Termination Detection of Diffusing Computations in Communicating Sequential Processes

JAYADEV MISRA and K. M. CHANDY
The University of Texas at Austin

In this paper it is shown how the Dijkstra-Scholten scheme for termination detection in a diffusing computation can be adapted to detect termination or deadlock in a network of communicating sequential processes as defined by Hoare.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Program Verification-*correctness proofs*; D.3.3 [**Programming Languages**]: Language Constructs—*concurrent programming structures*; D.4.1 [**Operating Systems**]: Process Management—*deadlocks*

General Terms: Languages, Verification

Additional Key Words and Phrases: distributed systems, networks of processes, termination detection, diffusing computation

1. INTRODUCTION

Dijkstra and Scholten [4] have introduced 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 any network. The generality of the solution makes it suitable for application to a number of problems arising in distributed programming. Using this algorithm, Dijkstra [3] gives a solution to the problem of determining whether a process is in a knot. We have applied a variation of this algorithm to compute shortest paths in weighted, directed networks [2].

In this paper, we show how Dijkstra and Scholten's scheme can be used to detect deadlock (and/or proper termination) in a system of communicating sequential processes [6]. This deadlock detection algorithm has found use in distributed simulation [1].

We assume the protocol proposed by Hoare [6]; that is, 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 send as well as to receive.

This protocol is different from that used by Dijkstra and Scholten. They

This research was supported in part by the National Science Foundation under grant MCS-79-25383 and by ARPA Systems Modeling Parts II and III under grant N00039-78-G-0080.
Authors' address: Department of Computer Sciences, College of Natural Sciences, The University of Texas at Austin, Austin, TX 78712.

assume that a process can send a message whenever it likes. In their model a process never has to wait to send a message.

Since our scheme is based on the Dijkstra–Scholten algorithm (abbreviated to DSA) and on Hoare's work on communicating sequential processes (abbreviated to CSP), we assume familiarity with these papers [4, 6].

In DSA, network computation terminates when every process in the network becomes idle; an idle process does not send a message unless it receives one. A process can decide *locally* whether it is idle. Thus, if all processes determine that they are idle, then the computation has terminated. On the other hand, in CSP a process may not send a message either because it is idle (as in DSA) and has nothing to send or because the intended recipient is not waiting to receive the message. In the latter case the sender is *unable to determine locally* whether it can ever send the message, because that depends on the state of the receiver as well. Hence, processes must communicate their waiting status to their neighbors (i.e., processes they can communicate with) every time they change waiting status. However, a process's perception of its neighbor's waiting status may be *inconsistent* with its neighbor's true status, because the neighbor may not yet have informed it about the neighbor's change of state. The contribution of this paper is to present a modification to DSA which detects termination despite this inconsistency.

Francez [5] also has considered the distributed termination problem with Hoare's protocol. His approach is radically different from that of Dijkstra and Scholten 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

A communicating sequential process as in Hoare [6] is either *executing, waiting,* or *terminated.* An executing process is one which can carry out some computation. A process is waiting if it cannot execute and is waiting to communicate. A terminated process has completed computation and hence is neither executing nor waiting. A waiting process may become executing following a communication. 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 messages to Q (receive messages from Q), Q is not waiting to receive messages from (send messages to) P. A blocked process P waiting to communicate with Q becomes unblocked if Q begins to wait for P. A blocked process cannot change its status to unblocked unilaterally. *We choose to consider terminated processes as being permanently blocked and executing processes as being unblocked.* Computation has ceased in a network if and only if every component process is blocked.

The following are given:

(1) a network S of communicating sequential processes in which computation has ceased, and
(2) a process outside S, called the *environment.*

A new computation is started when the environment communicates with some process I in S; we call I the *initiator.* It is required to design a signaling scheme to be superimposed on message communication which guarantees that the initiator will send a single signal to the environment in finite time after (and only after) computation in S has ceased. (The term "blocked" refers only to messages in the underlying computation and not to signals. A blocked process can send and receive signals but not messages.)

3. OVERVIEW OF THE SIGNALING SCHEME

We use two kinds of signals: *A-signals* or *Activity signals,* which correspond to the signals in DSA, and *B-signals* or *Blocking signals,* used by a process to inform a neighbor[1] that it has changed its waiting status (from waiting-to-send/receive to not-waiting-to-send/receive or vice versa) for that neighbor.

3.1 A-Signals

We present a method of implementing DSA by creating/deleting arcs rather than by the use of counters. This implementation is logically equivalent to the use of "cornets" in DSA. An *activity graph* consists of nodes which represent processes and arcs (called *activity arcs*) which summarize the history of communication and A-signaling. (An activity arc (i, j) can be maintained by i having j in a list of its activity arc successors and j having i in a list of its activity arc predecessors.) If processes i and j communicate by a message 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. An arc from i to j is destroyed by j sending an A-signal to i (destruction of this activity arc corresponds to i and j updating their activity arc successor and predecessor lists, respectively). There are no activity arcs initially.

We adopt the convention that the environment remains passive waiting for an A-signal from I during the computation of S. We assume that the environment becomes blocked immediately after communicating with the initiator and remains blocked indefinitely thereafter. A process other than the environment is said to be *engaged* if and only if it has some activity arc incident on it. A process that is not engaged is *disengaged.* Initially, the environment is engaged, and all other processes are 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.* Activity arcs other than tree arcs are *nontree* arcs. The tree arc (j, i) remains a tree arc until it is deleted by i sending an A-signal to j. Similarly, the nontree arc (i, j) remains a nontree arc until it is deleted by j sending an A-signal to i.

3.2 B-Signals

B-signals are used by processes to inform their neighbors of their waiting status. Every process i knows whether it is waiting to send messages to, or receive messages from, any neighbor j. Process j maintains local Boolean variables $r_j(i)$

[1] Process i is a neighbor of process j if it is possible for processes i and j to communicate.

(and $s_j(i)$) which are **true** when j "thinks" i is waiting to receive from (send to) j. Process j's information about process i, ($r_j(i)$, $s_j(i)$), may be incorrect; this happens when process i changes its waiting status for process j and has not yet informed process j of this change. When process j receives a B-signal from process i, it updates $r_j(i)$ and $s_j(i)$ appropriately. We associate a local Boolean variable *think-blocked*(j) with process j; *think-blocked*(j) is **true** if (1) process j is not executing, (2) for every process i that j is waiting to receive from, $s_j(i)$ is **false**, and (3) for every process k that j is waiting to send to, $r_j(k)$ is **false**. *think-blocked*(j) denotes whether or not process j "thinks" it is blocked; as r_j's and s_j's may be inconsistent with the true waiting status of the neighbors of process j, *think-blocked*(j) may also be inconsistent with the true blocking situation of process j. Note that, from the definition, *think-blocked*(j) is **true** if j is terminated.

In earlier schemes [4, 5] there was no such inconsistency between the true status of a process and the status that it thought it was in. We show below that the entire signaling scheme is correct even though the algorithm is based upon (possibly inconsistent) *think-blocked* variables. The inconsistency regarding *think-blocked* variables does not occur in DSA because each process can determine its status locally.

4. RULES OF PROCESS OPERATION

The original computation (including waiting behavior, message transmission, and execution sequences) is not affected by signaling. The only way messages affect the signaling is by the possible creation of activity arcs (as given in Section 3.2) following a message transmission.

A nonexecuting process waits at all times to receive A- and B-signals. Hence there can be no blocking for signal transmission. Receipt of either type of signal results in immediate modification of the local variables of a process (r, s, or the lists of activity arc successors), after which the value of the signal can be discarded and the process waits to receive further signals and messages.

A process's signal transmission is carried out according to the following rules.

Rule 1 (waiting condition for transmission of B-signal). Process i waits to send a B-signal to process j if and only if $r_j(i)$ or $s_j(i)$ is inconsistent with process i's true waiting status. (Note that i can deduce $r_j(i)$, $s_j(i)$ from the B-signals that it has already sent.)

Rule 2 (waiting condition for transmission of A-signal; deletion of nontree arc). Process j waits to send an A-signal to process i, where (i, j) is a nontree arc, if *think-blocked*(j) is **true**.

Rule 3 (waiting condition for transmission of an A-signal; deletion of tree arc). Process j waits to send an A-signal to process i, where (i, j) is a tree arc, if and only if (1) *think-blocked*(j) is **true**, (2) there is no other incident (i.e., outgoing or incoming) activity arc on process j, and (3) process j has ensured (by sending appropriate B-signals) that, for every neighbor k, $r_k(j)$ and $s_k(j)$ truly reflect the waiting status of process j.

Initial Condition. $r_j(i)$ and $s_j(i)$, for every i, j, truly reflect the waiting status of process i for process j. The environment is engaged and all other processes are disengaged.

We show that these rules result in correct signaling to the environment.

Note. A process may wait simultaneously to send and receive A-signals, B-signals, and messages. The algorithm makes no assumption about which transmission takes place first. In particular, r, s may be out-of-date for some processes. The *only* requirement is that a process cannot send an A-signal deleting a tree arc unless it has corrected all its neighbors' expectations about its own waiting status by sending B-signals.

5. CORRECTNESS OF THE PROPOSED SCHEME

We use the following two observations for the proofs.

Observation 1. A message communication between processes i and j cannot change the engaged/disengaged status or the waiting status of any process other than i or j; both i and j are engaged immediately after the communication.

Observation 2. A-signals that delete nontree arcs and all B-signals have no effect on the engaged/disengaged status or the waiting status of any process.

THEOREM 1

(1) *The waiting status of every disengaged process i is truly reflected in* $r_j(i)$, $s_j(i)$ *for every neighbor j of i.*

(2) *Two disengaged processes cannot be waiting to communicate by messages with each other.*

PROOF. The proof is by induction on n, which is the number of messages plus the number of A-signals and B-signals transmitted in the network.

Initially ($n = 0$), part (1) holds from the initial conditions described in Section 4; part (2) holds because the only communication possible initially is between the environment and the initiator and the environment is engaged.

Assume that the theorem holds for $n \le N$. If the $(N + 1)$th transmission is a message, the theorem follows trivially from Observation 1. If the $(N + 1)$th transmission is an A-signal that deletes a nontree arc, the theorem follows trivially from Observation 2. From the induction hypothesis and Rule 1, disengaged processes cannot send B-signals. Hence the theorem holds if the $(N + 1)$th transmission is a B-signal. Now consider the case where the $(N + 1)$th transmission is an A-signal that deletes a tree arc. Suppose the A-signal was sent from process j to process i, causing process j to become disengaged. Part (1) is guaranteed by Rule 3. By the induction hypothesis, for every disengaged neighbor k of process j, $r_j(k)$, $s_j(k)$ truly reflect k's waiting status prior to the $(N + 1)$th transmission. The $(N + 1)$th transmission does not alter $r_j(k)$, $s_j(k)$, or k's waiting status. Hence $r_j(k)$, $s_j(k)$ correctly reflect k's waiting status after the $(N + 1)$th transmission. Since *think-blocked*(j) is **true** when j disengages itself, it follows that immediately after the $(N + 1)$th transmission j and k cannot be waiting to communicate with one another. This proves part (2). □

THEOREM 2. *The set of engaged nodes and the set of tree arcs form a rooted directed tree of which the environment is the root.*

PROOF. The proof follows as in DSA by induction on the number of messages. □

We call this tree the *engagement tree.*

THEOREM 3. *The environment receives an A-signal within a finite time after the computation ceases; at that point all processes are disengaged.*

PROOF. By Rule 2, within a finite time after computation ceases $r_j(i)$, $s_j(i)$ truly reflect the waiting status of process i for process j, for all i, j. Therefore, *think-blocked*(j) is set to **true**, for all j, within a finite time after computation ceases. Hence, the leaves of the engagement tree disengage themselves within a finite time after computation ceases by Rules 2 and 3. The theorem follows, as in DSA, by induction on the height of the tree. □

The correctness of the signaling scheme follows from part (2) of Theorem 1 and from Theorem 3.

6. DISCUSSION

As in DSA, it is possible to give a bound on the number of signals transmitted during a computation in which n messages are transmitted. An A-signal deletes an activity arc, which could only be created through a message transmission. Hence the total number of A-signals cannot exceed the number of message transmissions. B-signals are sent only when a process changes its waiting status. The waiting status of a process can change only if that process sends or receives a message. Therefore, process j sends no more than $(M_j * N_j)$ B-signals, where M_j is the number of message transmissions in which process j participates and N_j is the number of neighbors of process j. A simple upper bound on the number of B-signals is the number of messages times the number of processes in the network. The exact number of A- and B-signals are difficult to estimate as they depend not only on the number of messages but also on the number of times *think-blocked* becomes **true** for various processes. There are many problems for which *think-blocked* becomes **true** relatively infrequently; in these cases the signaling overhead is low.

It should be noted that we do not require that the underlying scheduler for message and signal transmission be fair; that is, we do *not* assume that a message or signal is transmitted within finite time, if both the sender and the receiver are waiting to communicate. We make the following weaker assumption: if there are one or more sender-receiver pairs waiting to communicate, then one of them communicates in finite time.

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.

ACKNOWLEDGMENTS

We are grateful to R. Kieburtz, S. Owicki, A. Silberschatz, and C. S. Scholten for comments on this paper. Constructive comments of the referees helped considerably in improving the exposition.

REFERENCES

1. Chandy, K.M., and Misra, J. Asynchronous distributed simulation via a sequence of parallel computations. *Commun. ACM 24*, 11 (April 1981), 198–206.
2. Chandy, K.M., and Misra, J. On diffusing computations. Tech. Rep. TR-150, Dep. Computer Sciences, Univ. Texas Austin, July 1980.
3. Dijkstra, E.W. In reaction to Ernest Chang's "Deadlock Detection." EWD 702, Plataanstraat 5, 5671 AL Nuenen, Netherlands.
4. Dijkstra, E.W., and Scholten, C.S. Termination detection for diffusing computations. *Inf. Process. Lett. 11*, 1 (Aug. 1980), 1–4.
5. Francez, N. Distributed termination. *ACM Trans. Program. Lang. Syst. 2*, 1 (Jan. 1980), 42–55.
6. Hoare, C.A.R. Communicating sequential processes. *Commun. ACM 21*, 8 (Aug. 1978), 666–677.

Received April 1980; revised October 1980 and June 1981; accepted June 1981