SENDING SEQUENCES:

FOR SPECIFICATION AND RUN-TIME

CHECKING OF PROTOCOLS

Mohamed G. Gouda

Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712

TR-171 March 1981

ABSTRACT

We propose a new model, called sending sequences, to specify the communication interactions between two or more processes in a protocol layer. In this model, a communication protocol is specified by the set of all sequences of sending operations executed by any process in the layer. Unlike specifications based on communicating finite state machines, sending sequence specifications do not exhibit inconsistencies such as deadlocks or unspecified receptions. We use the model to specify a window protocol and an alternating-bit protocol. We also use the model to design a run-time checking system which observes the communication between given processes to verify that it proceeds according to a given protocol specification. Such a system can be used in protocol testing or to increase the system reliability in face of failures.

Keywords: ✓Communication protocols, ✓protocol run-time checking, protocol specifications, ✓protocol testing.

I. INTRODUCTION

The most widely used model to specify communication interactions in protocol layers is communicating finite state machines [4] and [8]. Nevertheless, specifications based on that model exhibit inconsistencies such as deadlocks and unspecified receptions [1], [2] and [9]. This problem has guided research in two directions:

(i) Analysis: Find techniques to uncover these inconsistencies in any given complete specification [1], [3], and [9]. Unfortunately, the problem is shown to be undecidable in the most general case [3].

(ii) Synthesis: Find techniques to complete an incomplete specification such that these inconsistencies do not occur [5] and [9].

In this paper, we propose a new solution to the problem, namely a new model to specify communication interactions where these inconsistencies never occur. The paper is organized as follows. In Section II, the model is formally defined; and a specification language based on the model is detailed in Section III. Two specification examples, namely, a window protocol and an alternating-bit protocol are discussed in Section IV. In Section V a run-time checking system based on the model is outlined. Concluding remarks are in Section VI. For brevity, we limit the discussion to protocol layers with two communicating processes; it is straightforward to extend the discussion beyond that; see Section VI.

Example: Consider a communication protocol where a process p sends two data messages to a process q which answers back by sending two acknowledgement messages. This protocol C can be formally defined in our model as C = ({p,q}, {data,ack}, S). The set S is defined as follows. Let "d_1" and "d_2" be two messages of type data and let "a" be a message of type ack. Define the sequence s_1 of C, as follows

$$s_1 = send(p,d_1).send(p,d_2).send(q,a).send(q,a);$$

then, $S=\{s_1\} \cup \{r \mid r \text{ is a prefix of } s_1\}$. This definition of S implies that p sends the two data messages before q sends any ack message. To modify the protocol such that process q can send the first ack message after p sends its first data message, set S should be modified as follows: $S = \{s_1,s_2\} \cup \{r \mid r \text{ is a prefix of } s_1 \text{ or } s_2\}$, where $s_2 = send(p,d_1).send(q,a).send(p,d_2).send(q,a)$. In this example, a sequence which has send(q,a) as a prefix is an erroneous sequence. []

Notice that since receiving operations are not explicitly defined in the sending sequences, protocol specifications in this model can never suffer from deadlocks or unspecified receptions as characterized in [1] and [9].

Let s be a sequence of a communication protocol C=(P,T,S), i.e., either s = empty

or $\quad s = send(u_1,m_1). \; \ldots \; .send(u_n,m_n)$ for $n \geq 1$

where $u_1,\ldots,u_n$ are in P and $m_1,\ldots,m_n$ are messages whose types are in T. For $n \geq 1$, s can be rewritten in the following form

$$s = send(u_n,m_n,\ldots\; send(u_2,m_2,send(u_1,m_1,empty))\ldots)$$

In this form, "send" can be viewed as a function whose domain and range are as follows:

$$\text{send}: P \times M \times SUE_C \dashrightarrow SUE_C$$

where M is the set of all messages whose types are in T, and E_C is the set of all erroneous sequences of C. In other words, "send" maps a process u, a message m and a sequence r of C into a "bigger" sequence s of C such that s=send(u,m,r) or equivalently s=r.send(u,m). Function "send" is called a <u>constructor</u> since it can be used to construct a bigger sequence from a smaller one.

Lemma: Let s=send(u,m,r). <u>If</u> r is an erroneous sequence <u>then</u> s is an erroneous sequence.

Proof: Assume that r is an erroneous sequence, and that s is a sending sequence. Thus, s is in S but its prefix r is not in S contradicting the conditions of S. []

The inverse of this lemma is not necessarily true; i.e., if r is a sending sequence, then s may or may not be a sending sequence.

Let C=(P,T,S) be a communication protocol; and let M be the set of all messages whose types are in T. A <u>validation</u> <u>condition</u> "cond_C" for C is a predicate defined as follows:

$$\text{cond}_C: P \times M \times S \dashrightarrow \{\underline{\text{true}}, \underline{\text{false}}\}$$

where $\text{cond}_C(u,m,r)$ = <u>true</u> iff send(u,m,r) is in S. Notice that if r is an erroneous sequence "$\text{cond}_C(u,m,r)$" is undefined.

Theorem: Let C=(P,T,S) be a communication protocol; and let cond_C be a validation condition for C. For any sequence s of C, cond_C can determine whether or not s is in S.

Proof: Let s be a sequence of C; i.e.,

either $s = \text{empty}$

or $s = \text{send}(u_1,m_1). \ldots .\text{send}(u_n,m_n)$ for $n \geq 1$.

If s=empty then it is in S by definition. Otherwise, let $r_j=\text{send}(u_1,m_1). \ldots .\text{send}(u_j,m_j)$ for $1 \leq j \leq n$; r_j is a prefix of s. Therefore, s is in S iff (for all j=1,...,n) ($\text{cond}_C(r_j)$). Since $\text{cond}_C(u,m,r)$ is undefined for erroneous sequence then cond_C should be applied to r_i first, then it is applied to r_{i+1} iff $\text{cond}_C(r_i)$=<u>true</u>. []

From the above theorem, the set of sending sequences for a protocol C is completely defined by a validation condition of C. In the next section, we present a specification language to specify communication protocols based on the above model; most of the language features are dedicated to define validation conditions.

III. A SPECIFICATION LANGUAGE FOR PROTOCOLS

A general format for a protocol specification is as follows. (Reserved words are underlined; and numbers are added to the left for ease of reference.)

```
1.  Protocol <protocol name>

2.  process p,q
3.  msg typ $t_1,\ldots,t_k$
4.  init snd seq empty
5.  constructor  send : process x msg x seq --> seq

6.  state <declare state functions>

7.  declare r, s: snd seq; u : process
8.          $m_1$ : $t_1$ msg; ...; $m_k$ : $t_k$ msg;
9.  constructor rules
10.         s = send(u,$m_1$,r) iff <$cond_1$(u,$m_1$,r)>;
                 .
                 .
                 .
11.         s = send(u,$m_k$.r) iff <$cond_k$(u,$m_k$,r)>;
12. state rules
13.         <define state functions>

14. End <protocol name>;
```

The specification is divided into two sections, declarations (lines 1 to 8) and rules (lines 9 to 14). In the declaration section, the following are declared:

- the protocol´s name (line 1);
- the communicating processes in the protocol (line 2),
- the types of exchanged messages (line 2),
- the initial sending sequence denoted "empty" (line 4),
- the constructor function denoted "send" (line 5),
- some state functions (line 6), and
- some symbols which are used in the rules section. For example in the above declarations, symbols r and s are two sending sequences; u is a process; i.e., it is either p or q; and $m_1,\ldots,m_k$ are messages of types $t_1,\ldots,t_k$ respectively.

The rules section consists of constructor rules which define a validation condition for the protocol and state rules which define the state functions mentioned earlier. A constructor rule is of the form:

$$s = \text{send}(u,m_i,r) \ \underline{\text{iff}} \ \text{cond}_i(u,m_i,r)$$

where r and s are sending sequences, u is a process, and $\text{cond}_i(u,m_i,r)$ is a validation condition for a message m_i of type t_i, $i=1,\ldots,k$. In other words, a validation condition cond_C for the protocol can be expressed in terms of these cond_i's as follows:

```
cond_C(u,m,r) = case type(m) of
                t_1: cond_1(u,m,r)
                .
                .
                .
                t_k: cond_k(u,m,r)
                end case
```

where type(m) returns the type of message m. The predicates cond_i, $i=1,\ldots,k$, are defined in terms of the state functions in the specification.

The domain and range of a state function are defined in the declaration section (line 6) as follows:

<function name> : <domain> --> <rang>;

Each state function is defined by one state rule in the rules section; the definition is by recursion using the initial sending sequence "empty" and the constructor function "send". Such a definition is similar to a state function definition in the algebraic specifications of abstract data types [6] and [7].

IV. EXAMPLES

A. A Simple Window Protocol

Two processes p and q communicate by exchanging "data" and "ack" 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 positive integer.

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

A specification for this protocol has the following two state functions:

cntd(u,s):	gives the total number of data messages send by process u in the sending sequence s, and
cntk(u,s):	gives the total number of ack messages sent by process u in the sending sequence s.

The protocol can be specified in terms of a positive integer parameter N as follows.

```
Protocol window (N : integer)

process p,q
msg typ data,ack

init snd seq empty
constructor  send: process x msg x seq --> seq

state        cntd: process x snd seq --> integer
             cntk: process x snd seq --> integer

declare r,s: snd seq;
        u,v: distinct process; w: process;
          d: data msg; a: ack msg; m: msg;

constructor rules
     s = send(u,d,r) iff cntd(u,r) - cntk(v,r) < N;

     s = send(u,a,r) iff cntk(u,r) - cntd(v,r) < 0;

state rules
     cntd(u,s) = if s = empty
                    then 0
                    elsif s = send(u,d,r)
                       then cntd(u,r) + 1
                       elsif s = send(w,m,r)
                          then cntd(u,r);

     cntk(u,s) = if s = empty
                    then 0
                    elsif s = send(u,a,r)
                       then cntk(u,r) + 1
                       elsif s = send(w,m,r)
                          then cntk(u,r);

End window;
```

Notice that the two symbols u and v denote two distinct processes from the set {p,q}; and w denotes a process in the same set. Thus, if u,v, and w appear in a (constructor or state) rule, and if u denotes process p then v denotes q and w denotes either p or q. Otherwise, u denotes q, v denotes p and w denotes either p or q.

B. An 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 d is lost after being sent by a process, then the process will not receive an ack and it has to time itself out to resend message d again.

(iii) If an ack message is lost after being sent by the process, then the process will 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 cannot explicitly describe message loss and time-outs; rather, it specifies sending sequences which result from message loss and time-outs. For example, the sending sequence

... .send(p,d0).send(p,d0). ...

where d0 is a data0 message means that p may send the same data0 message twice (implying that the first message has been lost and p times itself out to sent it again). Also, the sending sequence

... .send(p,d0).send(q,a).send(p,d0). ...

where "a" is an ack message means that p may resend the last message even after q sends an ack message (implying that the ack has been lost after being sent). The sending sequence

... .send(p,d0).send(q,a).send(p,d1). ...

where d1 is a data1 message means a message transmission without loss. An example of an erroneous sequence is as follows:

... .send(p,d0).send(p,d1). ...

A specification for this protocol has the following two state functions:

lastd(u,s):	gives the last data message sent by process u in the sending sequence s, and
ack?(u,s):	is a boolean function which returns the value true iff process u has sent an ack after the other process has sent its last data message in the sending sequence s.

The protocol can be specified as follows.

```
Protocol alternating-bit

process p,q
msg typ data0, data1, ack

init snd seq empty
constructor  send: process x msg x seq --> seq

state        lastd: process x snd seq --> msg
              ack?: process x snd seq --> boolean

declare r,s: snd seq;
        u,v: distinct process; w: process;
         d0: data0 msg; d1: data1 msg; a: ack msg; m: msg;

construction rules
     s = send(u,d0,r) iff (lastd(u,r) = d0) or
                          (lastd(u,r) = d1 and
                          ack?(v,r));

     s = send(u,d1,r) iff (lastd(u,r) = d1) or
                          (lastd(u,r) = d0 and
                          ack?(v,r));

     s = send(u,a,r)  iff not ack?(u,r);

state rules
     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)
                          then lastd(u,r);

     ack?(u,s) = if s = empty or s = send(u,a,r)
                   then true
                   elsif s = send(v,d0,r) or s = send(v,d1,r)
                      then false
                      elsif s = send(w,m,r)
                         then ack?(u,r);
End alternating-bit;
```

V. A RUN-TIME CHECKING SYSTEM

Let C be a communication protocol; and assume that two communicating processes p and q are designed to exchange messages according to C. It is required to design a run-time checking system which observes the message exchanges between p and q to verify that they indeed follow C. Such a system can be used in testing p and q to detect their design errors; or it can be used to increase the system reliability against hardware or software failures.

Figure 1 shows an outline for the run-time checking system. It consists of two checking processes cp and cq placed between p and q. Processes cp and cq store the sending sequence constructed so far between p and q; let rp and rq denote the sending sequences stored in cp and cq respectively. When process p sends a message m_1, the message first goes to cp to check the protocol validity condition $cond_C(p,m_1,rp)$. If it is <u>false</u>, process cp sends an "err" message to cq; and both cp and cq reach an error state and stop. This should alert an external operator to stop the system, reinitialize it, or perform any other checking or correction action. On the other hand, if $cond_C(p,m_1,rp)=$<u>true</u>, then process cp sends a "null" message to process p so that p can resume execution. Then, cp sends m_1 to cq to update its current sending sequence rq and sends back a "null" message to process cp to update rp. Finally, process cq sends m_1 to process q; and process cp waits to receive the next message from p. Process cp can be defined as follows.

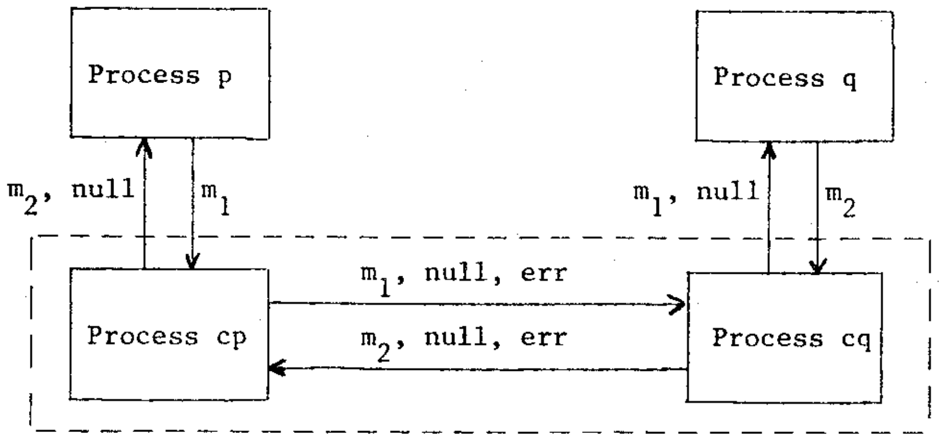

Figure 1. A run-time checking system for the communication protocol between p and q.

```
process cp;
var rp : snd seq init empty;

begin
loop
 [receive err from cq: ERROR

[]receive $m_2$  from cq: send $m_2$  to p;
                         send null to cq;

                         rp := rp.send(q,$m_2$)

[]receive $m_1$  from  p:
   if [not cond$_C$(p,$m_1$,rp): send err to cq; ERROR
      []cond$_C$(p,$m_1$,rp)    : send null to p;
                              send $m_1$  to cq;
                              if [receive err from cq : ERROR

                                 []receive null from cq :

                                    rp := rp.send(p,$m_1$)
                                 []receive $m_2$ from cq:
                                    rp := rp.send(p,$m_1$).send(q,$m_2$)
                                 ]fi
      ]fi

]end loop
end cp;
```

Note 1: The code for cq is similar to that for cp with minor changes: rp becomes rq; m_1 becomes m_2 and vice versa; and processes p and cq become q and cp respectively. There is one exception to this rule, namely the statement rp := rp.send(p,m_1).send(q,m_2)
in cp becomes rq := rq.send(p,m_1).send(q,m_2)
in cq. This ensures that both cp and cq update rp and rq (respectively) in exactly the same way. []

Note 2: Two assumptions are made in designing processes cp and cq. First, processes p and q are assumed to communicate indefinitely; thus, cp and cq never terminate. Second, all sent messages are assumed to reach their destinations without loss or corruption. Both these assumptions can be relaxed on the expense of making processes cp and cq more complicated. []

Note 3: Assume that at some instant, when rp=rq=r, process cp receives m_1 from p and cq receives m_2 from q. Assume also that $cond_C(p,m_1,r)$ = $cond_C(q,m_2,r)$ = <u>true</u>. Thus, cp and cq send null messages to p and q respectively and proceed to update rp and rq to become as follows.

$$rp = rq = r.send(p,m_1).send(q,m_2)$$

This implies that $cond_C(q,m_2,send(p,m_1,r))$ should be <u>true</u>. In other words, $cond_C$ should satisfy the following <u>parallelism condition</u> for any m_1 and m_2, and any sending sequence r:

$$cond_C(p,m_1,r) \wedge cond_C(q,m_2,r) ==> cond_C(q,m_w,send(p,m_1,r))$$

where $\wedge$ and ==> denote the logical "and" and "implication" respectively. As an example, consider the window protocol discussed earlier. To verify that the validation condition for this protocol satisfies the parallelism condition, four cases need to be considered.

Case 1 m_1 and m_2 are data messages d_1 and d_2 (respectively):

$$\begin{aligned}
& cond_C(p,d_1,r) \wedge cond_C(q,d_2,r) \\
==> \; & cond_C(q,d_2,r) \\
= \; & (cntd(q,r) - cntk(p,r) < N) \\
==> \; & (cntd(q,send(p,d_1,r)) - cntk(p,send(p,d_1,r)) < N) \\
= \; & cond_C(q,d_2,send(p,d_1,r))
\end{aligned}$$

Case 2 m_1 is a data message d and m_2 is an ack a:

$$
\begin{aligned}
& \mathrm{cond}_C(p,d,r) \wedge \mathrm{cond}_C(q,a,r) \\
\Longrightarrow\ & \mathrm{cond}_C(q,a,r) \\
=\ & (\mathrm{cntk}(q,r) - \mathrm{cntd}(p,r) < 0) \\
\Longrightarrow\ & (\mathrm{cntk}(q,r) - \mathrm{cntd}(p,r) - 1 < 0) \\
=\ & \mathrm{cond}_C(q,a,\mathrm{send}(p,d,r))
\end{aligned}
$$

Case 3 m_1 is an ack message a and m_2 is a data message d:

It is identical to case 2 except replace p with q and vice versa.

Case 4 m_1 and m_2 are ack messages a_1 and a_2 (respectively):

$$
\begin{aligned}
& \mathrm{cond}_C(p,a_1,r) \wedge \mathrm{cond}_C(q,a_2,r) \\
\Longrightarrow\ & \mathrm{cond}_C(q,a_2,r) \\
=\ & (\mathrm{cntk}(q,r) - \mathrm{cntd}(p,r) < 0) \\
=\ & \mathrm{cond}_C(q,a_2,\mathrm{send}(p,a_1,r))
\end{aligned}
$$

This completes the proof that cond_C for the window protocol satisfies the required parallelism condition. Thus, a run-time checking system with cp and cq as defined earlier can be constructed for this protocol. []

VI. CONCLUDING REMARKS:

The discussion in this paper can be extended to include the case of n (n>2) communicating processes $p_1,\ldots,p_n$. This requires to extend the definitions of the constructor function "send" and the validity condition "cond_C" to become as follows.

$$
\begin{aligned}
&\mathrm{send} : \underline{\mathrm{process}} \times \underline{\mathrm{msg}} \times \underline{\mathrm{process}} \times \underline{\mathrm{seq}} \longrightarrow \underline{\mathrm{seq}} \\
&\mathrm{cond}_C : \underline{\mathrm{process}} \times \underline{\mathrm{msg}} \times \underline{\mathrm{process}} \times \underline{\mathrm{snd}}\ \underline{\mathrm{seq}} \longrightarrow \underline{\mathrm{boolean}}
\end{aligned}
$$

Thus, $s = \text{send}(p_i,m,p_j,r)$ is the sequence generated from sequence r by process p_i sending a message m to process p_j; s is a sending sequence iff r is a sending sequence and $\text{cond}_C(p_i,m,p_j,r)$ = <u>true</u>.

In this case, the run-time checking system should be also extended to include n checking processes $cp_1,\ldots,cp_n$, where one cp_i is assigned to each communicating process p_i. Each cp_i stores a copy rp_i of the sending sequence constructed so far by the processes. When a process p_i sends a message m to process p_j, the checking process cp_i broadcasts m to all the other cp_k´s (including cp_j) so that they can all update their rp_k´s; then only cp_j delivers m to its process p_j. Since the checking processes always communicate in a broadcast fashion, it is recommended to connect them to a broadcast medium, e.g., a shared bus.

Although sending sequence specifications do not exhibit inconsistencies such as deadlocks or unspecified receptions, they may exhibit non-executable sequences (also a problem for communicating finite state machines [3] and [9]). For example, let r be a sending sequence for the two processes p and q in a protocol C; and assume that

(i) $\text{cond}_C(p,m_1,r)$ = <u>true</u>,
(ii) $\text{cond}_C(q,m_2,r)$ = <u>true</u>, and
(iii) $\text{cond}_C(p,m_1,\text{send}(q,m_2,r))$ = <u>false</u>.

From (i), it seems reasonable to assume that p can send m_1 when the sending sequence is r. But from (ii) and (iii), p shouldn´t do so because q might have sent m_2 prior to p sending m_1 (without p knowing it because of the transmission delay) causing an erroneous sequence. A similar situation occurs if (iii) is replaced by

(iv) $\text{cond}_C(q, m_2, \text{send}(p, m_1, r)) = \underline{\text{false}}$.

To verify that a sending sequence specification for some protocol doesn't exhibit non-executable sequences, one should prove that the specification satisfies the following property for any m_1, m_2, and r:

$$\begin{aligned} & \text{cond}_C(p, m_1, r) \wedge \text{cond}_C(q, m_2, r) \\ \Longrightarrow\ & \text{cond}_C(p, m_1, \text{send}(q, m_2, r)) \wedge \text{cond}_C(q, m_2, \text{send}(p, m_1, r)) \end{aligned}$$

This is stronger than the parallelism condition discussed in Section V. Still, it can be proved in a similar way as illustrated on the window protocol in Section V.

ACKNOWLEDGEMENTS: I am thankful to Dr. C. Sunshine for reading an earlier draft of this paper, to Professors Chandy and Misra for many helpful comments, and to K. F. Carbone for her careful typing.

REFERENCES

[1] G. V. Bochmann, "Finite state description of communication protocols," Computer Networks, Vol. 2, 1978, pp. 361-371.

[2] G. V. Bochmann and C. Sunshine, "Formal methods in communication protocol design," IEEE Trans. on Comm., Vol. COM-28, No. 4, April 1980, pp. 624-631.

[3] D. Brand and P. Zafiropulo, "On communicating finite state machines," IBM Research Report, RZ1053 (# 37725), Jan. 1981.

[4] A. Danthine, "Protocol representation with finite state models," IEEE Trans. Comm., Vol. COM-28, No. 4, April 1980, pp. 632-643.

[5] M. G. Gouda and Y. Yu, "Designing deadlock-free and bounded communication protocols," Tech. Rep. 179, Dept. of Computer Sciences, Univ. of Texas at Austin, June 1981.

[6] J. Guttag, "Abstract data types and the development of data structures," Comm. of the ACM, Vol. 20, June 1977, pp. 396-404.

[7] 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.

[8] C. A. Sunshine, "Formal modeling of communication protocols," USC/Inform. Science Institute, Research Report 81-89, March 1981.

[9] P. Zafiropulo, et. al., "Towards analyzing and synthesizing protocols," IEEE Trans. Comm., Vol. COM-28, No. 4, April 1980, pp. 651-661.