Information Processing Letters 31 (1989) 203–208
North-Holland

22 May 1989

AN EFFICIENT DISTRIBUTED TERMINATION TEST

Friedemann MATTERN

Department of Computer Science, University of Kaiserslautern, P.O. Box 3049, D 6750 Kaiserslautern, Fed. Rep. Germany

Communicated by T. Lengauer
Received 2 November 1987
Revised 4 December 1988

Keywords: Termination detection, distributed algorithm, CSP, distributed computing

1. Introduction

In recent years, a surprising number of distributed termination detection algorithms with various characteristics have been presented. One of the most elegant solutions is derived stepwise together with an invariant and is due to Dijkstra, Feijen and van Gasteren [9]. Discussions of the principle and variants can be found in [1,14,15]. We present a similar algorithm which detects termination faster under the assumption that whenever a (synchronous) message is sent from some process P_i to another process P_j, an acknowledgement carrying a one-bit status information is sent in the opposite direction. This can usually be done at negligible cost without an explicit message.

We consider n processes $P_1, \ldots, P_n$ $(n \geq 2)$, each being either *active* or *passive*. In an underlying computation, the processes cooperate by exchanging *synchronous messages* (so called *basic messages*). The computation satisfies the following conditions [14]:

(1) only active processes may send messages,
(2) a process may change from passive to active only on receipt of a message,
(3) a process may change from active to passive spontaneously.

When *all* processes are passive, a stable state has been reached and the underlying computation is said to have *terminated*. In contrast to [9,14], we are not only interested in a method by which a given process P_n is enabled to *detect* termination when it has occurred, but in an efficient control algorithm which enables P_n to determine whether the underlying computation

(a) has terminated before the control algorithm finished, or
(b) had *not* yet terminated when the algorithm was started.

If the underlying computation terminates while the control algorithm is running, either result is acceptable. We call an algorithm with these properties a *termination test*. Such an algorithm can be regarded as a specialization of a general stable property detection algorithm for distributed systems [8]. Notice that since nontermination is not a stable property, it is impossible to state that a distributed computation is still active "now". We assume that the underlying computation is started before the termination test.

As opposed to a termination *detection* algorithm which, once started, remains active until the system has terminated, a termination *test* algorithm can be reactivated in specific situations (e.g., timer driven) when required by the environment or the underlying computation. This might be a more reasonable and economic use in practice.

Notice that, trivially, every termination detection algorithm can be used as a (possibly nonterminating) termination test algorithm which

never reports a "negative" result. Those detection algorithms which proceed in *rounds* (e.g., [4,12]) may also detect case (b). On the other hand, any termination test algorithm can easily be transformed into a detection algorithm (with possibly unbounded message complexity) by simply restarting the algorithm after a negative result.

2. The DFG algorithm

Although the algorithm in [9] is not efficient when used as a termination test, we use it as a starting point by presenting a version with a slightly changed set of rules, to allow for the early announcement of a "negative" result. We assume that for the purposes of the termination test, a *control token* circles around a ring P_n, $P_{n-1}, \ldots, P_1$, P_n. Notice that being passive does not prevent a process from sending or receiving the token. We further assume that the token and the processes can be either *black* or *white*; initially, all processes are supposed to be white. The following rules determine the algorithm:

***Rule* 1.** A process sending a basic message to a recipient with a higher index than its own becomes black.

***Rule* 2.** P_n, when passive, may initiate a test by sending a white token to P_{n-1}.

***Rule* 3.** An active process keeps the token until it becomes passive.

***Rule* 4.** A passive process P_i $(i \neq n)$ which has the token propagates a black token if P_i or the token is black, otherwise it propagates a white token.

***Rule* 5.** A process transmitting the token becomes white.

***Rule* 6.** If P_n receives a white token, it announces termination.

***Rule* 7.** If P_n receives a black token for the first time, it starts a second round.

***Rule* 8.** If P_n receives a black token the second time, it announces the failure of the test.

Rules 1–5 (and implicitly also Rule 6) are merely a paraphrase of the original rules of [9], their correctness will not be considerred further here. (P_0 was renamed P_n. While this might delay the detecttion of termination when a message is sent to P_n that has no effect, it simplifies Rules 6–8.)

To see why a second round is necessary in most cases (Rule 7), a time diagram (Fig. 1) is useful. According to Rule 1, P_2 becomes black when sending message (a). If this were the last message of the underlying computation and all processes were passive shortly afterwards, the system would already be terminated at the time instant the "current" control round is started. The token then becomes black when passing by P_2, resulting in a *false alarm*. A second round is guaranteed to return a white token, since the first round made all processes white (Rule 5). To avoid false alarms, an algorithm has to distinguish case (a) from case (b) where a message is sent to a node which has already been visited by the token in the *current* control round.

3. An efficient variant

According to Rule 3, active processes delay the propagation of the token. While this is acceptable for termination *detection* (in every control round except the last one, at least one basic message is exchanged, thereby bounding the number of token passes), it slows down the termination *test*. Especially to enable a fast negative response in the case the computation is not terminated, Rule 3 could be replaced by the following rule.

***Rule* 3′.** An active process P_i $(i \neq n)$ which has the token propagates a black token.

When the token encounters an active process, a second confirmation round (Rule 7) is not necessary. The changes of the rules which take into account this possibility are straightforward.

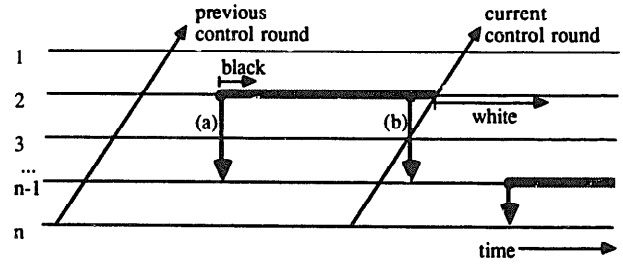

Fig. 1.

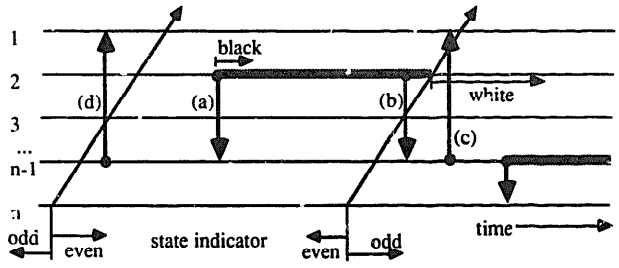

Fig. 2.

We now show how to modify the algorithm in order to enable a test in *one single round* (even if the token did not encounter an active process) such that Rule 7 and Rule 8 can be replaced by the following one.

Rule 7'. If P_n receives a black token it announces the failure of the test.

The general idea of the improvement is simply that (according to the comments concluding the last section) a process has to register precisely those messages it sends to other processes which have already been visited by the token in the current control round. These are necessarily processes with a higher index. For that purpose a *binary state indicator* with values 0 (or "even") and 1 ("odd") is postulated to exist in each process which is initialized to 0. To achieve that the control token changes the value of the state indicator (see Fig. 2), Rule 5 is replaced by Rule 5'.

Rule 5'. A process sending the token becomes white and changes its state.

If care is taken that at any time at most one token exists (i.e., P_n does not restart the algorithm before the completion of a previous control round), then exactly those messages which cross the diagonal line representing a "control wave" in the time diagram (messages (b), (c), and (d) in Fig. 2) are received in a different state than they were sent. Only messages of type (b) are of interest here, leading to the following replacement of Rule 1.

Rule 1'. A process sending a basic message to a recipient in a different state and with a higher index than its own becomes black.

Obviously, the sender of the message must be informed about the state of the receiver. We will discuss in the next section how this can be achieved. Surprisingly, in CSP this can be done *without explicit messages* and without changing the underlying communication protocol.

Notice that Rule 3' on the one hand and Rules 1', 5', and 7' on the other hand are orthogonal (i.e., an algorithm based on 1', 2, 3 (instead of 3'), 4, 5', 6, 7' is also a valid termination test).

When used as a termination *detector* analogously to [9] (i.e., when the next round is initiated automatically after an unsuccessful test), the original Rule 3 should be used instead of Rule 3' in order to bound the number of control messages. Rule 7' then becomes as follows.

Rule 7''. If P_n receives a black token it starts a new round.

The new *detection algorithm* (based on Rules 1', 2, 3, 4, 5', 6, 7'') usually generates *less messages* and detects termination *faster* than [9] (based on Rules 1–6, 7''), since no extra round after termination is necessary. The original algorithm [9] needs an extra confirmation round whenever some process sent a message to a process with a higher index after the previous round. In particular, a *single round* is sufficient if the algorithm is initiated *after* termination.

In [9] an invariant is established which (when stated verbally) reads "if the token is white then all processes the token has visited in the current round are passive or at least one process the token has not yet visited is black". It is easy to see that our variant of the algorithm satisfies this invariant and is therefore correct.

4. An implementation in CSP

Rule 1' requires that the sender is informed about the state of the receiver. We contend that in virtually all cases this can be done at no extra cost once a connection between two processes has been established by an underlying protocol and the processes have synchronized for communication. Since usually the transmission of a message is acknowledged by a control message (which unblocks the sender), it should be easy to let the one-bit state information "piggyback" on the acknowledgement. Abstract implementations of synchronous communication protocols demonstrate that when processes communicate, some

information about the state of each process must be exchanged by the nontrivial handshaking protocol anyhow in order to reach agreement among the processes in selecting matching communication commands [7,11,13].

We now show how a CSP program $P = [P_1 \| \cdots \| P_n]$ without a termination test can be systematically transformed into another program P' with an incorporated termination test based on the previously stated rules. As usual we make use of an extended version of the original CSP definition [10], where not only input commands are allowed in guards, but also output commands [5]. This extension is powerful since it allows signals to propagate backwards as has been notified by Bougé [6]. As shown in [2,16] we may assume without loss of generality that each process P_i is already transformed into a semantically equivalent *normal form*

P_i:: INIT;
*[$\square_{k \in \Gamma_I} B_k ; P_{j_k}?V_k \rightarrow S_k$
$\square_{k \in \Gamma_O} B_k ; \{P_{j_k}!e_k\} \rightarrow S_k$
]

with a top level repetitive command, where each B_k is an optional boolean expression list and none of the lists INIT and S_k contains an I/O command or repetitive command. The index sets Γ_I and Γ_O are assumed to be disjoint ($\Gamma_I \cap \Gamma_O = \emptyset$). We only consider simple variables and expressions in I/O commands; structured variables and expressions [10] can be handled analogously by tagging the construction identifier with "even" or "odd".

For the purpose of termination detection, those alternative parts $\square B_k ; P_{j_k}?V_k$ with input guards for which $j_k < i$ (a message is received from a process with a smaller index) are split up into two parts:

$\square$ state = 0 ; B_k ; P_{j_k}?even(V_k) $\rightarrow$ S_k,
$\square$ state = 1 ; B_k ; P_{j_k}?odd(V_k) $\rightarrow$ S_k.

The alternative parts $\square B_k ; P_{j_k}!e_k$ with output guards for which $j_k < i$ (a message is sent to a process with a higher index) are split up into

$\square$ B_k ; P_{j_k}!even(e_k) $\rightarrow$
black := black $\vee$ state = 1 ; S_k,
$\square$ B_k ; P_{j_k}!odd(e_k) $\rightarrow$
black := black $\vee$ state = 0 ; S_k.

The expression constructors "even" and "odd" match a corresponding input command only if the receiver is in state 0 or 1, respectively. The sender sets the boolean flag "black" whenever it finds out that it is in a different state, thus complying with Rule 1'.

For the propagating of the token we add two guarded commands at the top-level loop to all processess $P_1, \ldots, P_{n-1}$. They implement Rules 3', 4, and 5' (if $i = 1$, then P_{i-1} denotes P_n).

$\square$ P_{i+1}?token(color) $\rightarrow$ have_token := **true**
$\square$ have_token ;
P_{i-1}!token(color $\vee$ black $\vee$ $\neg$PASSIVE) $\rightarrow$
have_token := **false** ;
black := **false** ;
state := (state + 1) mod 2.

PASSIVE may be seen as a system variable indicating whether the process is active or passive. A possible interpretation of PASSIVE, which is consistent with the specifications in Section 1, consists of the following definition:

PASSIVE = "the process is at its top level loop"
$\wedge \forall k \in \Gamma_O$: $\neg B_k$.

This predicate can easily be implemented. Notice, however, that it does not handle the cases where some output commands are enabled but permanently (or temporarily) blocked because the receiving processes are not ready to accept the messages. For the sake of simplicity we also ignore the so-called Distributed Termination Convention [3] and other problems caused by processes which terminate by leaving the top-level loop of the original program.

For the purpose of termination *detection* the token should only be accepted or propagated if the process is passive (Rule 3 instead of Rule 3'). The guarded commands may then be changed to

$\square$ PASSIVE ; P_{i+1}?token(color) $\rightarrow$ $\cdots$
$\square$ have_token ; PASSIVE ;
P_{i-1}!token (color $\vee$ black) $\rightarrow$ $\cdots$.

It is not possible, however, to combine the commands into one single command

$\square$ PASSIVE ; P_{i+1}?token(color) $\rightarrow$
P_{i-1}!token(color $\vee$ black) ; $\cdots$

since the use of an unconditional output command may result in a deadlock.

P_n has a special role since it initiates the control round and has to announce the result. Because of Rule 1', P_n can never become black. We assume that START and TERMINATED are two flags used in an obvious way by the "environment" or parts of the program not shown here according to rules 2, 6, and 7'.

```
□ START; have_token; PASSIVE;
   P_{n-1}!token(false) →
      START := false;
      have_token := false;
      state := (state + 1) mod 2
□ P_1?token(color) →
      have_token := true;
      TERMINATED := ¬color.
```

When detecting termination, P_n should inform all other processes which may then terminate properly in the sense of [10] by leaving the top level loop. Otherwise P_n may *restart* the algorithm at some later time by setting START to **true**.

Finally, we note the *initializations* of the control variables added to the INIT parts:

```
state := 0;
black := false (P_{i≠n});
have_token := false (P_{i≠n});
have_token := true (P_n);
TERMINATED := false (P_n).
```

The default initial value of START should be **false**.

5. Discussion

The implementation in CSP makes use of a trick due to Bougé [6] which allows to transmit information backwards along unidirectional channels without explicit messages. While this feature is peculiar to CSP (it relies on the semantics of the "alternative send"), it is always the case that in a *synchronous* communication scheme (e.g., Ada, Occam) the sender of a message gets some knowledge about the receiver's state when the message is accepted. The only potential problem is the efficient realization of Rule 1'. In practice, however, it should be possible to "piggyback" the additional one-bit state information on the implicit acknowledge message of the underlying protocol at no cost.

Previous solutions to the distributed termination problem based on synchronous communication which require only one control round after termination have been published by Rana [12] and by Apt and Richier [4]. Rana's solution uses synchronized *real time* clocks. Apt and Richier present a symmetric algorithm with tightly synchronized virtual clocks. In their solution the clocks of the sender and the receiver are synchronized after every basic communication, requiring *two* control messages for each basic message.

Our principle can also be applied to control topologies other than rings (e.g., trees [14], star networks, and parallel or sequential graph traversal schemes) where a particular process can "collect" the colors of the other processes. To detect messages which are sent to already visited processes and to differentiate these messages from messages which, during a previous control round, have been sent by already visited processes to processes which have not yet been visited (messages of type (b) and (d) in Fig. 2), a state modulo 3 is sufficient. Rule 1' is then replaced by the following rule.

Rule 1''. A process in state s sending a basic message to a recipient in state $(s+1)$ mod 3 becomes black.

References

[1] K.R. Apt, Correctness proofs of distributed termination algorithms, *ACM Trans. Programming Languages Systems* **8** (1986) 388–405.

[2] K.R. Apt, L. Bougé and Ph. Clermont, Two normal form theorems for CSP programs., *Inform. Process. Lett.* **26** (1987) 165–171.

[3] K.R. Apt and N. Francez, Modeling the distributed termination convention of CSP, *ACM Trans. Programming Languages Systems* **6** (1984) 370–379.

[4] K.R. Apt and J.-L. Richier, Real time clocks versus virtual clocks, In: M. Broy, ed., *Control Flow and Data Flow: Concepts of Distributed Programming* (Springer, Berlin, 1985) 475–501.

[5] A.J. Bernstein, Output guards and nondeterminism in "communicating sequential processes", *ACM Trans. Programming Languages and Systems* **2** (1980) 234–238.

[6] L. Bouge, On the existence of generic broadcast algorithms in networks of communicating sequential

processes, In: J. van Leeuwen, ed., *Proc. 2nd Internat. Workshop on Distributed Algorithms*, Lecture Notes in Computer Science, Vol. 312 (Springer, Berlin, 1988) 388–407.

[7] G.N. Buckley and A. Silberschatz, An effective implementation for the generalized input–output construct of CSP, *ACM Trans. Programming Languages Systems* **5** (1983) 223–235.

[8] K.M. Chandy and L. Lamport, Distributed snapshots: Determining global states of distributed systems, *ACM Trans. Comput. Systems* **3** (1985) 63–75.

[9] E.W. Dijkstra, W.H.J. Feijen and A.J.M. van Gasteren, Derivation of a termination detection algorithm for distributed computations, *Inform. Process. Lett.* **16** (1983) 217–219.

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

[11] R.B. Kieburtz and A. Silberschatz, Comments on "Communicating sequential processes", *ACM Trans. Programming Languages Systems* **1** (1979) 218–225.

[12] S.P. Rana, A distributed solution of the distributed termination problem, *Inform. Process. Lett.* **17** (1983) 43–46.

[13] A. Silberschatz, Communication and synchronization in distributed systems, *IEEE Trans. Software Eng.* **SE-5** (1979) 542–546.

[14] R.W. Topor, Termination detection for distributed computations, *Inform. Process. Lett.* **18** (1984) 33–36.

[15] J.P. Verjus, On the proof of a distributed algorithm, *Inform. Process. Lett.* **25** (1987) 145–147.

[16] D. Zöbel, Normalform-Transformationen für CSP-Programme, *Informatik — Forschung und Entwicklung* **3** (1988) 64–76.