

Termination Detection of Diffusing Computations
in Communicating Sequential Processes

K. M. Chandy and J. Misra

TR-144

April 1980

Abstract

A symmetric communication protocol, as in communicating sequential processes, may lead to deadlock where some processes are waiting to send and others are waiting to receive. Deadlock or termination detection in such a system cannot be based directly on the Dijkstra-Scholten scheme for termination detection. We propose a generalization of the Dijkstra-Scholten scheme for communicating sequential processes. Our scheme is shown to be efficient.

1. Introduction

In [2] Dijkstra and Scholten introduce the notion of diffusing computation in a distributed system of processes and suggest an elegant algorithm for detecting the termination of an arbitrary diffusing computation in an arbitrary network. The generality of the solution makes it suitable for application in a number of problems arising in distributed processing. Dijkstra [3] gives a solution to the problem of determining if a process is in a knot, using this algorithm.

In this paper, we adapt their scheme to detect deadlock (and/or proper termination) in a system of communicating sequential processes [5,8]. Dijkstra and Scholten assume that a process P_1 can send a message to another process P_2 whenever it wishes to do so; thus a process is never permanently blocked waiting to send. We however assume the protocol proposed by Hoare [5], i.e. a message can be sent from process P_1 to process P_2 only if P_1 is waiting to send to P_2 and P_2 is waiting to receive from P_1 . Thus a process may have to wait indefinitely to output as well as to input. This difference in protocol leads to significant changes in the solution. Francez [4] has considered the distributed termination problem with Hoare's protocol. His approach is radically different from that of Dijkstra and Scholten, and ours in that his solution is predicated upon preanalysis of the topology and construction of a spanning tree. One advantage of Francez's approach is that it allows arbitrary pairs of processes to communicate spontaneously without any of them having received prior messages.

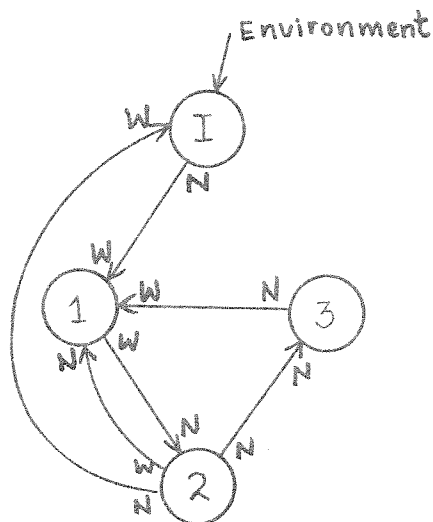
2. Problem Definition

We are given:

- (1) a set S of communicating sequential processes and
- (2) a process outside S , which we call the environment.

A computation is started when the environment communicates with some process I in S ; we call I the initiator. It is required to design a signalling scheme to be superimposed on message communication which will guarantee that the initiator will send a single signal to the environment when, and only when, computation in S has ceased. We also use "node" for a process.

Example 1



I: Initiator

$S = \{I, 1, 2, 3\}$, the set of processes

An arc from process i to process j denotes that i can send messages to j , and j can receive messages from i . An arc from i to j labeled W at i denotes that i is waiting to send to j ; similarly if the arc is labeled W at j it denotes j is waiting to receive from i ; N denotes not waiting.

In example 1, the set of processes S is deadlocked. (Note that process 3 has terminated and is not waiting to communicate.) If the initiator receives a message from the environment it may, for instance, start to wait to send to 1 and the computation may restart.

Consider a possible scenario. I may begin waiting to send to 1. A message is transmitted from I to 1. As a result of this message 1 waits to receive from 2 which results in a message being sent from 2 to 1. Now 2 waits to receive from 3 and 1, while 1 waits to receive from 2 and 3, resulting in another deadlock.

Note that 2 resumes computation as a result of sending a message. In the Dijkstra-Scholten model the only way a process can resume computation is by receiving a message.

3. Solution

A communicating sequential process as in Hoare [5], is either executing, waiting or terminated. An executing process is carrying out some computation. A process is said to be waiting if it is not executing and is waiting to communicate. A terminated process has completed computation. A waiting process may become executable as a consequence of communicating a message. A waiting process cannot change its status until it communicates.

A waiting process may be blocked or unblocked. A waiting process P is said to be blocked if for every process Q such that P is waiting to send to (receive from) Q , Q is not waiting to receive from (send to) P . It is convenient to think of terminated processes as being blocked. A blocked process P waiting to communicate with Q becomes unblocked if Q begins to wait on P .

A signalling scheme for informing a process whether it is blocked is discussed in section 5. This kind of signal will be called a B-signal to distinguish it from another kind of signal, A-signal, defined in section 3.1. Signal will refer to both A and B- signals. We assume in the next section that a process can determine its status: executing, blocked or unblocked.

3.1 A- Signalling Scheme

The history of message communication and A- signalling is captured by an activity graph, where the nodes are the processes and arcs (called activity arcs) summarize the history of communication and A- signalling. The implementation of the activity graph is discussed in section 4. All communication dealing exclusively with maintaining the activity graph will be called signals to distinguish this form of overhead communication from message communication inherent to the computations of processes.

Activity arcs are created in the following way: if processes i and j communicate and if there are no activity arcs currently between i and j (in either direction), a pair of activity arcs is created - one from i to j and the other from j to i . There are no signals involved in arc creation.

Activity arcs are destroyed in the following way: an arc from i to j is destroyed by j sending an A-signal to i . Note that i cannot destroy an arc from i to j -- only j can. Furthermore, note that communications of messages result in (possible) creation of activity arcs and communications of A- signals result in (definite) deletion of activity arcs. In order for the scheme to work, a blocked process must be able to send and receive signals; hence the term blocked only refers to messages in the underlying

computation and not to signals.

We adopt the convention that the environment remains passive waiting for an A- signal from I during the computation of S. The environment carries out no computation at all. We assume that the environment becomes blocked immediately after communicating with the initiator, and remains blocked indefinitely thereafter. Hence after the first message communication with I, two activity arcs (environment, I) and (I, environment) are created and (I, environment) is subsequently deleted by the environment.

A process is said to be engaged if and only if it has some activity arc incident on it. Following the first communication with the environment, the initiator is engaged. A process that is not engaged is disengaged. When a disengaged process i communicates with a process j , a pair of activity arcs (i,j) and (j,i) is created, thus engaging process i ; the activity arc (j,i) is defined to be a tree arc. Thus a tree arc (x,y) is created when a disengaged node y becomes engaged, as a result of communication with x . Activity arcs, other than tree arcs, are non-tree arcs.









All message transmissions carried out by a process must be done sequentially as in Hoare [5]. Hence status changes in a process (such as from blocked to unblocked, or disengaged to engaged) which are implemented by transmissions of signals must also be carried out sequentially.

Every process implements the following rules for deletion of arcs.

- PI1) A process that is blocked waits to delete all incoming, non-tree arcs (by sending A- signals).
- PI2) A blocked process with only one incident arc (which will be shown to be an incoming tree arc) waits to delete that arc.
- PI3) Every process having outgoing arcs waits to receive A- signals (which result in the deletion of outgoing arcs).

Example 2

Consider example 1. The following set of activity arcs are created by the communication described in example.

COMMUNICATION AND SIGNALLING	ACTIVITY GRAPH
Environment send message to I	(tree arcs are solid edges) ENVIRONMENT 
I sends message to 1. Tree arc (I,1) created; Non-tree arc (1,I) created; The environment gets blocked and hence deletes (I,environment)	ENVIRONMENT 
2 sends message to 1 Tree arc (1,2) created; Non-tree arc (2,1) created;	ENVIRONMENT 
1 is blocked; hence it deletes (2,1), an incoming non-tree arc.	ENVIRONMENT 
I is blocked; it deletes (1,I), an incoming non-tree arc.	ENVIRONMENT 
2 is blocked; it deletes the incoming tree arc (1,2) because it is blocked and has no other incident arcs	ENVIRONMENT 
1 is blocked; it deletes the incoming tree arc (I,1) because it is blocked and has no other incident arcs	ENVIRONMENT 
I is blocked; it deletes (Environment,I)	ENVIRONMENT 

From the above it is deducible that:

- PD1) Every engaged process has exactly one incident incoming tree arc.
- PD2) A process must be blocked at the instant following its disengagement.
- PD3) Assume that the underlying communications scheduler is fair. Then from PI1 and PI2, every indefinitely blocked process will delete all incoming non-tree arcs in a finite time.

3.2 Properties of the Activity Graph

Lemma 1: If there is a communication between processes i and j then at least one of i and j must be engaged immediately prior to the communication.

Proof: Assume that there is a communication between i and j where both i and j are disengaged. We show that this assumption leads to a contradiction.

A blocked node does not change the set of lines it is waiting on. A disengaged node is blocked at the point of disengagement (by PD2). Since i, j communicate they must both be unblocked just prior to communication. Hence one of them (say j) must have begun to start waiting for the other following its last disengagement. This can only happen if j has had a communication following its last disengagement because j was waiting at the point of disengagement and hence must continue waiting for communication with the same set of processes until a communication occurs. This communication will cause j to be engaged, contradicting the assumption that j was disengaged at the time of communication with i .

Theorem 1: The set of engaged nodes and the set of tree arcs form a rooted directed tree where the initiator is the root (and the paths are directed away from the root). This tree is called the engagement tree.

Proof: By induction on the number of message communications.

The lemma holds initially since the initiator is the only engaged process following the first communication. Assume that the lemma holds after n message communications. The next message communication must be between (1) two engaged processes or (2) an engaged process and a disengaged process by lemma 1.

In the former case no tree arc is created (because a tree arc is created only when a disengaged node becomes engaged). In the latter case a tree arc is created from the engaged process to the disengaged process, thus adding the newly engaged node to the tree. Hence the lemma holds after the $n + 1$ th communication.

Lemma 2: If all processes are indefinitely blocked then after a finite time all processes will become disengaged.

Proof: After all processes become blocked no new arcs are created because arcs are created only when messages are communicated and blocked processes cannot communicate. After a finite time period all non-tree arcs must be deleted by PD3. We show that the number of engaged processes will decrease in finite time after all non-tree arcs are deleted. By lemma 2, all engaged processes are on the engagement tree. Consider a process which is a leaf in the engagement tree. This process has no outgoing arcs because (by assumption) all outgoing non-tree arcs have been deleted and the process has no outgoing tree arcs because it is a leaf. By PI2 PI3 and the assumption of fair scheduling, this process must delete the single incoming tree arc and thus disengage itself in finite time.

Theorem 2: The A- signal is sent to the environment by the initiator a finite time after the computation ceases.

Proof: It follows from lemma 2 that the initiator sends a signal to the environment a finite time after the computation ceases. Furthermore, after the initiator sends the signal, all nodes must be disengaged (from theorem 1). By lemma 1, no communication can take place among disengaged nodes. Therefore, the computation must have ceased when the initiator sends the signal.

4. Implementation of the Activity Graph

For any engaged process x define the following:

$\text{father}(x)$ is a singleton set consisting of the process from which the single incoming tree arc of x emanates.

Note that $\text{father}(\text{initiator}) = \text{environment}$.

$\text{pred}(x)$ is a set of processes such that $j \in \text{pred}(x)$ if and only if there is a non-tree arc (j,x) in the activity graph.

$\text{succ}(x)$ is a set of processes such that $j \in \text{succ}(x)$ if and only if there is an arc (x,j) in the activity graph.

Every arc (i,j) in the activity graph is thus represented at both processes i and j : $j \in \text{succ}(i)$ and $i \in \text{pred}(j) \cup \text{father}(j)$.

4.1 A- Signalling Logic of the i th Process

(a) Following every message communication with process j :

Operation

If $\text{father}(i) = \emptyset$

then (* establish a tree arc from j and a non-tree arc to j *)

father := {j};

succ := {j};

pred := \emptyset

```

else (* tree arc to i already exists *)
  if j  $\notin$  father(i)  $\cup$  pred(i)  $\cup$  succ(i)
    then (* no arc exists between i and j *)
      pred := pred  $\cup$  {j}
      succ := succ  $\cup$  {j}
      (* established non-tree arcs (i,j) and (j,i) *)
    end-if
  end-if
end-if

```

(b) Upon receipt of an A- signal from j where $j \in \text{succ}(i)$:

Note 1: $\text{succ}(i) \neq \emptyset$.

Note 2: Immediately after transmitting this signal j will remove i from $\text{father}(j) \cup \text{pred}(j)$ and it is i's responsibility to remove j from $\text{succ}(i)$ thus deleting arc (i,j).

Operation $\text{succ}(i) := \text{succ}(i) - \{j\}$

(c) when i is blocked and $\text{pred}(i) \neq \emptyset$:

Operation (* delete incoming non-tree arc *)

send A- signal to some $j \in \text{pred}(i)$;

$\text{pred}(i) = \text{pred}(i) - \{j\}$

(d) when i is blocked and $\text{pred}(i) = \emptyset$ and $\text{succ}(i) = \emptyset$:

Operation (* delete single incoming tree arc thus disengaging i *)

send A- signal to the $j \in \text{father}(i)$;

$\text{father} := \emptyset$

In Hoare's [5] notation the conditions (a), (b), (c) and (d) will be guards in a single repetitive command, and the corresponding operations are the command lists associated with corresponding guards. PI1, PI2, PI3 are implemented by condition (c), (d), and (b) respectively.

4.2 Overhead Estimation

Define an active period of a process as the continuous time span during which it remains unblocked. Define an idle period as the time during which it is continuously blocked. The number of incoming arcs that can be set up during an active period of a process is the number of different processes that it communicates with during that period. Thus the number of incoming arcs to any process during an active period is bounded by n , the total number of processes. The total number of A- signals sent by a process during its idle periods (no A- signal is sent during an active period) is bounded by the number of incoming arcs. Hence total number of A- signals sent by a process is bounded by the number of its active periods times n . It is obvious that the total number of A- signals cannot exceed the number of messages transmitted in a system.

5. Implementation of Blocking Signals in CSP

We assumed earlier that a process knows whether it is blocked or unblocked. Whether a process is blocked or unblocked depends on the waiting status of its neighbouring processes. Hence it is necessary to have B- signals which inform neighbouring processes of the waiting status one process. It is, however, impossible for a process to deduce the instant at which it changes from blocked to unblocked or vice versa, if a signalling scheme is used. Define a local variable in process i called think-blocked(i) which i sets to true if i thinks it is blocked and false otherwise. Note that think-blocked(i) may be true when i is actually unblocked and vice versa. The program for process i , in particular conditions (a), (b), (c) (d) must be based upon think-blocked rather than whether the process is truly blocked, so as to be implementable in Hoare's notation. We will

show that our algorithm is correct even though $\text{think-blocked}(i)$ may not be consistent with i 's true blocking situation.

With a send/receive pair of communicating processes (i,j) we associate $r(i,j)$ and $s(i,j)$ which denote whether j is waiting to receive a message from i , and i is waiting to send a message to j . $r(i,j)$ can only be affected by a change in the waiting status of j . Similarly $s(i,j)$ can only be affected by i . We implement r and s as follows. $r(i,j)$ is a local boolean variable in process i . Process i is always waiting to receive a B- signal from process j updating $r(i,j)$. Process j waits to send a B- signal to process i updating $r(i,j)$ when $r(i,j)$ is inconsistent with process j 's waiting status. Fairness in the underlying scheduler will ensure that $r(i,j)$ cannot be inconsistent for an indefinite period. The operations to maintain $s(k,i)$ are similar.

$\text{Think-blocked}(i)$ is set to true if for every process j that i is waiting to send a message to, $r(i,j) = \text{false}$ and for every process k that i is waiting to receive a message from $s(k,i) = \text{false}$.

There are two possible inconsistencies in the value of $\text{think-blocked}(i)$, since following the last transmission of a B- signal, a process may have changed its waiting status.

- (i) process i may have $\text{think-blocked}(i) = \text{true}$ even though i is actually unblocked.
- (ii) Process i may have $\text{think-blocked}(i) = \text{false}$ even though i is actually blocked.

The only requirements for the correct operation of the signalling scheme are PI1, PI2 and PI3. PI3 is maintained true by all processes whether they are blocked or unblocked. From PI1 and PI2, it is sufficient

for correctness that a process that remains indefinitely blocked must have `think-blocked = true` within a finite time of the instant that it actually becomes blocked. This requirement is met since, as we have remarked earlier, $r(i,j)$ and $s(k,i)$ cannot remain inconsistent for an indefinite period, assuming that the underlying scheduler is fair.

Note that inconsistency (i) is irrelevant to the correctness of the signalling scheme! Correctness merely requires that inconsistency (ii) cannot exist for an indefinite period.

6. Discussion

It is sometimes convenient to run a network of processes until deadlock; deadlock is then broken by the environment and the processes are allowed to run until the next deadlock. This repetition of deadlock and breaking deadlock is more efficient in some cases [1] than avoiding deadlock altogether. Levin [6] gives an elegant algorithm which is guaranteed to compute the correct results when the network is deadlocked.

References

1. K. M. Chandy, and J. Misra, "Diffusing Simulation: Parallel simulation via Diffusion Computation," University of Texas, Computer Sciences Department, Austin, TX 78712 (1980).
2. E. W. Dijkstra, C. S. Scholten, "Termination Detection for Diffusing Computation," EWD 687a, Plataanstraat 5, 5671 Al Neunen, The Netherlands.
3. E. W. Dijkstra, "In Reaction to Ernest Chang's "Deadlock Detection"," EWD 702, Plataanstraat 5, 5671 Al Neunen, The Netherlands.
4. N. Francez, "Distributed Termination," ACM Transactions on Programming Languages and Systems, Vol. 2, No. 1, Jan. 1980.
5. C. A. R. Hoare, "Communicating Sequential Processes," Commun. of ACM, Vol. 21, No. 8, August 1978.
6. G. Levin, Doctoral Dissertation, Computer Science Department, Cornell University, Ithaca, NY (1980).
7. R. B. Kieburtz and A. Silberschatz, "Comments on "Communicating Sequential Processes," " ACM TOPLAS, Vol. 1, No. 2, October 1979.