DEADLOCK ABSENCE PROOFS FOR NETWORKS OF COMMUNICATING PROCESSES *

K.M. CHANDY and J. MISRA
Computer Sciences Department, University of Texas, Austin, TX 78712, U.S.A.

Received 15 June 1979; revised version received 16 August 1979

Processor networks, synchronisation, deadlock avaoidance

1. Network model

A network is a finite collection of processes which communicate with one another exclusively through messages as in Hoare's model [5] and similar models [1,6]. Our model is motivated by the work of Hoare; it will be helpful if the reader is familiar with this work. A network may be represented logically as a labeled graph where the vertices are processes and the edges are communication links. There may be several edges between a pair of processes corresponding to many communication links. We assume a simple protocol for message transmission as in [5] and [1]: a message is transmitted along a link only when the two processes at both ends of the link are waiting to communicate along that link.

A process is said to be *executable* if some statement in the process can be executed. Note that if processes h_i and h_j are waiting to communicate along an edge e, then both processes are executable since the statements causing message transmission can be executed. A process is said to be *blocked* if it is not executable and it is waiting to communicate along one or more links. A blocked process h waiting on a set of edges $\bar{e} = \{e_1, e_2, ...\}$ must continue to wait on each one of the edges in $\bar{e}$ until a message is transmitted along at least one edge in $\bar{e}$ (after which it may wait on any arbitrary, and possibly null, set of edges). Processes which are neither executable nor blocked are said to be *terminated*; a process which has terminated cannot later become executable nor blocked.

* This research was supported by NSF Grant MCS77-09812.

As in Hoare's model, a process may have parallel constructs which allow it to execute a statement and simultaneously wait on an edge. Note that a process which can execute a statement is executable in our model regardless of whether it is simultaneously waiting on some edge.

1.1. Process state transitions

A process may be in one (and only one) of three states: executable, blocked and terminated. We now discuss the three possible state transitions:

(1) Executable to blocked: A process transits from executable to blocked when no statement in the process can be executed and if the process is waiting (to send or receive a message) on an edge e and the process at the other end of e is not waiting on e.

(2) Blocked to executable: A blocked process must be waiting on a non-empty set of edges $\bar{e}$; the process will transit to an executable state when a process at the other end of any edge e of $\bar{e}$ starts waiting on e.

(3) Executable to terminated: A process transits from executable to terminated when it executes a special statement such as a HALT statement at which point it ceases execution and ceases waiting on edges (if any) that it was waiting on.

A process cannot transit from blocked to terminated. A terminated process cannot change states.

There are references to suspended and unborn processes in the literature, where suspended processes are temporarily in a quiescent state waiting to be invoked, and unborn processes are created later by invocations from other processes. In our model suspended and

unborn processes are treated as blocked processes because a suspended process is waiting for a message which will invoke it, and an unborn process may be thought of as waiting for a message to create it.

In practice there may be buffers on the communication links. In our model, each buffer is a process, and a *communication link itself does not have buffers.* Thus a buffered communication link between processes h_i and h_j will be represented in our model as two unbuffered links connecting h_i and h_j to a buffer process.

Our model can be used to prove properties of programs which are written in Hoare's [5] notation. Note that Hoare's notation is very powerful and has been used to describe distributed data base and communication protocols [4], and can be used to describe resource scheduling and a number of situations arising in concurrent computation.

2. Deadlock

A set of processes $\bar{h}$ in a network N is said to be *deadlocked at some stage in the computation* if and only if

Deadlock conditions:

(1) *termination condition*: not all the processes in $\bar{h}$ have terminated and

(2) *executability condition*: no process in $\bar{h}$ is executable and

(3) *closure condition*: if h_i in $\bar{h}$ is waiting on edge e, and e is incident on h_j, then h_j is in $\bar{h}$.

If $\bar{h}$ is deadlocked, then no process in $\bar{h}$ can proceed with execution and at least one process in $\bar{h}$ will never terminate. N is said to be *deadlock-free at some stage* of the computational process if no set of processes $\bar{h}$ is deadlocked at that stage. *N is deadlock-free* if it is deadlock-free at every stage in every computation sequence.

To prove that a network is deadlock-free at some stage we shall assign a number called priority g(e) to each edge e, and show that the priorities satisfy certain conditions; the existence of such priorities will be shown to be necessary and sufficient for the absence of deadlock.

Proper priority conditions:

Priorities g(e) are said to be *proper* for N at some stage if for every process h in N at that stage, h is either

(1) *executable* or

(2) *terminated* with $g(e) = \infty$ for all edges e incident on h or

(3) *blocked*, waiting on *at least* the set of edges $\bar{e}$ where,

(i) $\bar{e} = \{e_j \mid g(e_j) \leq g(e)$, for any e incident on h$\}$ and

(ii) $g(e_j)$ is finite if $e_j \in \bar{e}$.

Thus if a network is waiting properly on priorities g, then every blocked process must be waiting on every edge of minimum priority incident on it; in addition it may wait on other incident edges.

Theorem 1. N is deadlock-free at any stage of computation if and only if there exists a proper set of priorities g(e) at that stage.

Proof. We first show that the assumption of a proper g(e) and a deadlocked set of processes $\bar{h}$ leads to a contradiction. Let e^* be the edge of minimum priority among all edges incident on processes in $\bar{h}$. Since all processes in $\bar{h}$ cannot have terminated, $g(e^*)$ is finite. From the closure condition for deadlock, and the waiting condition for proper priorities, the processes on either end of e^* must be waiting on e^*, and are therefore executable, whence $\bar{h}$ is not deadlocked.

We next present an algorithm for assigning proper priorities g(e) to all edges e given that N is deadlock-free at some stage; this algorithm also assigns numbers called labels to the processes. Let label(h) be the label for process h.

Algorithm

Initialization: label all executable processes 0 (zero);

Iteration: **while** there exists an edge e incident on h_i, h_j where h_i is unlabeled and blocked and waiting on e, and h_j is labeled **do**
  label (h_i) := label (h_j) + 1;
  g(e) := label (h_i)
enddo;

Termination: for every edge e not already assigned a priority set $g(e) = \infty$.

We now outline a proof that the algorithm will terminate with proper priorities on every edge:

(1) The iteration step terminates because the number of labeled processes increases by one in each iteration and there are a finite number of processes.

(2) We now show by contradiction that every non-terminated process is labeled. Assume the contrary, i.e. no deadlock and there exists an unlabeled process h_i which has not terminated.

Since h_i is unlabeled it follows from the initialization step that it is blocked. Suppose h_i is waiting on some edge e, whose other end is incident on a process h_j. If h_j were labeled, then h_i could be labeled; hence h_j must be unlabeled. Thus the set of unlabeled processes must be deadlocked since this set satisfies the executability, termination and closure conditions. Contradiction!

(3) Every edge is given a priority because of the termination step.

(4) Every edge is assigned a priority only once – obvious.

(5) The assigned priorities are proper. It follows from the iteration step that a process labeled m, where $m > 0$, must have one and only one edge incident on it with priority m, and all other edges incident on this process must either have a priority of $m + 1$ (assigned by the iteration step) or a priority of infinity (assigned on the termination step); furthermore, the process labeled m must be waiting on (at least) the single edge with priority m incident upon it. Hence the priorities are proper for blocked processes. Edges incident on only terminated processes are given (infinite) priorities only on the termination step, because only non-terminated processes are labeled in the initial and iteration steps. Hence the priorities are proper for terminated processes as well. Thus the priorities assigned by the algorithm are proper.

This completes the proof of the theorem.

Note that the theorem merely says that there exists a set of proper priorities; it does not imply that every set of priorities for deadlock-free networks is proper. In fact, it is possible to have a set of priorities that is not proper for a set of processes that is deadlock-free.

Corollary 1. Let g be a set of proper priorities over a network. If at some stage for some e, g(e) is finite, then the network has at least one executable process.

Proof. Let e be the edge for which g(e) is minimum and hence finite. Suppose e is incident on h_i, h_j. Since the priorities are proper and g(e) is finite, h_i, h_j have not terminated. Hence if neither of h_i, h_j is executable, then both of them must be blocked; in this case, from the proper priority condition, they must both be waiting on e and hence executable. Contradiction!

This corollary is useful in showing that computation must be proceeding within a network.

Recall that a network is deadlock-free if it is deadlock-free at every stage during all possible computations. We can therefore prove absence of deadlock in the following manner. At every stage we show either

(i) by direct, obvious reasoning that there exists an executable process or

(ii) a set of proper priorities, which implies the existence of an executable process if all processes have not terminated.

Note that the priority assigned to an edge will in general change from stage to stage; we emphasize that ours is a *dynamic* priority scheme in contrast to static priority schemes. We also emphasize that deadlock-free computation implies the existence of a set of dynamic priorities whereas the same cannot be said for static priorities. It may seem that the priorities have to be contrived. This is not so. Our experience shows that dynamic priorities are naturally related to the problem at hand. The naturalness of dynamic priorities is the most important issue in this methodology. An elegant method of constructing deadlock-free networks is to first postulate a set of dynamic priorities and then design the network to wait properly on these priorities. We give a brief example of such a methodology applied to an important practical problem: distributed simulation. Readers interested in deadlock absence proofs of communication and distributed data base protocols and other related problems may refer to [4].

Example (for more details, see [3]). Consider a network of real-time processes which communicate exclusively through messages. Whether a process h sends a message m along an edge e at time t depends only upon the sequences of messages received along the input edges of h up to time $t - \epsilon$, where ϵ is a positive constant. An example of such a network is a job-shop where processes are work-stations and messages are

jobs. We will represent a work-station as a black-box with one or more input edges and one or more output edges. Jobs arrive and depart along edges after being processed in a First-Come-First-Served (FCFS) manner by the work-stations. Let a_i be the time of the ith arrival to the work-station, d_i the time of ith departure, and s_i the processing time for the ith job. Then because of the FCFS discipline,

$$d_i = s_i + \max\{a_i, d_{i-1}\} \,, \quad \text{Convention: } d_0 = 0. \tag{1}$$

Suppose we wish to carry out a distributed simulation of this network by a network of asynchronous (non-real time) processes which we call a *logical* network of *logical* processes. Each physical process (PP) in the real-time network is simulated by a logical process (LP). The event 'PP_i sends a message m to PP_j at time t' will be simulated by LP_i sending LP_j a message which is a 2-tuple (t, m); our objective is to allow asynchrony by encoding time into every message. Assume that we wish to simulate the real-time network for an indefinite period. How can we design a logical network which is guaranteed not to deadlock?

The first thing that a designer must do is to settle upon a set of dynamic priorities. The obvious dynamic of an edge is the time to which it has been simulated, i.e. the t-component of the last tuple (t, m) transmitted along that edge (which is 0 (zero) by convention when no messages have been transmitted along that edge). The time to which an edge has been simulated is called its *clock-value*. A reasonable choice for the dynamic priority of an edge is its clock-value. This choice of dynamic priorities implies that a process waits on all edges of minimum clock-value when it is blocked.

To clarify these concepts let us once again consider the job-shop example in detail. A message in the logical system has the form (t, m) where m is an integer representing the number of jobs which traverse the edge at time t, assuming that all jobs are of the same type. Consider a work-station h. Let the nth job have departed h at or before time t. Then, if the $(n + 1)^{th}$ job arrives at or before t, we can predict the departures, if any, on *all* output edges up to time:

$$d_{n+1} = s_{n+1} + \max\{a_{n+1}, d_n\} \,. \tag{2}$$

If the $(n + 1)^{th}$ job arrives after t, we can predict that

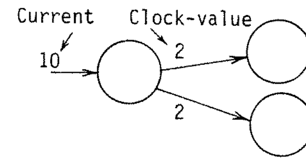

Fig. 1. Logical process showing clock-values of edges.

there will be no departures on any output edge up to time

$$t' = s_{n+1} + t \,. \tag{3}$$

Consider the example shown in Fig. 1. The logical process must wait on the output edges because they have minimum clock-values. Assume that the next departure occurs at time t = 15 (this departure must occur after time 10 from (2)), along the upper edge. Since we are also waiting on the lower edge we must send some message along it which increases its clock-value; this forces us to invent a new message (t, 0) which implies that no message was sent in the physical network between the time equal to the current clock-value and t.

We are now in a position to discuss the waiting behavior of logical processes. An LP always waits on all edges of minimum clock-value. If it is waiting on a set of edges $\bar{e}$ it continues waiting until message transmission has been completed along *all* the edges in $\bar{e}$. For any message (t, m) received along an input edge, the LP uses (2) and (3) to process the message. If it is waiting on an output edge, the LP outputs some message whose m-component may be 0 (zero). This waiting condition is sufficient to guarantee absence of deadlock.

References

[1] K.M. Chandy and J. Misra, An axiomatic proof technique for networks of communicating processes, Technical Report 98, Department of Computer Sciences, University of Texas, Austin, TX (1978).

[2] K.M. Chandy and J. Misra, Proving temporal properties for networks of communicating processes, Technical Report, Department of Computer Sciences, University of Texas, Austin, TX (1978).

[3] K.M. Chandy and J. Misra, Distributed simulation: A case study in design and performance of distributed systems, IEEE Trans. Software Engrg., to appear.

[4] K.M. Chandy, J. Misra, L. Myers and P. Verman, Proofs of absence of deadlock in distributed programs and protocols, Technical Report, Department of Computer Sciences, University of Texas, Austin, TX (1979).

[5] C.A.R. Hoare, Communicating sequential processes, Comm. ACM 21 (8) (1978) 666–677.

[6] G. Kahn, The semantics of a simple language for parallel programming, in: J.L. Rosenfeld, Ed., Information Processing 74 (North-Holland, Amsterdam, 1974) 471–475.