DISTRIBUTED STATE EXPLORATION FOR PROTOCOL VALIDATION

Mohamed G. Gouda

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

TR 185 October 1981

ABSTRACT

We discuss a distributed algorithm to validate a protocol layer by generating all reachable states and checking whether or not any of them is an error state, e.g., a deadlock state, an overflow state, ...etc. The algorithm is to be executed by an array of N processes ($N \geq 1$) which communicate exclusively by exchanging messages. The algorithm has a novel termination scheme based on a circulating token between the different processes. We discuss a correctness proof for the algorithm and its termination scheme.

KEYWORDS: Communication protocols, Distributed algorithms, Protocol validation, State exploration.

I. INTRODUCTION

State exploration is a constructive technique to validate communication protocol layers. It is based on generating and checking all reachable states of a protocol layer starting from a specified initial state [5] and [6]. State exploration can be used in conjunction with other techniques to completely verify communication protocols [1] and [2]. It can be also used during the synthesis of such protocols to ensure their correctness [9]. Some automated tools for state exploration have already been developed (e.g. [3] and [6]) and applied with useful results to real protocols [8].

A potential problem of state exploration is due to the possibly large number of states of a protocol layer. If the number of states is large, state exploration becomes slow and expensive to use. To remedy this, we develop in this paper a distributed algorithm for state exploration. Implementing such an algorithm on a network of communicating microcomputers can speed up protocol validation while reducing its cost.

In section II, a simple model of a protocol layer is presented. For conciseness, we limit the discussion to this model even though the developed algorithm can be applied to more complex models. In section III, a distributed algorithm for state exploration is discussed in detail; then its termination mechanism is discussed in section IV. Concluding remarks about the algorithm efficiency and applicability are in section V.

II. A SIMPLE MODEL OF A PROTOCOL LAYER

A _protocol layer_ consists of two processes which communicate by exchanging messages through two channels. A _channel_ is a fixed-capacity FIFO queue which stores the messages sent by one process until they are received without errors by the other process; two channels are needed to "transmit" messages in both directions between two processes. A _process_ is a directed labelled graph where each edge is labelled send(m) or receive(m) for some m in a finite set of message names. One of the nodes in a process is identified as its _initial node_.

Associated with each process is a _control token_ which at any instant resides at one node in the process. Initially, the control token of a process is at its initial node. If the control token of a process R is at a node with an output edge e labelled send(m) and if the output channel of R is not full, then a message m can be added to the output channel and the control token of R traverses e in zero time to the next node in R. If the control token of R is at a node with an output edge e labelled receive(m) and if the head message in the input channel of R is m, then m can be removed from the input channel and the control token of R traverses e in zero time to the next node in R.

The _state of a protocol layer_ at a time instant t is defined by (i) the positions of the control tokens of its two processes at t, and

(ii) the contents of its two channels at t. The *initial state* s_o is when the control token of each process is at its initial node, and the two channels are empty.

Some of the states of a protocol layer are error states. An *error state* can either be a deadlock state, an overflow state, or an unspecified reception state. The exact definitions of these error states are irrelevant to this paper; and so they are not discussed here. An interested reader can find these definitions in [1] and [9].

Let s and s′ be two states of a protocol layer L. s′ is *next* to s if the following two conditions are satisfied: (i) s is not an error state. (ii) Starting from state s, if exactly one process in L sends or receives exactly one message, then the state of L becomes s′.

A state s′ is *reachable from* state s if either s=s′ or there are states $s_1,\dots,s_r$ such that $s=s_1$, $s'=s_r$, and s_{i+1} is next to s_i for $i=1,\dots,r-1$. A state s is *reachable* if it is reachable from the initial state s_0.

One of the goals of a protocol layer validation is to check that no error state is reachable. This can be achieved by generating all reachable states and checking whether any of them is an error state. This is possible since the two channels in the layer have finite capacities. This technique is called *state exploration*.

III. DISTRIBUTED STATE EXPLORATION

Let L be a protocol layer which consists of two processes R and S and two finite channels of capacities m and n. We describe a distributed program which uses R, S, m, and n to generate all reachable states of L and determines whether or not any of them is an error state. The program uses an array of N processes P[i=0..N-1] which communicate exclusively through messages. An "interface" process P[N] starts the computation then receives the final result 'error' or 'noerror' when the computation terminates.

The following assumptions are made about the processes P[i:0..N]: (i) Each process can send messages to any other process and to itself. Messages are transmitted without errors. (ii) Each process has one unbounded input channel to hold all incoming messages until they are received by the process. (iii) Each process has a complete knowledge about R, S, m, and n. This restriction can be relaxed somewhat; see section V.

The basic idea of the distributed program is to partition the nodes of one process, say R, in L into N disjoint partitions A_1, A_2,...,A_N. This also partitions the state space of L into N disjoint partitions. Assign each process in the process array to explore one of the partitions of the state space. So, if s is a state where the control token of R is at a node in partition A_i, then s is to be handled by process P[i], i=0..N-1. This partitioning and process assignment can be defined by a function PRS(s) which returns the index value i for the process P[i] assigned to handle state s. We assume that each process in the process array knows the function PRS(s).

The basic construct in the code of P[i=0..N-1] is as follows:

```
receiveloop [ new(st):  EXPLORE
            [] err(st):  stop
             ] endloop
```

Thus, each process P[i] is always waiting to receive a message. If it receives a "new(st)" message, it explores the state "st" using the code block "EXPLORE" then waits to receive the next message. If it receives an "err(st)" message, it stops. The computation starts by the interface process P[N] sending the initial state s_0 to P[PRS(s_0)].

Block "EXPLORE" uses a permanent data structure which consists of the following three variables:

"st"	is the current state being examined by P[i].
"nxtst"	is a set of all states next to the current state "st".
"oldst"	is a set of all previously generated states whose next states have already been generated by P[i].

Block "EXPLORE" also uses the following two functions:

"ERROR(st,R,S,m,n)"
uses the definitions of R, S, m, and n to return a _true_ value if the current state "st" is an error state; otherwise it returns a _false_ value.

"NEXT(st,R,S,m,n)"
returns the set of all states next to the current state "st".

The code for block "EXPLORE" is as follows:

```
EXPLORE::begin
           if ERROR(st,R,S,m,n)
              then for j=0..N do send err(st) to P[j]

              elsif not (st in oldst)
                 then oldst  := oldst + [st];
                      nxtst  := NEXT(st,R,S,m,n);
                      for st in nxtst do
                         send new(st) to P[PRS(st)]
                 endif
              endif
       end
```

Correctness of the above program is established by the following theorems. (Proofs are in the Appendix.)

Theorem 1: Each generated state in the process array is a reachable state in the protocol layer.

Theorem 2: If no error state is reachable in the protocol layer, then all reachable states are generated in the process array.

Theorem 3: If an error state is reachable in the protocol layer, then an error state is generated in the process array and all the processes in the array stop in a finite time.

IV. TERMINATION

So far, processes in the process array will stop only if an error state is reachable. If no error state is reachable, then each of the processes will reach an indefinite waiting state for "new(st)" messages that never arrive. If this global waiting state can be detected, then the interface process P[N] can conclude that no error state is reachable in the protocol layer being validated.

One way to detect this global waiting state is to have a "token(k:<u>integer</u>)" message circulating in a fixed order between the processes in the process array [4]. When a process P[i] receives "token(k)" and recognizes that it has received a "new(st)" message after it has received "token(k)" last, it resets k to zero and passes "token(k)" on to P[i+1 mod N]. On the other hand, if P[i] recognizes that it has not received any "new(st)" messages since the last time it has received "token(k)", it increments k and sends "token(k)" to P[i+1 mod N]. A global waiting state for the process array is detected when k=N.

The correctness of the above scheme is based on the following assumption concerning <u>message transmission delay</u>. The following is true for any $i_1,...,i_r$ in {0,...,N-1} where $r \geq 2$. If a process $p[i_1]$ sends a message m_1 to $P[i_r]$ then sends a message m_2 to $P[i_2]$ which on receiving m_2 sends m_3 to $P[i_3]$ which on receiving m_3 sends m_4 to $P[i_4]$ and so on until finally m_r is sent to $P[i_r]$, then m_1 reaches the input channel of $P[i_r]$ before m_r. Notice that if r=2, the assumption becomes as follows. If $p[i_1]$ sends two messages m_1 then m_2 to $P[i_2]$ then m_1 reaches the input channel of $P[i_2]$ before m_1. Later, this transmission delay assumption is used to verify the above termination scheme. Now, we characterize this scheme more formally by defining process P[i] as follows:

```
process P[i=0..N-1];
    msg new, err, noerror, token;
    type state = ...;
    var st : state;
        oldst, nxtst : setof state;
        more : boolean;
        j, k : 0..N;
begin
    oldst := [ ];
    more := true;
    receiveloop [  new(st) : more := true;
                             EXPLORE
                [] err(st) : stop
                [] token(k) : if more
                                then more := false; k := 0;
                                     send token(k) to P[i+1 mod N]
                                elsif k < N-1;
                                   then k := k+1;
                                        send token(k) to P[i+1 mod N]
                                   else for j = 0..N do
                                         send noerror to P[j]
                                   endif
                                endif
                [] noerror : stop
                ]  endloop
end.
```

When P[i] receives a "token(k)" message and recognizes that it has not received any "new(st)" messages since the last time it has received a "token(k)" message, then it sends a "noerror" message to all the processes in the array and to the interface process P[N]. The computation starts by P[N] sending "new(s_0)" and "token(0)" messages to P[PRS(s_0)] where s_0 is the initial state of the given protocol layer. P[N] can be defined as follows:

```
process P[N];
    msg new, err, noerror, token;
    type state = ...
    var st : state;

begin
    st := INITST(R,S);
    send new(st) to P[PRS(st)];
    send token(0) to P[PRS(st)];
    receive [  err(st) : ...
            [] noerror : ...
            ]
end.
```

To verify the above termination scheme, we need to prove that a global waiting state is reached in the process array P[i:0..N-1] iff a global waiting state is detected using the circulating "token(k)" mechanism. The proof is based on the following three lemmas. (Proofs are in the Appendix.)

Lemma 1: If a global waiting state is reached in the process array, then a global waiting state will be detected using the circulating "token(k)" scheme.

Lemma 2: A global waiting state is detected in the process array iff the "token(k)" message makes a complete cycle during which each process in the array recognizes that it has received the "token(k)" message two successive times without receiving any "new(st)" messages in between.

Lemma 3: If a global waiting state is detected using the circulating "token(k)" scheme, then a global waiting state has been reached in the process array.

The proof of Lemma 3, in the Appendix, is based on Lemma 2. From Lemmas 1 and 3, the following theorem is immediate.

Theorem 4: A global waiting state is detected using the circulating "token(k)" scheme iff a global waiting state has been reached in the process array.

V. CONCLUDING REMARKS

To uniformally distribute the work load over the N processes in the process array, the state space of L should be equally partitioned. Since the partitioning of the state space of L is achieved by partitioning the nodes in one process R in L, each of the partitions of R should have the same number of nodes as a first approximation. Also, it is preferable that the nodes in any partition in R be "adjacent". This is to decrease the number of messages between different processes in the process array while increasing the number of messages from each process to itself. Notice that messages from one process to itself can be avoided completely on the expense of making the code of P[i] little more complicated. A "good" partitioning of the nodes in R can cause a speed-up by a factor of N; finding such a good partitioning is a point for further research.

Each process P[i] uses the definition of R only in evaluating the two functions ERROR(st,R,S,m,n) and NEXT(st,R,S,m,n). In either case, "st" is a state which is to be handled by P[i], i.e., PRS(st) = i. Therefore, P[i] does not need to know all R; it only needs to know the following: (i) the nodes in one partition A_i of R (see section III); (ii) any node which can be reached by a directed edge in R from a node in A_i; and (iii) all the edges in R which start from nodes in A_i.

To decrease the number of "token(k)" exchanges in the system, each process P[k] should not receive the "token(k)" message from its input channel until there are no "new(st)" messages in this channel. Similarly, to speed up termination whenever an error state is detected, each P[i] should not receive any "new(st)" or any "token(k)" messages from its input channel if there is an "err(st)" message in this channel. The "err(st)" message should be received first causing P[i] to stop.

The above distributed algorithm can be applied to more complex models of protocol layers. For instance it can be extended easily to the case where the layer has more than two communicating processes. Similarly, it can be extended to the case where the communication medium exhibits some transmission errors, e.g., message loss or corruption. This is done by modelling the nonideal communication medium as an additional process which may lose or corrupt received messages before resending them as described in [9].

ACKNOWLEDGEMENTS

The author is thankful to Mr. Yao-Tin Yu for helpful discussions. He is also thankful 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. 361-372.

[2] G. B. Bochmann and J. Gecsei, "A unified model for the specification and verification of protocols," Proc. IFIP Congress, 1977, pp. 229-234.

[3] J. Hajek, "Automatically verified data transfer protocols," Proc. Int. Comp. Conf., 1978, pp. 749-756.

[4] G. Leland, "Distributed systems - towards a formal approach," Proc. IFIP Congress, 1977, pp. 155-160.

[5] C. A. Sunshine, "Interprocess communication protocols for computer networks," Ph.D. dissertation, Dept. of Comp. Sci. Stanford Univ., Stanford, CA, 1975.

[6] C. A. Sunshine, "Formal modeling of communication protocols," USC/Information Sciences Institute Tech. Report 81-89, March 1981.

[7] C. H. West, "An automated technique of communications protocol validation," IEEE Trans. Commn., Vol. COM-26, pp. 1271-1275, Aug. 1978.

[8] C. H. West and P. Zafiropulo, "Automated validation of a communications protocol: The CCITTX.21 recommendation," IBM J. Res. Develop., vol. 22, pp. 60-71, Jan. 1978.

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

APPENDIX: PROOFS OF THEOREMS AND LEMMAS

Proof of Theorem 1: States are generated in the process array as follows. First, the initial state s_0 is generated at P[N] and sent to P[PRS(s_0)]. Then, other states are generated using the function "NEXT(st,R,S,m,n)" where state "st" has already been generated. Thus, for every generated state s, there exist states s_0, s_1,...,s_r of the protocol layer such that s_0 is the initial state, s_r=s, and s_{i+1} is next to s_i for i=0,...,r-1. Therefore, s is a reachable state of the given protocol layer. []

Proof of Theorem 2: Assume that no error state is reachable in the protocol layer. From Theorem 1, no error state is generated in the process array; i.e., for any generated state "st" in the process array ERROR(st,R,S,m,n) is false. From the code of EXPLORE, no process P[i] sends any "err(st)" messages. Thus no process P[i] ever stops; and each of them continues to receive and process "new(st)" messages sent to it.

Let s be a reachable state in the protocol layer; i.e., there are states s_0,...,s_r of the protocol layer such that s_0 is the initial state, s_r=s, and s_{i+1} is next to s_i for i=0,...,r-1. s_0 is generated at the interface process P[N] and is sent to P[PRS(s_0)]. Then, s_1 is generated at P[PRS(s_0)] and is sent to P[PRS(s_1)], and so on until finally s_r (i.e., s) is generated at P[PRS(s_{r-1})]. []

Proof of Theorem 3: Let s be a reachable error state of the protocol layer; i.e., there exist states $s_0,\ldots,s_r$ of the protocol layer such that s_0 is the initial state, $s_r=s$, and s_{i+1} is next to s_i for $i=0,\ldots,r-1$. There are two cases; either each s_j is generted at $P[PRS(s_{j-1})]$ for $j=1,\ldots,r$ or there exists k in $\{1,\ldots,r-1\}$ such that when s_k is generated and sent to process $P[PRS(s_k)]$ to generate its next states, this process has already stopped after receiving an earlier err(st) message. In the first case, the error state s_r (or s) is generated; in the second case, another error state "st" is generated.

Assume that an error state s is generated in the process array and is sent to P[PRS(s)] for processing. From the code of EXPLORE, P[PRS(s)] detects that s is an error state (i.e., ERROR(s,R,S,m,n) is true) and sends err(s) messages to all the processes in the array. Thus all processes stop in a finite time. []

Proof of Lemma 1: Assume that a global waiting state is reached in the process array; i.e. no more "new(st)" messages are exchanged between the processes P[i:0..N-1]. Thus, only the message "token(k)" is circulated between the processes. In a finite time, each P[i] recognizes that it has received "token(k)" two successive times without receiving any new(st) messages in between and increments k in token(k). When k=N a global waiting state is detected in the process array. []

<u>Proof of Lemma 2</u>:

<u>If Part</u>: Assume that "token(k)" makes a complete cycle during which each process recognizes that it has received "token(k)" two successive times without receiving "new(st)" messages in between. During this cycle each P[i:0..N-1] increments k; thus k=N at the end of this cycle and a global waiting state is detected.

<u>Only If Part</u>: Assume that a global waiting state is detected; i.e., k=N in the circulating "token(k)" message in the process array. Since each process either increments "k" or sets it to zero, all the processes must increment k during one complete cycle for k to reach the value N. []

<u>Proof of Lemma 3</u>: Assume that a global waiting state is detected in the process array. From Lemma 2, "token(k)" has finished a complete cycle during which each process has recognized that it has received "token(k)" two successive times without receiving any "new(st)" messages in between. Call this cycle the <u>critical cycle</u>. If all the processes in the process array do not receive any "new(st)" messages after receiving "token(k)" during its critical cycle, then Lemma 3 is true. Otherwise, let P[u] be the first process to receive a "new(st)" message after receiving "token(k)" in its critical cycle. Assume that this "new(st)" message is sent to it by process P[v]. P[v] cannot send this message after it has received "token(k)" in its critical cycle otherwise P[v] (and not P[u]) is the first process to receive a "new(st)" message after it has received token(k) in its critical cycle. Thus, P[v] must send this "new(st)" before receiving token(k)

in its critical cycle. Because of this critical cycle definition, P[v] must send this "new(st)" message before receiving "token(k)" two successive times with no "new(st)" messages in between. In other words, P[v] sends a "new(st)" message to P[u] followed by a "token(k)" message to P[v+1 mod N] which eventually reach P[u] before the "token(k)" message. This contradicts the transmission delay assumption. Thus, no P[u] can receive a "new(st)" message after receiving the "token(k)" message in its critical cycle and Lemma 3 is true. []