectively.
cntm(u,s):	is the number of messages sent so far by process u when the sending sequence is in state s.
cntd(u,s):	is the number of data messages sent so far by process u when the sending sequence is in state s.
cntcr(u,s):	is the number of remaining credits granted by process

u to the other process to send data messages to u.

The sending sequence specification is as follows:

```
SendSeq S (N: integer)
       3
   process P, Q
   init-seq-st empty
   constructor send: process x msg x seq-st --> seq-st
   state       lastm: process x seq-st --> (nmg,con,crd,dta,dsc)
                cntm: process x seq-st --> integer
                cntd: process x seq-st --> integer
               cnter: process x seq-st --> integer

   declare r,s: seq-st; k: integer
           u,v: distinct process; w: process;
             c: connect msg; e(k): credit(integer) msg; d: data msg;
             i: disconnect msg; m: msg;

   rules
   [1] send-by-prs-u:
      send(u,c,s) = error iff   lastm(u,s) = con
                             or lastm(w,s) = crd
                             or lastm(w,s) = dta

      send(u,e(k),s) = error iff    lastm(w,s) = nmg
                                 or lastm(u,M) = dsc
                                 or (lastm(v,s) = dsc
                                    and cntm(u,s) = N)

      send(u,d,s) = error iff    lastm(w,s) = nmg
                              or lastm(u,s) = dsc
                              or (lastm(v,s) = dsc
                                 and cntm(u,s) = N)
                              or cntd(u,s) = cntcr(v,s)

      send(u,i,s) = error iff    lastm(u,s) = nmg
                              or lastm(u,s) = dsc

   [2] last-msg-from-prs-u:
      lastm(u,s) = case s of
                   empty: nmg
                   send(u,c,r): con
                   send(u,e(k),r): crd
                   send(u,d,r): dta
                   send(u,i,r): dsc
                   send(v,m,r): lastm(u,r)
```

```
[3] count_msgs_from_prs_u:
   cntm(u,s) = if s = empty or s = send(v,i,r)
                  then 0
                  elsif s = send(u,e(k),r) or s = send(u,d,r)
                     then cntm(u,r) + 1
                     elsif s = send(w,m,r)
                        then cntm(u,r)

[4] count_data_msgs_from_prs_u:
   cntd(u,s) = if s = empty or s: send(u,c,r)
                  then 0
                  elsif s = send(u,d,r)
                     then cntd(u,r) + 1
                     elsif s = send(w,m,r)
                        then cntd(u,r)

[5] count_credits_from_prs_u:
   cntcr(u,s) = if s = empty or s = send(u,c,r)
                   then 0
                   elsif s = send(u,e(k),r)
                      then cntcr(u,r) + k
                      elsif s = send(w,m,r)
                         then cntcr(u,r)
```

APPENDIX B: A CP PROCESS FOR $S_1(N)$

A CP process can be constructed from the specification of $S_1(N)$ in section 2 as follows. The state functions cntd(P,...), cntd(Q,...), cntk(P,..), and cntk(Q,...) in the $S_1(N)$ specification are mapped to the local variables cntdP, cntdQ, cntkP, and cntkQ respectively in CP. The sending error conditions in $S_1(N)$ specification are mapped to the predicates seq-err(...) in CP. A program for CP is as follows.

```
Process CP;
   msg data, ack, null
   var cntdP, cntdQ, cntkP, cntkQ: integer

   begin
   cntdP := cntdQ := cntkP := cntkQ := 0;
   *[P? data: [ cntdp - cntkQ = N: CQ! err; ERROR
              []cntdP - cntkQ <> N: P! null; CQ! data
                        [ CQ? null: cntdP := cntdP+1
                        []CQ? data: cntdP := cntdP+1; cntdQ := cntdQ+1
                        []CQ? ack: cntdP := cntdP+1; cntkQ := cntkQ+1
                        []CQ? err: ERROR
                        ]
              ]
   []P? ack: [ cntdQ - cntkP = 0: CQ! err, ERROR
             []cntdQ - cntkP <> 0: P! null; CQ! ack
                       [ CQ? null: cntkP := cntkP+1
                       []CQ? data: cntkP := cntkP+1; cntdQ := cntdQ+1
                       []CQ? ack: cntkP := cntkP+1; cntkQ := cntkQ+1
                       []CQ? err: ERROR
                       ]
   []CQ? data: CQ! null; P! data; cntdQ := cntdQ+1
   []CQ? ack: CQ! null; P! ack; cntkQ := cntkQ+1
   []CQ? err: ERROR
   ]
end CP;
```