REAL TIME CLOCKS
VERSUS
VIRTUAL CLOCKS

Krzysztof R. Apt
L.I.T.P, Université Paris 7
2, Place Jussieu
75251 Paris, France

Jean-Luc Richier
IMAG, LGI,
B.P. 68
St Martin d'Hères Cedex, France

Abstract Symmetric distributed termination algorithms are systematically developed. Solutions are first presented in an abstract setting of Dijkstra, Feijen and Van Gasteren [DFG] and then gradually transformed into solutions to the distributed termination problem of Francez [F]. The initially used global real time clock is eventually replaced by local virtual clocks. A dependence between the degree of clock synchronization and the efficiency of the solutions is indicated.

1. ABSTRACT SOLUTIONS

Following Dijkstra and Scholten [DS] and Dijkstra, Feijen and Van Gasteren [DFG] and Topor [T] we consider a fixed finite set of machines which communicate by exchanging messages. Communication is considered instantaneous. At each moment machines are either active or passive. Only active machines can send messages. The receipt of a message by a machine is the only mechanism that triggers its transition passive → active. In contrast, the transition active → passive may occur within each machine "spontaneously". If there is an active machine then either a machine will turn passive or a message will eventually be sent. But when all machines are passive no communication is any more possible and the activity ceases.

Similarly as in the above papers we are interested in designing an algorithm allowing to detect termination of the computation, i.e. to detect the situation when all machines are passive. As in [DFG] we assume that the machines can be arranged in a ring along which detection messages will be propagated. Unlike in [DFG] we develop here *symmetric* termination detection algorithms.

Suppose that all machines have access to a *global clock*. Whenever a machine turns passive it notes down the current time instant t and sends to its right-hand side neighbour a *detection message* with the time stamp t. The aim of this message is to verify whether other machines were also passive at the time instant t. If this is the case then termination is detected : at the time instant t all machines were passive. By assumption detection messages do not affect machine state. Detection messages are sent clockwise. When machine *a* receives from its left-hand side neighbour a detection message created by machine *b* two things may happen.

NATO ASI Series, Vol. F14
Control Flow and Data Flow: Concepts of
Distributed Programming. Edited by M. Broy
© Springer-Verlag Berlin Heidelberg 1985

If upon its reception it is active then this attempt of machine *b* to detect termination fails. The detection message is thus annihilated.

But if upon reception of the detection message machine *a* is passive then there is a possibility that the inquiry is successful. To this purpose it suffices to consult the time instant t_o at which machine *a* turned for the last time passive. Machine *a* was passive since the time instant t_o until the moment t_1 of reception of the detection message. We have $t \leqslant t_1$. Thus if $t_o \leqslant t$ then machine *a* was passive at the time instant t and the detection message is forwarded to the right-hand side neighbour.

If $t_o > t$ then machine *a* was active at a time instant t_2 such that $t < t_2 \leqslant t_o$. Thus some machine (*not necessarily* machine *a*) was not passive at the time instant t. Indeed, otherwise no machine could have been active at a later time instant. Thus this attempt of machine *b* to detect termination fails and the detection message is annihilated.

Summarizing, the above algorithm consists of the following two rules.

Rule 1

Whenever a machine turns passive it notes down the current time instant t and sends to its right-hand neighbour a detection message with the time stamp t.

Rule 2

When a detection message with the time stamp t is received by a machine, the message is

- destroyed if
    i) it was created by the machine, or
    ii) the machine is active, or
    iii) the machine is passive but the time instant when it turned passive for the last time is larger than t,

- forwarded to the right-hand side neighbour otherwise.

Actions prescribed by each rule are considered atomic, i.e. they cannot be interrupted.

This algorithm satisfies the following three properties.

Property 1

When a machine receives back its detection message then termination is detected.

Property 2

When the computation proper terminates its termination will be eventually detected by one of the machines.

Property 3

If some of the machine is active then eventually either a machine will turn passive or a message belonging to the computation proper will be sent.

Proof of property 1

By construction if a detection message with the time stamp t makes a full cycle then every process was passive at the time instant t.

Proof of property 2

Suppose that the computation proper terminates. Consider a machine which became passive as the last one, say at the time instant t. The detection message of this machine with the time stamp t will make a full cycle. By property 1 termination will thus be eventually detected.

Proof of property 3

Suppose that at least one of the machines remains active but none of them performs the computation proper. By construction only finitely many detection messages can be created - one per passive machine. All of them will be eventually destroyed. By assumption the next action performed will belong to the computation proper.

The last property states that the superimposed detection scheme does not delay indefinitely the computation proper.

To implement the above algorithm each machine should be able to recognize its own detection messages. One possibility to achieve this consists of making available to each of the machines the total number of machines in the system. Then to each detection message a field can be attached indicating the total number of machines which have received it. If this number equals the number of the machines then the detection message returned to the machine that created it. Another possibility consists of "signing" each message with a name of the machine that created it provided all names are different and each machine knows its name.

An obvious defficiency of the algorithm is the relatively large size of the atomic actions which limits the possible parallelism. We shall now present an alternative solution obtained by splitting the first rule into two

Rule 1 a

Whenever a machine turns passive it notes down the current time instant t.

Rule 1 b

When a machine is passive and noted down the time instant t at which it turned passive for the last time it sends a detection message with the time stamp t.

In contrast, rule 2 is adopted without changes.

This algorithm allows for more computations than the previous one and yet its correctness proof is exactly the same as that of the previous one.

There are several ways the above algorithm can be improved. First of all we might try to replace the global real time clock by the local ones. Next, we might try to replace real time clocks by the virtual ones.

Independently on the clock considerations we might also reconsider rule 2 which could be also split into subrules. Also in the present setting detection messages cannot overtake each other and are destroyed when received at an inappropriate moment. These conditions might be appropriately relaxed.

It is possible to present the consecutive versions of the algorithm in the current setting of the abstract machines. However, lack of a rigorous programming notation impedes a compact and unambigous presentation. Therefore we shall present all the further versions of the algorithm in a concrete programming language. We choose here a variant of CSP of Hoare [H] in which output guards are allowed.

2. FROM ABSTRACT TO CONCRETE SOLUTIONS

We first model the behaviour of the machines. Each machine will be represented by a process. Processes have distinct names $P_1,\ldots,P_n$ and are connected by unidirectional communication channels. For $i = 1,\ldots,n$ Γ_i stands for the set of indices of the processes with which process P_i is connected by a channel outgoing from P_i and Δ_i stands for the set of indices of the processes with which P_i is connected by a channel ingoing into P_i. By definition for $i,j = 1,\ldots,n$ $j \in \Delta_i$ iff $i \in \Gamma_j$.

Behaviour of each process P_i is represented by the following program text reflecting the previous assumptions about communication and status of the components of the system :

```
$P_i$ :: * [ $\square_{j\in\Gamma_i}$ active$_i$ ; $P_j$ ! $t_j$ → skip
             $\square_{j\in\Delta_i}$ $P_j$ ? $x_j$ → active$_i$:=true
             □ active$_i$ → active$_i$:=false
           ].
```

Expression t_j represents a message sent by process P_i to process $_j$. This message is assigned to variable x_i of P_j. To make the processes isjoint we should actually rename the variables appropriately. The status of ach process is described by its Boolean variable active.

This representation of machines behaviour can be a starting point to a ranslation of abstract distributed termination algorithms into solutions ritten in CSP. However, this representation is not satisfactory as it models n extremely restrictive situation in which processes either communicate or hange their status. No useful work is done and the distributed computation onsists of a useless exchange of messages.

A more realistic representation takes into account a possible work arried out by each process. Sending or receiving of a message will now be ollowed by a local computation.

In CSP communication is symmetric in the sense of synchronization etween input and output commands. Therefore it is more convenient to consider idirectional channels. Moreover, we assume that two processes can be onnected by more than one channel. For $i = 1,\dots,n$ Γ_i will now be a ultiset consisting of the indices of the processes with which P_i is onnected.

Summarizing, processes will now have the following bodies :

$$S_i \equiv INIT_i \; ; \; * [\underset{j\in\Gamma_i}{\Box} g_{i,j} \to S_{i,j}]$$

here

i) each guard $g_{i,j}$ contains an I/O guard adressing process P_j,
ii) none of the statements $INIT_i$, $S_{i,j}$ contains an I/O command.

We still have to make precise the rules for communications between rocesses. With each process we associate a Boolean expression B_i, called a *tability condition*. B_i involves variables of P_i and possibly some other uxiliary variables. A process will be called *active* if its control is at the ain loop entry and $\neg B_i$ holds and will be called *passive* if its control is t the main loop entry and B_i holds. Thus a process has a status only when ts control is at the main loop entry.

Previous rules for communication related directly the status of each achine to its involvement in communication. This point of view has now to be odified once we introduced a possibility of carrying out a local computation fter being engaged in a communication.We adopt the following two assumptions:

a) no communication can take place between a pair of passive processes,
b) if there is an active process some communication will take place.

Assumption a) reflects the symmetric role of sending and receiving - eing active is not any more a necessary condition to be able to send. ssumption b) can be stated differently as : whenever deadlock takes place, ll processes are passive.

The problem consists of detecting in the program $P \equiv [P_1::S_1 \| \cdots \| P_n::S_n]$ the situation when all processes are passive. But once we deal with processes which have a concrete syntax and not with abstract machines, we can strengthen the requirements.

The problem will be now phrased as follows : transform P into another program P' which eventually properly terminates whenever all processes become passive. Obviously, detection by a process of the fact that all processes are passive will be a crucial step of a solution.

In such a way we arrived to a formulation of the problem within CSP which becomes exactly the *distributed termination problem* of Francez [F].

To describe more precisely the desired properties a solution to the distributed termination problem should satisfy, we should take into account a relation between the computations of P and P'. In general, P' will use some additional variables and allow for some additional communications of new types. P' will be of the form $[P_1 :: S'_1 \| \ldots \| P_n :: S'_n]$. Every computation of P' can be naturally restricted to a computation of P by disregarding all new variables and statements refering to them and all new communications. We shall then say that in a computation of P' a process becomes passive if it becomes passive in the restriction of the computation to a computation of P.

We should also define when a property involving variables belonging to different processes holds in a distributed computation. This would require an introduction of a formal semantics for CSP programs, a subject we prefer not to discuss here. Informally, a global property B holds in a distributed computation if it holds in its possible global state.

A solution P' to the distributed termination problem should satisfy the following four properties.

Property 1

Whenever P' properly terminates then all processes are passive.

Property 2

There is no deadlock in P'.

Property 3

If all processes become passive then eventually P' will properly terminate.

Property 4

If not all processes are passive then eventually a statement from the original program will be executed.

Properties 1,3,4 correspond to properties 1,2,3 from the previous section, respectively. Property 2 is new and makes no sense when considered in the framework of abstract machines - deadlock coincides there with termination as there is no distinction between final and non-final states of the machines.

As in section 2 we are interested here in symmetric solutions to the problem. Therefore we assume that the processes can be arranged in a ring. Neighbours of process P_i are processes P_{i-1} and P_{i+1} where counting is done within $\{1,...,n\}$ clockwise. By assumption for each $i = 1,...,n$ $i-1\in\Gamma_i$, i.e. the neighbours in the ring are connected by a communication channel.

The notion of a symmetry in the context of concurrent programs is a subtle one. One can easily construct distributed programs which look symmetric yet some of their components are favoured with respect to the others. For a discussion of this issue reader is referred to Lehmann and Rabin [LR] and for extensive treatment in the framework of CSP programs to Bougé [B1]

The solutions we discuss are symmetric in a purely syntactic sense - texts of the transformations are identical for all components (for a more interesting semantic definition of symmetry for CSP programs see [B1]). Moreover, none of the processes knows its identity number (this excludes "tricks" of the form P_i :: ... [i = i_o → ...] ... for a constant i_o, which allows one to distinguish process P_{i_o} from the others). On the other hand the total number of the processes in the system is available to each of the processes.

3. A SOLUTION WITH A GLOBAL REAL TIME CLOCK

We now obtain a symmetric solution to the distributed termination problem by translating into CSP one of the algorithms of section 2. To this purpose we additionally assume existence of a global real time clock which is absent in the original CSP. Within process P_i its value can be read by executing the assignment T:= CLOCK-TIME which results in assigning to variable T of P_i the current clock time. This assignment is used to record the time when B_i holds.

A natural translation of rule 1 into CSP consists of adding to the main loop within process P_i an additional branch

$\square$ B_i → T:=CLOCK-TIME ;
       P_{i+1} ! detection-message (TIME)

Unfortunately such a modification of the program leads to difficulties. Namely consider a situation when all processes become passive and subsequently all choose the newly added branch. Then a deadlock arises which violates property 2.

Thus rule 1 cannot be adopted literally. A solution consists of adopting rules 1a and 1b instead.

Rule 1a can be translated as the branch

```
□ $B_i$ ; $\neg$ OK → T:=CLOCK-TIME ; OK:=true.
```

OK is a new boolean variable whose role is to ensure that reading of the clock value takes place at most once during a period during which the process is passive. Thus initially OK should be set to false and also reset to false after every original communication.

Before we translate rule 1b we fix the form of the detection messages. They will have two parameters : the time value T and a field COUNT indicating the total number of processes which have received it. Thus initially COUNT will equal 1.

Now rule 1b can be translated as the branch

```
□ OK ; $\neg$ SENT ; $P_{i+1}$!detection-message (T,1) → SENT:=true.
```

SENT is a new boolean variable whose role is to ensure that sending of a detection message takes place at most once during a period when the process is passive. Thus initially SENT should be set to false and also reset to false after every original communication.

Consider now rule 2. Its formulation presupposes that the step prescribed by rule 1a is carried out if upon reception of a detection message the process is passive. This will be formalized by the use of the guard B_i → OK in front of the reception of a detection message. Rule 2 will now be translated as the branch

```
□ $B_i$ → OK ; $P_{i-1}$ ? detection-message (TIME,COUNT) →
  [ COUNT = n → skip -- destroy the message
   □ COUNT < n →
     [ $\neg$ $B_i$ → skip -- destroy the message
      □ $B_i$ →
          [TIME < T → skip -- destroy the message
           □ TIME $\geq$ T → COUNT:=COUNT + 1 ;
                        $P_{i+1}$! detection-message (TIME,COUNT)
          ]
     ]
  ]
```

This concludes the translation of the algorithm. The transformed version of process P_i thus has the following form :

```
$P_i$:: INIT$_i$ ; OK:=false ; SENT:=false ;
      * [ □  $g_{i,j}$ → OK:=false ; SENT:=false ; $S_{i,j}$
        $j \in \Gamma_i$
         □  $B_i$ ; ¬ OK → T:=CLOCK-TIME ; OK:=true
         □  OK ; ¬ SENT ; $P_{i+1}$ ! detection-message (T,1) → SENT:=true
         □  $B_i$ → OK ; $P_{i-1}$ ? detection-message (TIME, COUNT) →
            [COUNT = n → skip
             □  COUNT < n →
                [ ¬ $B_i$ → skip
                  □ $B_i$ →
                    [ TIME < T → skip
                      □ TIME ≥ T → COUNT:=COUNT + 1 ;
                                  $P_{i+1}$ ! detection-message (TIME, COUNT)
                    ]
                ]
            ]
        ]
```

Here and elsewhere all variables are assumed to be local to P_i. When reasoning about the programs we shall use subscripts to distinguish between the variables of different processes.

The presented solution is clearly not the desired one-the obtained program can never properly terminate,so property 1 is not satisfied. In fact, once all processes become passive the program will eventually deadlock.It seems that we made no progress with respect to the original problem as we now have to transform the last program into another one which satisfies property 1.

But this step is simpler than the original task. We build into the last program a possibility of initiating by a process a wave of termination messages which will cause proper termination of all processes. This wave will be initiated by a process which receives back its own detection message. In general several termination waves can be propagated at the same time in the ring. Moreover, we have to cope with a possible presence in the ring of both detection and termination messages.

To solve these problems we introduce three new boolean variables FAIT, RECU and EMIS (French for done, received and emitted, respectively) all initialized to false. FAIT will be set to true once a detection message with COUNT = n is received. Setting FAIT to true will enable a process to exit the main loop and send a termination message. Within the second loop RECU will be set to true once a termination message is received and EMIS will be set to true once a termination message has been sent. A process will terminate when it will *both* send and receive a termination message (not necessarily the same one). Thus the program will terminate when each process will send a termination message.

Summarizing, this revised version of P_i is as follows :

```
  FAIT:=false ; RECU:=false ; EMIS:=false ;
* [ □ ¬FAIT ; $g_{i,j}$ → OK:=false ; SENT:=false ; $S_{i,j}$
   $j \in \Gamma_i$
    □ ¬FAIT ; $B_i$ ; ¬OK → T:=CLOCK-TIME ; OK:=true
    □ ¬FAIT : OK ; ¬SENT ; $P_{i+1}$! detection-message (T,1) →
                                  SENT:=true
    □ ¬FAIT ; $B_i$ → OK ; $P_{i-1}$ ? detection-message (TIME,COUNT) →
       [ COUNT = n → FAIT:=true
         □ COUNT < n →
            [¬$B_i$ → skip -- purge the message
             □ $B_i$ →
               [ TIME < T → skip -- purge the message
                 □  TIME ≥ T → COUNT:=COUNT + 1 ;
                               $P_{i+1}$! detection-message (TIME,COUNT)
               ]
            ]
       ]
    □ ¬FAIT ; $P_{i-1}$ ? termination-message → FAIT:=true ;
                                              RECU:=true
   ] ;
* [ ¬RECU v ¬EMIS ; $P_{i-1}$ ? detection-message (TIME,COUNT) →
                                   skip -- purge the message
    □ ¬RECU ; $P_{i-1}$ ? termination-message → RECU:=true
    □ ¬EMIS ; $P_{i+1}$ ! termination-message → EMIS:=true
  ]
```

A possible presence in the ring of both detection and termination messages is reflected by the added last guard of the first loop and the first guard of the second loop.

The resulting program is a correct solution to the distributed termination problem. We now wish to reduce in this solution the number of detection messages sent by the processes. To this purpose we shall ensure that whenever process P_i transmitted to P_{i+1} a detection message it had received from P_{i-1} then it will not send to P_{i+1} any detection message of its own (unless an original communication with P_i occurs). This can be simply achieved by setting SENT to <u>true</u> after the I/O command P_{i+1}!detection message (TIME, COUNT). Now it can be checked (and we shall prove it rigorously in the next section) that whenever P_i receives back its own detection message it will not receive any further detection messages. This allows us to restructure the program and bring it back to the form of one loop.

This final version of P_i has the following form :

```
$P_i$:: $INIT_i$ ; OK:=false ; SENT:=false ;
       FAIT:=false ; EMIS:=false ;
     * [ □ ¬FAIT ; $g_{i,j}$ → OK:=false ; SENT:=false ; $S_{i,j}$
        $j \in \Gamma_i$
         □ ¬FAIT ; $B_i$ ; ¬OK → T:=CLOCK-TIME ; OK:=true
         □ ¬FAIT ; OK ; ¬SENT ; $P_{i+1}$ ! detection-message (T,1) →
                                 SENT:=true
         □ ¬FAIT ; $B_i$ → OK ; $P_{i-1}$ ? detection-message (TIME, COUNT) →
           [ COUNT = n → FAIT:=true
             □ COUNT < n →
               [ ¬$B_i$ → skip -- purge the message
                 □ $B_i$ →
                   [TIME < T → skip -- purge the message
                    □ TIME ⩾ T → COUNT:=COUNT+1 ;
                                 $P_{i+1}$ ! detection message (TIME, COUNT) ;
                                 SENT:=true
                   ]
               ]
           ]

        □ ¬RECU ; $P_{i-1}$ ? termination-message → RECU:=true ;
                                                   FAIT  :=true
        □ FAIT ; ¬EMIS ; $P_{i+1}$! termination-message → EMIS:=true
       ]
```

Correctness of the above program, even though it has been systematically derived from the abstract solution presented in section 1, is by no means obvious. In the last phase we transformed the program into another one which is deadlock free. The provided reasoning was completely informal and it requires justification.

4. A CORRECTNESS PROOF

We now provide a correctness proof of the program presented in the previous section. More precisely, we claim that the assumption from the introduction implies assertions 1 and 2 from section 2.

For the sake of the proof we consider a failure (an attempt to communicate only with terminated processes) as a special case of a deadlock. In a deadlock situation no process can proceed and at least one of them did not terminate. When considering messages we shall often informally refer to their creation, sending, reception and an attempt to send them.

The following notion will be helpful in the sequel.

Definition 1 In a computation of P' a Boolean expression is *monotone* if, once it holds, it continues to hold.

The proof consists of a sequence of lemma's, all about the final program given in the previous section.

Lemma 1

i) The Boolean expression $\bigwedge_{i=1}^{n} B_i$ is monotone.

ii) The Boolean variables FAIT, RECU and EMIS are monotone.

Proof i) The condition $\bigwedge_{i=1}^{n} B_i$ is monotone for the original program. The parts added to P_i do not invalidate the condition B_i.

ii) All three variables are never reset to false. □

Lemma 2 If a process P_k received a detection message (TIME,n) then $\bigwedge_{i=1}^{n} B_i$ holds since the time instant TIME (i.e. each B_i holds since the time instant TIME).

Proof This detection message has made a full cycle so necessarily $T_j \leqslant$ TIME for all j. Consider the time instant t_j when P_j received this detection message. We have $t_j \geqslant$ TIME. B_j held at the time instant t_j, so it held throughout the time interval $[T_j,t_j]$. In particular B_j held at the time instant TIME. Thus for all j B_j held at the time instant TIME. The claim now follows by lemma 1. □

Proof of property 1

Consider a properly terminating computation of P'. A termination message must have been created by some process P_i. Thus P_i must have received a detection message (TIME,n). The claim now follows from lemma's 1 and 2. □

Lemma 3 If a process P_k received a detection message (TIME,n) then P_{k-1} will not attempt to send to P_k another detection message.

Proof Suppose otherwise. Call the first detection message m_1 and the second m_2. m_2 must have been created by some P_l, *after* the time instant t' at which P_l sent m_1 to P_{l+1}. $SENT_l$ held at t'. But $TIME \leq t'$ so by lemma 2 $\bigwedge\limits_{i=1}^{n} B_i$ holds since t'. So by the assumption from section 1 also $SENT_l$ holds since t'. Thus m_2 could not have been created by P_l. Contradiction.□

Lemma 4 No deadlock is possible with some process P_i being in front of the I/O command P_{i+1}!detection message (TIME,COUNT).

Proof Suppose otherwise. Observe first that it cannot be the case that all P_i's are blocked in front of P_{i+1} ! detection-message (TIME, COUNT). Indeed, till the time of deadlock only finitely many detection messages have been sent. Thus there must exist a process which did not receive any detection message after sending for the last time a detection message. This process cannot find itself in front of the above I/O command α.

Thus if some P_i is blocked in front of α then some other P_k is not. Consider then the first P_k counting from P_i clockwise which is not blocked in front of α. Then P_k is either at the outer level or has terminated.

If P_k is at the outer level then, since a deadlock takes place and P_{k-1} is in front of α, $FAIT_k$ holds. Thus, once again because of the deadlock, either $EMIS_k$ or $RECU_{k+1}$ holds. This means that a termination message has been sent from P_k to P_{k+1}. If P_k has terminated then a termination message must have been sent from P_k to P_{k+1}, as well.

Now, since P_{k-1} is in front of α, $\neg FAIT_{k-1}$ holds. Thus this termination message could not have been transmitted to P_k by P_{k-1}. So it has been created by P_k itself. In other words P_k must have received a detection message with COUNT = n. We get now a contradiction with lemma 3. □

Proof of property 2

Suppose a deadlock is possible. Consider a deadlock situation. By the previous lemma for all i P_i either has terminated or is at the outer level.

Case I For some i $FAIT_i$ holds.

Then $EMIS_i \vee RECU_{i+1}$ holds. Thus a termination message has been sent from P_i to P_{i+1}. Consider the first process P_l clockwise from P_i which did not receive a termination message (i.e. whose $RECU_j$ is false). If it does not exist then for all j $RECU_j \wedge FAIT_j$. But if $RECU_j$ then $EMIS_{j-1}$, so also for all j $EMIS_j$. Thus all processes have terminated and there is no deadlock. We thus have for some l $\neg RECU_l$ and $FAIT_{l-1} \wedge \neg EMIS_{l-1}$. But this means that a termination message can be sent from P_{l-1} to P_l so there is no deadlock.

Case II For all i $\neg FAIT_i$ holds.

Then all processes are at the outer level. No basic communication is possible, so for all i B_i holds. Thus also for all i OK_i holds and consequently for all i $SENT_i$ holds.

This means that every P_i has sent a detection message with its current value of $BTIME_i$ and all of them have been eventually purged. Consider a largest $BTIME_j$. The detection message with $BTIME_j$ has been purged by some P_k. This means that $\neg B_k$ held at some time instant $t \geqslant BTIME_j$. Thus also $\neg OK_j$ held at t so it must be the case that $BTIME_k > t$. Contradiction.

This concludes the proof.□

Lemma 5 There does not exist an infinite computation with finitely many original communications and infinitely many other communications.

Proof Suppose otherwise. Then by lemma 1 ii) some P_i has created infinitely many detection messages so the variable $SENT_i$ has changed its value infinitely often. But this is only possible if the basic communications took place infinitely often. Contradiction.□

We can now prove assertions 1 and 2 from section 2.

Proof of Property 3

Consider a computation of P'in which at a certain moment all processes are passive. By lemma 1 i) $\bigwedge_{i=1}^{n} B_i$ holds from this moment on so no original communication will take place any more. Due to property 2 this computation cannot end in a deadlock. By lemma 5 this computation cannot be infinite. So it has to terminate properly.□

Proof of property 4

By lemma 5 and the form of the program.□

Comparing the above correctness proof with the one presented in the first section we observe that the main burden lies here in proving property 2, i.e. deadlock freedom. In contrast, proofs of properties 1,3 and 4 are nothing else but rewordings of the corresponding proofs given in section 1.

5. A SOLUTION WITH LOCAL REAL TIME CLOCKS

An existence of a global real time clock in a distributed system is clearly not a realistic assumption. We now modify the solution presented in section 3 and replace the global real time clock by local real time clocks. The local clocks have to be synchronized properly. The solution we propose consists simply of integrating the standard clock synchronization procedure of Lamport [L] into the previously given program. Thus we advance the value CLOCK-TIME$_i$ of the local clock of P_i to max(CLOCK-TIME$_i$,TIME) upon reception by P_i of a detection message with the time stamp TIME. The program has now the following form :

```
$P_i$ :: $INIT_i$ ; OK:=false ; SENT:=false ;
      FAIT:=false ; RECU:=false ; EMIS:=false ;
    * [ □ ¬ FAIT ; $g_{i,j}$ → OK:=false ; SENT:=false ; $S_{i,j}$
       $j \in \Gamma_i$
      □ ¬ FAIT ; $B_i$ ; ¬ OK → T:=CLOCK-TIME$_i$ ; OK:=true
      □ ¬ FAIT ; OK ; ¬ SENT ; $P_{i+1}$ ! detection-message (T,1) →
                                  SENT:=true
      □ ¬ FAIT ; $B_i$ → OK ; $P_{i-1}$ ? detection-message (TIME,COUNT) →
        [COUNT = 2n → FAIT:=true
         □ COUNT < 2n → CLOCK-TIME$_i$:=max(TIME,CLOCK-TIME$_i$) ;
           [ ¬ $B_i$ → skip -- purge the message
             □ $B_i$ →
               [ TIME < T → skip -- purge the message
                 □ TIME ≥ T → COUNT:=COUNT+1 ;
                              $P_{i+1}$ ! detection-message (TIME,COUNT) ;
                              SENT:=true
               ]
             ]
         ]
       □ ¬ RECU ; $P_{i-1}$ ? termination-message → RECU:=true ;
                                                 FAIT:=true
       □ FAIT ; ¬ EMIS ; $P_{i+1}$ ! termination-message → EMIS:=true
    ]
```

There is one striking difference between this program and the previous one - upon receiving a detection message its COUNT value is compared in the above program with 2n instead of n. In other words termination is detected in the above program only when a detection message has successfully made two full cycles.

It is useful to see why one cycle is not sufficient here to detect termination. To this purpose consider a hypothetical execution of the program with $n \geq 3$. A detection message sent by P_1 can make successfully a full cycle even if not all processes become passive. Indeed, a communication can take place between P_i and P_j for $1 < i+1 < j \leq n$ at the moment when the detection message is "between" then and activate process P_i. This fact will not be discovered when comparing the time stamp of the message with the value T_j of P_j if the clock of P_j advances slowly. Observe that T_j refers to the reading of the clock *before* the synchronization.

Informally, the first cycle is used to synchronize the clocks whereupon the second cycle is needed to check that no process was active since then. A formal proof of the correctness of the above solution is a combination of the proof given in the last section and the proof of the next solution and is omitted.

6. A SOLUTION WITH VIRTUAL CLOCKS

Why where the clocks needed in the presented solutions ? Initially, the idea of the algorithm was to verify that all processes were passive at a time instant t. With the introduction of the local clocks this idea becomes less transparent. A close inspection of the last solution reveals that the clocks were needed in order to be able to find that a process was active after a given time instant. To this purpose it is needed that two successive readings of a clock give different, increasing values, i.e. that each clock advances. But this effect can be already achieved using virtual clocks. This brings us to another solution to the distributed termination problem.

In each process we replace a local real time clock by an integer variable T representing a virtual clock. T is initialized to zero. It is incremented by one at the places where before the clock was consulted, i.e. where the assignment $T{:=}\text{CLOCK-TIME}_i$ took place and is synchronized with other clocks in the same way as before.

Summarizing, this solution has the following form :

```
$P_i$ ::  OK:=false ; SENT:=false ; FAIT:=false ;
          RECU:=false; EMIS:=FALSE ; T:=0 ;
        * [  □ ¬ FAIT;$g_{i,j}$ → OK:=false ; SENT:=false ; $S_{i,j}$
           $j \in \Gamma_i$
             □ ¬ FAIT ; $B_i$ ; ¬OK → T:=T+1 ; OK:=true
             □ ¬ FAIT ; OK ; ¬SENT ; $P_{i+1}$! detection message (T,1)
                                       → SENT:=true
             □ ¬ FAIT ; $B_i$ → OK ; $P_{i-1}$ ? detection-message (TIME, COUNT) →
               [COUNT=2n → FAIT:=true
                □ COUNT < 2n →
                  [¬ $B_i$ → T:=max(TIME,T) -- purge the message
                   □ $B_i$ →
                     [TIME < T → skip -- purge the message
                      □ TIME ≥ T → T:=TIME ;
                                   COUNT:=COUNT+1 ;
                                   $P_{i+1}$! detection-message(TIME,COUNT) ;
                                   SENT:=true
                     ]
                  ]
               ]
             □ ¬ RECU ; $P_{i-1}$ ? termination-message → RECU:=true ;
                                                         FAIT:=true
             □ FAIT ; ¬ EMIS ; $P_{i+1}$! termination-message → EMIS:=true
          ]
```

T represents now both CLOCK-TIME$_i$ and T from the previous program. Therefore it is synchronized somewhat later than before.

7. A CORRECTNESS PROOF

The correctness proof of the solution with the virtual clocks is very similar to the proof of the solution given at the end of section 3. We prove a sequence of similar lemma's, this time about the program from the previous section.

Lemma 6

i) The Boolean expression $\bigwedge_{i=1}^{n} B_i$ is monotone.

ii) The Boolean variables FAIT, RECU and EMIS are monotone. □

Lemma 7 Suppose that a process P_k received a detection message (TIME,COUNT) with COUNT=2n. Let t be the time instant at which P_k received this message with COUNT = n. Then $\bigwedge_{i=1}^{n} B_i$ holds since t.

Proof This detection message has made two full cycles. Consider the time instants t'_j and t''_j at which P_j sent this detection message to P_{j+1} during the first and second cycle, respectively. We have both at the time instant t'_j and t''_j T_j=TIME. Since the value of TIME did not change, this means that the value of T_j did not change during the time interval $[t'_j,t''_j]$. But B_j and OK_j held both at t'_j and t''_j, so this implies that both B_j and OK_j held throughout the time interval $[t'_j,t''_j]$. In particular B_j held at the time instant $t \in [t'_j,t''_j]$. The claim now follows by lemma 6. □

Lemma 8 If a process P_k received a detection message (TIME, 2n) then P_{k-1} will not attempt to send to P_k another detection message.

Proof Suppose otherwise. Call the first detection message m_1 and the second m_2. m_2 must have been created by some P_l after the time instant t'_l at which P_l sent m_1 to P_{l+1} during its first cycle. $SENT_l$ held at t'_l. But by the proof of lemma 7 OK_l holds since t'_l. So also $SENT_l$ holds since t'_l. Thus m_2 could not have been created by P_l. Contradiction. □

Proof of property 2

The proof is identical to the proof given in section 4. The only difference is in the treatment of the case II.

So suppose that all processes are blocked at the outer level and that for all i $\neg FAIT_i$ holds. As before this means that every P_i has sent a detection message and all of them have been eventually purged. Moreover, at the moment of deadlock for all i OK_i holds. Consider a largest T_j. Process P_j has sent a detection message with the time stamp T_j. Since it was purged by some P_l, it must have been the case that $\neg B_l$ held upon its reception by P_l. Thus T_l was set to T_j. Since $\neg B_L$ implies $\neg OK_l$ and at the moment of deadlock OK_l holds, it must be the case that T_l was incremented. Thus at the moment of deadlock T_l is larger than T_j. Contradiction. □

The proofs of properties 1, 3 and 4 are the same as those given in section 4 with the only difference that a creation of a termination message by a process now means that it has received a detection message (TIME, 2n).

8. A "STORE AND FORWARD" SOLUTION

In the solution presented in section 6 detection messages cannot overtake each other. In other words, detection messages are transmitted by the processes in the order of their arrival. This can lead to an unnecessarily orderly traffic of the detection messages in the system. We now present a modified solution in which detection messages can overtake and destroy the detection messages which precede them. This is achieved by allowing the processes to store the detection messages before forwarding them.

The solution is obtained by transforming the program from section 6 appropriately. We simply move the inner I/O command P_{i+1}! detection-message (TIME, COUNT) to the guard position in the outer loop. To this purpose another Boolean variable called FORWARD is introduced. FORWARD is set to true when a process received a detection message which is supposed to be transmitted further. Once this or another detection message received later is forwarded, FORWARD is reset to false. Also, as soon as an original communication takes place, FORWARD is set to false which means that any pending message is then purged.

This solution has the following form :

```
$P_i$::  OK:=false ; SENT:=false ; FAIT:=false ; RECU:=false ;
       EMIS:=false ; FORWARD:=false ; T:=0 ;
     *[□ ¬ FAIT;$g_{i,j}$ → OK:=false;SENT:=false;FORWARD;=false;$S_{i,j}$
      $j \in \Gamma_i$
       □ ¬ FAIT ; $B_i$ ; ¬OK → T:=T+1 ; OK:=true
       □ ¬ FAIT ; OK ; ¬SENT ; $P_{i+1}$! detection-message (T,1)
                           → SENT:=true
       □ ¬ FAIT ; $B_i$ → OK ; $P_{i-1}$ ? detection-message (TIME, COUNT) →
          [COUNT = 2n → FAIT:=true
           □ COUNT < 2n →
             [¬ $B_i$ → T:=max(TIME,T) — purge the message
              □ $B_i$ →
                TIME < T → skip — purge the message
                 □ TIME ≥ T → T:=TIME ;
                              COUNT:=COUNT+1 ;
                              FORWARD:=true ;
                              SENT:=true
                ]
             ]
          ]
        □ ¬ FAIT ; FORWARD ; $P_{i+1}$! detection-message (TIME, COUNT)
                         → FORWARD:=false
        □ ¬ RECU ; $P_{i-1}$ ? termination-message → RECU:=true ;
                                                 FAIT:=true
        □ FAIT ; ¬ EMIS ; $P_{i+1}$! termination-message → EMIS:=true
       ]
```

Note that whenever P_i receives a detection message which should be forwarded (i.e. such that FORWARD is set to true), then SENT is set to true, as well. This effectively blocks P_i from sending at this moment its own detection message - P_i will be able to send further only a detection message it received from P_{i-1}. It will be always the last one received - others will be destroyed. This sending will take place only if in the meantime no basic communication happened.

Correctness of this solution can be proved in a similar way as that of section 6. In fact, the proof of property 2 is now simpler because there are fewer situations in which deadlock could arise. We leave the details of the proof of the reader.

Finally, it is useful to observe that an analogous "store and forward" version can be obtained for the solutions from sections 3 and 5.

9. A SOLUTION WITH PENDING MESSAGES

In the solutions so far presented detection messages are purged when they are received by a process which is not passive. This can be a source of inefficiency since a purged detection message could have already been received by a large number of processes. A possible improvement is to allow a process to keep a received detection message until it state becomes passive. We now present a solution in which this idea is realized but admittedly in a partial way.

The solution consists of the following modification of the program from section 8. The only criterion whether a detection message should be kept is now that its time stamp is at least as large as the time of the process which received it. Such a message will be forwarded only when the process is in a stable state and the above time constraint still holds. As in the previous solution detection messages can overtake each other.

This solution has the following form :

```
$P_i$::    OK:=false ; SENT:=false ; FAIT:=false ; RECU:=false ;
           EMIS:=false ; FORWARD:=false ; T:=0 ; TIME:=0 ;
    *[  □   ¬ FAIT ; $g_{i,j}$ → OK:=false ; SENT:=false ; $S_{i,j}$
      $j \in \Gamma_i$
        □ ¬ FAIT ; $B_i$ ; ¬OK → T:=T+1 ; OK:=true ;
                                [TIME<T → FORWARD:=false □ TIME ≥ T →  skip]
        □ ¬ FAIT ; OK ; ¬ FORWARD ; ¬ SENT ; $P_{i+1}$! detection-message (T,1)
                                 → SENT:=true
        □ ¬ FAIT ; $P_{i-1}$?detection message (TIME, COUNT) →
          [COUNT = 2n → FAIT:=true
           □ COUNT < 2n →
             [TIME < T → skip -- purge the message
              □ TIME ≥ T → COUNT:=COUNT+1 ;
                           FORWARD:=true
             ]
          ]
        □ ¬ FAIT ; OK ; FORWARD ; $P_{i+1}$! detection-message (TIME, COUNT)
                     → FORWARD:=false ; SENT:=true ; T:=TIME
        □ ¬ RECU ; $P_{i-1}$?termination message → RECU:=true ;
                                                 FAIT:=true
        □ FAIT ; ¬ EMIS ; $P_{i+1}$!termination-message → EMIS:=true
     ]
```

This program differs in various aspects from the "store and forward" version from the previous section. First, sending of its own detection message by P_i is additionally guarded by the condition $\neg$ FORWARD indicating that no detection message remains to be forwarded. FORWARD is set to true when a detection message with a sufficiently large time stamp ($TIME \geq T$) is received. This message remains to be kept as long as the local time T does not exceed its time stamp. If this happens then FORWARD is set to false which amounts to a purge of the message.

Secondly, the forwarding of a pending detection message only when the process is passive is ensured by the additional guard OK. Also note that the updating of the local time and setting SENT to true takes place only *after* forwarding a pending detection message. Finally, the incrementing of the local time by 1 happens only in one place in the program.

In spite of the above differences the correctness proof of this solution is essentially the same as that of the version given in the last section and is omitted.

Observe that the above approach does not lead to any improvements when applied to the "store and forward" versions of the solutions from sections 3 and 5. Indeed, detection messages received by an active process will be purged once the process becomes passive because the local time will then exceed the time stamp of the pending detection message.

10. A SOLUTION WITH TIGHTLY SYNCHRONIZED VIRTUAL CLOCKS

Solutions presented in sections 5 and 6 are markedly different from that of section 3 in that to detect termination a detection message has to make two full cycles instead of one. This difference is due to the inaccuracy of the local clocks. In the solutions from sections 5 and 6 these clocks are synchronized exclusively by means of detection messages. The traffic of the original communications is transparent to the clocks and as noticed in section 5 an existence of such a communication can pass unnoticed. To understand better the nature of the problem consider a hypothetical execution of the program from section 6 with $n \geq 3$ in which a detection message with $T = 1$ sent by P_1 makes successfully a full cycle. As already observed in section 5 this does not yet mean that termination is detected since an original communication could take place between P_i and P_j for $1 < i+1 < j \leq n$ at the moment when the detection message was "between" them.

In the solution from section 3 such a "hidden" original communication cannot pass unnoticed. It will be detected by a necessarily increased value of T_j which will cause a purge of the detection message sent by P_1.

To incorporate this observation into the framework in which local clocks are used, we have to synchronize the clocks also after every original communication. This has to be done *irrelevantly* of the direction of the communication. To see the use of this procedure let us reconsider the above hidden communication between P_i and P_j. As a result of it P_j will set its clock to $\max(T_i,T_j)$, i.e. to at least 1 and consequently when P_j becomes passive its clock value T_j will be at least 2. Thus the detection message of P_1 will be purged as desired.

Summarizing, this solution is obtained by modifying the programs from section 5 and 6 as follows. The lines involving basic communications now read

```
$\neg$ FAIT, $g_{i,j}$ $\rightarrow$ SYNCHR(i,j) ; OK:=false ; SENT:=false ; $S_{i,j}$  where

SYNCHR(i,j) $\equiv$ $P_j$! T ; $P_j$?T  if  $g_{i,j}$ contains
                  an output command  $P_j$ ! t,
               $P_j$?T' ; T:=max(T,T') ; $P_j$! T  if  $g_{i,j}$ contains
                  an input command  $P_j$?x.
```

Moreover COUNT is now compared with n as in the solution from section 3.

The correctness proofs of the above solutions are based on a straightforward combination of the arguments used in the proofs given in sections 4 and 7 and are left to the reader.

An obvious disadvantage of these solutions are the overhead caused by a frequent clock synchronization. Note also that the same modifications can be applied to the solutions from sections 8 and 9.

11. RELATED WORK

The problem of detection of termination in distributed systems has been extensively studied in the literature. The problem was originally posed and solved in the framework of CSP programs by Francez [F]. Francez called it *the distributed termination problem*, a name we adopted in this paper. A different solution was proposed in an abstract setting similar to the one of section 1 by Dijkstra and Scholten [DS]. Several other solutions appeared since then. They were presented in the framework of CSP programs in Francez and Rodeh [FR], Misra and Chandy [MC], Francez et al. [FRS] and Rana [R]. For other models of distributed computing solutions were presented in Dijkstra et al. [DFG], Gouda [G], Lerman and Schneider [LS], Misra [M] and Topor [T]. Also solutions for an extension of CSP allowing dynamic creation of processes were proposed by Cohen and Lehmann [CL] and Lozinsku [L].

Symmetric solutions were first studied by Rana [R]. His algorithm uses real time clocks. Even though based on a sound idea it is for several reasons incorrect. Our paper is motivated by an effort to correct Rana's solution. His algorithm corresponds to a direct translation of the abstract solution consisting of rules 1 and 2 given in section 1. As noted in section 3 such a translation leads to an algorithm containing deadlock. Moreover, the final termination wave was improperly built into this algorithm. The proposed program does not satisfy properties 2,3 and 4 of section 2 and property 1 is voidly satisfied since the program never properly terminates !

His version of transformed process P_i (after correcting some obvious misprints) looks as follows :

```
$P_i$:: *  [$S_i$
          □ $B_i$ → BTIME$_i$:=CLOCK-TIME ;
                   TIME:=BTIME$_i$ ;
                   COUNT:=1 ;
                   $P_{i+1}$!detection-message(TIME,COUNT)
          □ $P_{i-1}$?detection-message(TIME,COUNT) →
              [COUNT = n →
                 $P_{i+1}$!termination-message;
                   TERMINATE
               □ COUNT ≠ n →
                 [¬$B_i$ → purge the message
                  □ $B_i$ →
                   [TIME < BTIME$_i$ → purge the message
                    □ TIME ≥ BTIME$_i$ →
                      COUNT:=COUNT+1 ;
                      $P_{i+1}$!detection-message (TIME, COUNT)
                   ]
                 ]
              ]
           □ $P_{i-1}$?termination-message →
                 $P_{i+1}$!termination-message ;
                   TERMINATE
           ]
```

(In Rana's proposal S_i is replaced by S'_i but it is not explained how S'_i relates to S_i. The above discussion did not depend on this item).

We leave to the reader checking validity of our comments concerning the above algorithm.

An obvious disadvantage of the solutions presented in this paper is that the (virtual) clock variables can assume arbitrarily large values. A natural question to ask is whether there exist symmetric solutions to the distributed termination problem in which all variables are bounded. The answer is positive. Such solutions for a ring configurations were recently constructed by the second author in Richier [R] and for arbitrary strongly connected graphs in Bougé [B2]. These solutions are not based on the idea of clock synchronization.

12. CONCLUSIONS

Symmetric solutions to the distributed termination problem for the ring configuration are necessarily more complex than the asymmetric ones of [FRS] and [DFG] because more parallelism in the system is possible. In the asymmetric solutions two types of activities could take place in parallel - exchanges of basic communications and exchanges of the control (detection) messages. But the control messages were originated by one a priori designed process so at each moment at most one control message was present in the system. In contrast, in the case of symmetric solutions each processes can initiate a wave of control messages and the system can contain at a given moment up to $n-1$ control messages. Moreover, different processes at roughly the same moment can initiate a wave of termination messages, so the system can also contain several termination messages at the same time. In spite of these differences our second, third and fourth solutions are similar in nature to the solution of [FRS] in that the global stability of the system is detected essentially using the so-called *interval assertions* which allow to test whether a value of B_i has changed in a given interval of time. In our solutions this method is incorporated by the use of virtual clocks and (virtual) time stamps of the detection messages.

We conclude by observing that when clocks are synchronized *only* by means of detection messages then a detection message has to make *two* full cycles in order to detect termination. In contrast, when the clocks are synchronized using arbitrary messages containing time stamps then *one* cycle is already sufficient. Thus there is a direct trade off between the work needed for clock synchronization and the time needed for detection of the termination.

Acknowledgements. We thank L. Bougé and Ph. Darondeau for helpful comments on the first version of this paper.

REFERENCES

[B1] BOUGE, L., Symmetric election in CSP, Tech. Report 84-31, LITP, Université Paris 7, 1984.

[B2] BOUGE, L., More about the snapshot algorithm : repeated synchronous snapshots and their implementation in CSP, Tech. Report 84-56, LITP, Université Paris 7, 1984.

[CL] COHEN, S. and LEHMANN, D., Dynamic systems, their distributed termination and dead-lock detection, in : Proc. Symp. on Principles of Distributed Computing, Ottawa, pp. 29-33, 1982.

[DFG] DIJKSTRA, E.W., FEIJEN, W.H. and van GASTEREN, A.J.M., Derivation of a termination detection algorithm for distributed computations, Inform. Processing Letters 16, 5, pp. 217-219, 1983.

[DS] DIJKSTRA, E.W. and SCHOLTEN, C.S., Termination detection for diffusing computations, Inform. Processing Letters, 11, 1, pp. 1-4, 1980.

[F] FRANCEZ, N., Distributed termination, ACM-TOPLAS, 2,1, pp. 42-55, 1980.

[FR] FRANCEZ, N. and RODEH, M., Achieving distributed termination without freezing, IEEE Trans. Soft. Eng., SE-8, 3, pp. 287-292, 1982.

[FRS] FRANCEZ, N., RODEH, M. and SINTZOFF, M., Distributed termination with interval assertions, in : Proc. Int. Colloq. Formalization of Programming Concepts, Peniscola, Spain, Lecture Notes in Comp. Science, vol. 107, 1981.

[G] GOUDA, M.G., Distributed state exploration for protocol validation, Tech. Report 185, Dept. of Computer Sciences, University of Texas at Austin, 1981.

[H] HOARE, C.A.R., Communicating sequential processes, CACM 21, 8, pp. 666-677, 1978.

[L] LAMPORT, L., Time, clocks and the ordering of events in a distributed system, CACM 21, 7, pp. 558-565, 1978.

[LR] LEHMANN, D. and RABIN, M.O., On the advantages of free choice : A symmetric and fully distributed solution to the dining philosophers problem, Proc. of the 8th Annual ACM Symp. on POPL, Williamsburg, Virginia, January 1981.

[LS] LERMAN, C.W. and F.B. SCHNEIDER, Detecting distributed termination when processors can fail, Tech. Report TR 80-449, Cornell University, Dept. of Computer Science, 1980.

[Lo] LOZINSKII, E.L., Yet another distributed termination, Tech. Report 84-2, Dept. of Computer Science, The Hebrew University of Jerusalem, 1984.

[M] MISRA J., Detecting termination of distributed computations using markers, in : Proc. 2nd Annual Symp. on Principles of Distributed Computing, Quebec, pp. 290-294, 1983.

[MC] MISRA, J. and CHANDY, K.M., Termination detection of diffusing computations in communicating sequential processes, ACM-TOPLAS, 4, 1, pp. 37-43, 1982.

[R] RANA, S.P., A distributed solution of the distributed termination problem, Inform. Processing Letters 17, 1, pp. 43-46, 1983.

[Ri] RICHIER, J.-L., Distributed termination in CSP : symmetric solutions with minimal storage, Proc. STACS 85, to appear.

[T] 'TOPOR, R.W., Termination detection for distributed computations, Inform. Processing Letters 18, 1, pp. 33-36, 1984.