ACHIEVING DISTRIBUTED TERMINATION
WITHOUT FREEZING

by

Nissim Francez* and Michael Rodeh**

Technical Report No. TR-180

July 1980.

† This work was partially supported by NSF grant No.MCS 78-673 while
Dr. Nissim Francez was at USC, Los Angeles.

* Computer Science Dept., The Technion-Israel Institute of Technology,
Haifa, Israel.

** IBM-Israel Scientific Center, the Technion City, Haifa, Israel.
ABSTRACT

An efficient algorithm for achieving distributed termination without introducing new communication channels and without delaying the basic computations ("freezing") is presented. The algorithm is related to the methodology of designing distributed programs where the programmer is relieved from the problem of distributed termination. An informal correctness proof and complexity analysis are included.

Key words: Distributed programs, concurrency, distributed termination, CSP, communication.

1. INTRODUCTION

The subject of communication processes is gaining increasing attention from several directions; for example, languages and semantics (Hoare [9], Brinch Hansen [2], Milne and Milner [13], Sintzoff [14], Francez et al. [7], Feldman et al. [8]), algorithms and correctness proofs (Dijkstra [3, 4, 5], Sintzoff [15], Lamport [12], Kimura [11]), and implementations (Hunt [10], Schwarz [16]), as well as many others. This multitude indicates the importance of obtaining efficient solutions to problems which arise in distributed programming. One significant such problem is that of distributed termination posed and solved in Francez [6] where other aspects of termination are also discussed. The solution presented here is better; it is more efficient and it allows the basic activity of the processes to continue without "freezing" is introduced.

The model of computation considered here is similar to the one assumed by Hoare [9]: There is a fixed network of processes, capable of communicating in pairs along predefined channels. Communication means a message passing from a named source process (sender) to a named destination process (receiver). A communication may occur between any two processes which agree to communicate. Although a process may be willing to communicate with several processes (as alternatives), a communication will occur only with at most one of them at a time. No assumptions may be made on the scheduling strategy which determines which alternative is going to occur.

Section 2 briefly describes the distributed termination problem and the solution suggested in [6]. Section 3 presents an improved
solution. Section 4 consists of an informal correctness proof and complexity analysis. An example is given in Section 5.

2. THE DISTRIBUTED TERMINATION PROBLEM

Suppose a program $P$ is required which transforms initial states into final states satisfying upon termination a predicate $B$. Moreover, by some insight, the state space has been split into $n > 1$ disjoint subspaces and predicates $B_1, \ldots, B_n$ over those subspaces are found such that

$$(GI \land \bigwedge_{i=1}^{n} B_i) \rightarrow B$$

where $GI$ is a global predicate.

To find such $B_i$ certain augmentations of the subspaces may be needed. Based upon such decompositions a reasonable candidate for $P$ is a parallel program $\hat{P} := [\hat{P}_1 \| \hat{P}_2 \| \ldots \| \hat{P}_n]$ where:

(i) Each $\hat{P}_i$ is a communicating process with the corresponding subspace as its own local state space. These communications will be called basic.

(ii) By means of some finite sequence of basic communications preserving $GI$ as an invariant $\hat{P}$ reaches global stability, in which every process $\hat{P}_i$ is locally stable namely, $B_i$ holds, $\hat{P}_i$ initiates no further communication and waits in its top level being ready to accept communication initiated by another process. For the exact role of the invariant see [1]. In the sequel we always assume that such an invariant is maintained, without explicitly referring to it.
The still missing link is the termination of \( \hat{P} \) once global stability is reached. Observe that global stability is actually a deadlock situation where all processes are ready to accept but none is willing to initiate communication. In order for \( \hat{P} \) to terminate, all \( \hat{P}_i \)'s must terminate. Since for \( \hat{P}_i \) the property of being locally stable is in general non-monotonic, \( \hat{P}_i \) cannot decide to terminate on its own even when \( B_i \) is true, for there may be some process \( \hat{P}_j \) not yet locally stable which initiates communication with \( P_i \), possibly destroying the local stability of \( \hat{P}_i \).

Being unable to terminate according to some local criterion, one could envisage a process \( \hat{P}_0 \) inspecting the local states of all other processes by means of shared memory. The disjointness of the processes in the model of computation prohibits this possibility.

Hence, we are led to a solution which detects global stability by means of extra (control) communications rather than sharing memory. The problem of designing such control communications, without adding new channels (besides those which exist in \( P \)) is referred to as the problem of distributed termination. A solution adds to every \( \hat{P}_i \) a control communication part \( C_i \) (as a new set of alternatives in the top level loop of \( \hat{P}_i \)) and also requires some modifications to the text of \( \hat{P}_i \) itself. Such modifications are needed:

(i) For interface with the control communication part \( C_i \).

(ii) For being able to express a stronger version \( B_i \) of \( \hat{B}_i \) which besides \( \hat{B}_i \) implies local stability, namely, assures that \( \hat{P}_i \) will not initiate any basic communication.

A desirable solution is one which modifies each \( \hat{P}_i \) only slightly.

Let \( P = [P_1 \| \ldots\| P_n] \) be that program resulting from \( P \) by adding
appropriate control communication sections $C_i$ and making the above mentioned modifications.

Before presenting an improved solution, the structure of the solution in [6] is briefly described. First an (arbitrary) rooted spanning tree $T$ of the communication graph is chosen. (Directions are ignored; if the underlying undirected graph is not connected, the same construction is repeated for each connected component.) The sub-graph $T$ is the carrier of control communication. Each control cycle consists of four waves. A wave $W$ is a pattern of communications over a tree of channels. A wave is directed either upwards (spreading from the leaves to the root) or downwards. In an upwards wave every internal node must first receive the wave from all its children and later send it to its parent. Likewise, in the downwards case, every internal node must first receive the wave from its parent and later spread it to all its children. In both cases the order of communication with the children is immaterial. A communication in a wave may further depend on local conditions. All communications of a wave may be interleaved with other activities. Therefore, it must be verified that the target of the wave is eventually reached. An important property of the wave pattern is that along each channel, only one communication occurs.

Following is the description of the four waves of [6]:

W1 - Spreading down an inquiry about local stability.

W2 - Every node $P_i$ sends up an accumulated reply that is true if and only if all the nodes in the subtree rooted at $P_i$ are locally stable (i.e. all nodes have themselves replied with true, and $P_i$ itself is locally stable). To avoid $P_i$ becoming locally unstable after having reported local stability, all
basic communications are "frozen", thereby creating potentially significant delays.

W3 - Depending on the outcome of W2, the root may send down either a terminating signal (not discussed here) or a signal which allows resumption of basic communication ("unfreezing").

W4 - A signal is sent up by a node to its parent after having received it from all its children and either being locally stable or having performed at least one basic communication since the last W3. W4 guarantees the absence of an infinite control loop. When W4 reaches the root a new control cycle may be initiated.

Let $ba(P)$ and $cc(P)$ be the number of basic and control communications of $P$ respectively. Then $cc(P)$ may be as large as $4n \cdot ba(P)$ for some programs $P$ with $n$ processes and a certain scheduling of communications. In [6] the role of the invariant GI was not stressed as it should.

3. AN IMPROVED SOLUTION

The alternative solution to the distributed termination problem proposed here is better than the one just described in two major respects:

(i) The number of control communications $cc(P)$ is bounded by $2n \cdot ba(P) + n$ but for many programs $P$, $cc(P) \leq 3n$ while the former solution may require $O(4n \cdot ba(P))$ control communications;

(ii) No "freezing" of basic communications takes place, namely, no basic communication is necessarily delayed in favor of control communication. All control communications are always alternatives to basic communications, whereas in the previous algorithm they
are sometimes the only possibility. Still nothing guarantees that basic communications are preferred over control communications. As a result, the behavior may be bad if certain resolutions to nondeterminism are employed by the underlying scheduling.

The control communication scheme suggested below is indulgent in the following sense: For each two communicating processes \( P_i \) and \( P_j \) one of them, \( P_i \) say, is passive, i.e. has an alternative which is an unconditional readiness for communication with \( P_j \); the other process, \( P_j \), is active, i.e. has some condition which when true enables the communication with \( P_i \); the enabling of the communication is intentionally separated from its actual occurrence, thereby allowing basic activity in the meantime. A complementary concept is that of an insistant communication scheme. In such a scheme there may exist a communication which once enabled prevents any other activity until it occurs (even if there are alternative enabled communications). The algorithm described in [6] is insistant.

Here indulgency is applied to the spreading of waves where the arrival of a wave to a node (as part of its enabling) is separated from its further spreading. As will be seen from the text, indulgency is implemented in CSP by a combination of Boolean and i/o guards.

In general, the indulgency of a communication scheme reduces the danger of a deadlock situation, which is more likely in case of insistant communication. It also allows for a higher degree of concurrency in the whole program. (Compare also to the discussion in [7] about the relationship between the local and global nondeterminism on the one hand and deadlock on the other.)
The proposed algorithm uses a spanning tree as in [6]. The control communication is arranged in three conceptual waves: W1, W2, and W3. Then a way to combine W1 and W3 to form a single wave is shown.

**W1** - A signal is indulgently spread up. It succeeds in passing a node $P_i$ only when $P_i$ is locally stable (and therefore every node $P_j$ in the subtree rooted at $P_i$ has been locally stable once since the last spreading of W3). To keep track of basic communications which might have occurred in $P_i$ after reporting local stability a flag is used. The flag is set 'on' in conjunction with each basic communication (this is part of the above mentioned modification of the basic part, for interface with the control communication), and set 'off' in conjunctions with the reporting of local stability. The state of all flags is interrogated in W2 and reported in W3.

**W2** - Upon receiving W1 the root initiates an indulgent spreading down of a wave, the unique task of which is to enable the initiation of W3.

**W3** - This wave spreads up indulgently and inspects the states of the nodes in the dominated subtrees. If W3 indicates that after the last W1 no further basic communication occurred in any process then the root may conclude that global stability has been reached and terminate. This conclusion strongly depends on the fact that if some process *did* perform some basic communication *after* having reported by W3 that it did not, then there is another process which did a basic communication *before* having reported W3, and hence will report it eventually. This follows from the fact that a locally stable process may not initiate any communication.
The root may not overlook this basic communication and thus may not terminate prematurly. A side effect of W3 is the enabling of a new W1, thereby starting a new control cycle in case termination cannot occur. At no node may the new W1 overtake the old W3. Since W3 may be postponed until W1 arrives it may be combined with the new W1 to form a single wave, denoted by W1-3.

Next assume that a program \( P' = [P_1' || ... || P_n'] \) is given for which reaching global stability is guaranteed, and the appropriate Boolean conditions \( B_i \) which imply local stability are known. If the root were aware of the program being globally stable, it could initiate a wave signaling termination to all processes. This aspect of the distributed termination problem is not discussed here since it has a structure identical to that of W2 above.

To obtain a program \( P \) which is functionally equivalent to \( P' \) but explicitly terminates, the following changes are made: For each process \( P_i' \), all basic communications are accompanied by an assignment of the value \text{true} \ to the local variable \text{advance} \ (which means — an advance towards the globally stable state has been made in \( P_i' \)). This is the only change made in the text \( P_i' \) itself. In addition each \( P_i' \) is augmented by a control communication part \( C_i \) defined separately for the leaves, the internal nodes and for the root. \( C_i \) is inserted as an additional set of alternatives at the top level loop of \( P_i' \).

(i) Let \( P_i' \) be a leaf. \( C_i \) is expressed in some notationally extension of the CSP language [9]. A very short description of the extensions appears in the Appendix.
$C_i$: comment: $w2$-arrived is a Boolean variable (initially true) which
manages the transmission of the waves. $w2(\_)$ is a constant message
representing $W2$. advance is a Boolean variable initially true.
$p(i)$ is the index of the parent of node $i$ in the spanning tree
end of comment

$L1: \Box B_i \land w2$-arrived $\land P_p(i)! \implies$ advance $\land w2$-arrived := false; advance := false
comment: If $W1$-3 has not been sent yet ($w2$-arrived = true) and $P_i$
is locally stable ($B_i$ = true) and the parent $P_p(i)$ of $P_i$ is ready to
accept the Boolean value advance then sent it, turn $w2$-arrived to
false (to avoid sending $W1$-3 again) and reset advance to false
(to indicate that no basic communication has occurred since the
last $W1$-3 end of comment

$L2: \Box P_p(i)! \land w2(\_)$ $\implies w2$-arrived := true
comment: The acceptance of $W2$ is recorded in the Boolean variable
$w2$-arrived end of comment

(i) Let $P'_i$ be an internal node.

$C_i$: comment: $\Gamma_i$ is the set of indices of the children of $P_i$ and
$\gamma = |\Gamma_i|$. The variables send-$w2[j]$ (initially false) indicate
whether $W2$ has been sent to $P_j$ for $j \in \Gamma_i$. $W13[j]$ stores the value
of $W1$-3. $m$ counts the number of children of $P_i$ which have already
transmitted $W1$-3 and is initially 0. advance is initially true
end of comment

$I1: \Box P_j?w13[j]$ $\implies m := m + 1$ $\forall j \in \Gamma_i$

$I2: \Box (\gamma = m) \land B_i \land P_p(i)! \implies$ (advance $\lor (\bigvee_{j \in \Gamma_i} w13[j])$) $\implies m := 0; \implies$ advance := false
comment: If $W1$-3 has been received from all $\Gamma_i$ and $P_i$ is locally
stable then the accumulated value of advance in the subtree rooted
at $P_i$ is sent to the parent of $P_i$ end of comment

$I3: \Box P_p(i)! \implies w2(\_)$ $\implies \prod_{j \in \Gamma_i} \text{send}-w2[j]$ := true
I4:  \( \forall j \in \Gamma \) \( \text{send-w2}[j], P_j \cdot \text{w2}(\cdot) \rightarrow \text{send-w2}[j] := false \)

(iii) For the root, \( P_{root} \):

\[ C_{root} : \text{comment: The initial values are: } m=0; y=|\Gamma_{root}|; \text{advance}=true \]
\[ \text{and } \text{send-w2}[j] = \text{false} \text{ for all } j \in \Gamma_{root} \]  
end of comment

R1:  \( \forall j \in \Gamma_{root} \) \( P_j \cdot \text{w13}[j] \rightarrow m := m+1 \)

R2:  \( (m=y); B_{root} \rightarrow [\lnot \text{advance}; \bigwedge_{j \in \Gamma_{root}} \text{w13}[j] \rightarrow \text{halt} \]
\[ \text{advance} \lor (\bigvee_{j \in \Gamma_{root}} \text{w13}[j] \rightarrow m := 0; \text{advance} := \text{false} \]
\[ \prod_{j \in \Gamma_{root}} \text{send-w2}[j] := \text{true} \]

comment: If the entire W1-3 arrived \( (m=y) \) and the root is locally stable then if all the values of \text{advance} are \text{false} (implying global stability has been reached) then halt (in fact, an additional terminating wave should be spread among the processes). Otherwise, a new wave W2 is enabled end of comment:

R3:  \( \forall j \in \Gamma_{root} \) \( \text{send-w2}[j]; P_j \cdot \text{w2}(\cdot) \rightarrow \text{send-w2}[j] := \text{false} \)

Note that the above transformation of \( P' \) to \( P \) is mechanical and can be done by a pre-processor, thereby relieving the programmer from this concern. We believe that finding such \( P' \) and applying the transformation will be easier in many cases than finding directly a program \( P \) that is designed to terminate in the first place.
4. CORRECTNESS AND ANALYSIS

The correctness of the algorithm relies on the following idea:

Assume a sequence \textit{test}_1, \textit{test}_2, \ldots has been designed such that:

(i) Every test inquires whether global stability has occurred. If \textit{test}_i concludes that global stability has not been reached then \textit{test}_{i+1} is initiated. Otherwise no further tests take place.

(ii) During a test which reports non-stability, at least one basic communication occurs.

Notice that if every test is finite then the whole process is finite, since the number of basic communications is given to be finite. For every \( i \), \textit{test}_i comprises two steps: First, each process \( P_j \) is inspected for being locally stable. If it is not then the algorithm waits until \( P_j \) becomes locally stable. At that moment denoted by \( t^i_j \), a local variable \textit{advance}_j is set to \textit{false} (such a moment exists, since eventual global stability is given). Any subsequent basic communication involving \( P_j \) will turn \textit{advance}_j to \textit{true}. Let \( T^i = \max_j t^i_j \). At some later time instances \( t^i_j > T^i \) the variable \textit{advance}_j of every process \( P_j \) is inspected. If all of them have \textit{false} as their value, then this was true at time \( T^i \) (otherwise someone has changed \textit{advance}_j from \textit{false} to \textit{true}, and then back to \textit{false}, which is impossible), and therefore at that time all processes were locally stable. This implies the global stability and correctness of the whole program. Otherwise at least one process \( P_j \) has been involved in some basic communication prior to time \( T^i \) and therefore before \( T^i_j \) (otherwise all processes were locally stable in time \( T^i \), and therefore could not become active later). Thus the value of \textit{advance}_j in \( P_j \) is \textit{true} and \textit{test}_{i+1} will occur.

It is not necessary to have \textit{test}_{i+1} start strictly after \textit{test}_i finishes.
Having $i^j_j \leq i^{j+1}_j$ for every $j$ suffices. In fact, by unifying $W_1$ and $W_3$ we have $i^j_j = i^{j+1}_j$.

In terms of the algorithm of Section 3, the wave $W_1-3$ is responsible for:

(i) detecting local stability;
(ii) accumulating the values of the local variables $\text{advance}$, and
(iii) resetting the variables $\text{advance}$ for the next test.

The moment in which the root completes the first communication of $W_2$ plays the role of $T^*_k$. The only task of $W_2$ is to invoke $W_1-3$ and assure that it does not start before $T^*_k$. Having received $W_1-3$ the root may check whether all variables $\text{advance}$ have the value false. As will be shown, nodes in the "front" of every wave may be busy only for a finite time before becoming ready to further propagate the wave. Acceptance of waves is always unconditional and is therefore eventually enabled (by the finiteness of all other activity).

To deduce the properties of $W_1-3$ and $W_2$ from the program segments $C_i$, consider the initial state of the processes. For internal nodes, $m = 0$ and $\text{send-w}_2[j]=false$. Therefore none of them is able to send a message (Line L2). The root cannot send any message since $\text{send-w}_2[j]=false$ (Line R2). The only nodes which may initiate a message are the leaves. They may send the value of $\text{advance}$ (the wave $W_1-3$ in Line L1). The parent of a leaf $P_k$ may not be always busy since his basic activity is finite and the only other control communication in which he may be involved at this point is the reception of $W_1-3$ from some of his other children (this will be shown below), the number of which is finite. In conjunction with sending $\text{advance}$ to its parent, the leaf sets $\text{advance}$ to false. Since the leaf is locally stable (the $B_k$
is true, see Line L1), the instant at which the communication occurred may serve as the required \( t^1_k \) of the correctness proof. Another results of this communication are the setting of \( w2\text{-}arrived \) to false (disabling for some time the sending of advance again), the update of \( w13[k] \) at the parent and the increase of \( m \) by one [Line II]. When \( m \) becomes equal to \( y \) (Line II) in \( P_i \), this process may initiate a communication with its parent (provided that \( P_i \) is locally stable) sending the accumulated value of advance (Line I2). By a similar reasoning it follows that the father may not be busy forever. If the accumulated value of advance is actually sent then advance is set to false and \( m \) is set to 0 disabling the resent of \( W1\text{-}3 \) until another wave \( W1\text{-}3 \) arrives from all the children. This analysis establishes the wave structure of the communication concerning \( W1\text{-}3 \).

Consider next the case of \( W1\text{-}3 \) reaching the root. Unless \( W1\text{-}3 \) is received from all the children, no communication may be initiated by the root. When the last child sends the \( W1\text{-}3 \) \( m \) becomes equal to \( y \). At this moment (taken as \( t^1_1 \)) the accumulated value of the variables advance are checked. If all of them are false then global stability has been reached and \( P\text{-}root \) may halt (in fact it must initiate a final wave to indicate to all processes that they may halt). Otherwise, \( send\text{-}w2[j] \) are set to true (Line R2). By now the root is the only node which can initiate a message (Line R3). The message which the root may send is \( w2( ) \) to one of his children (none of which may be always busy). \( W2 \) is on its way to the leaves.

The acceptance of \( w2( ) \) by an internal node enables the transmission of \( w2( ) \) to all its children and no other communication. Once a leaf accepts \( W2 \), \( W1\text{-}3 \) is enabled. \( W1\text{-}3 \) collects information concerning the
local stability of the nodes during the time which has passed since the last setting of advance to false. The instant of sending \( W_{i-3} \) from a node \( P_j \) is the \( \tau^i_j = \tau^{i+1}_j \) mentioned above.

Both \( W_{i-3} \) and \( W_2 \) traverse once each edge of the spanning tree. Therefore at most \( 2(n-1) \) control communications may be performed per control cycle. Note that for the worst case (i.e. the worst scheduling) a control cycle may occur after every basic communication. The more essential improvement is that the control communications are never necessarily performed in order to enable proper continuation. An important property of the construction, absent in [6] is that if the \( B_i \)'s are all monotonic (in the sense that once \( B_i \) becomes true it remains so) then only one control cycle will be performed.

5. EXAMPLE — COMPUTING THE GCD OF \( n \) NUMBERS

Let there be given \( n \geq 2 \) positive integers \( \sigma_1, \ldots, \sigma_n \). A distributed computation of \( u = \gcd(\sigma_1, \ldots, \sigma_n) \) is a distributed algorithm which starts with \( n \) variables \( x_1, \ldots, x_n \) with the \( \sigma_i \)'s as their initial value and ends with \( x_i = u \) for every \( 1 \leq i \leq n \). Let

\[
B = \bigwedge_{i=1}^{n} (x_i = u).
\]

Naturally, by choosing \( \hat{B}_i \) to be the predicate \( x_i = u \),

\[
(\bigwedge_{i=1}^{n} \hat{B}_i) \Rightarrow B.
\]

This suggest \( n \) processes \( \hat{P}_1, \ldots, \hat{P}_n \) connected along a straight line, trying to establish \( \hat{B}_i \) with \( x_i \) in their local memory. They will communicate (each with its two neighbors, and the end-processes with their only neighbor) in such a way that the relation...
is kept invariant. However, \( u \) is unknown to each process, hence the above \( \hat{B}_i \) is not a local predicate. Observing that

\[
x_i = u \land x_{i-1} = u \Rightarrow x_i = x_{i-1}
\]

the predicate \( x_i = x_{i-1} \) could be used instead. Still \( \hat{P}_i \) must receive a copy \( y \) of \( x_{i-1} \) from \( \hat{P}_{i-1} \) by communication and record in some local Boolean variable, say \( \text{lequal} \), whether \( x_i = y \) is true. Defining \( B_i \) to be \( \text{lequal}_i \) (the local \( \text{lequal} \)) yields:

\[
(\bigwedge_{i=2}^{n} \text{lequal}_i) + (\bigwedge_{i=2}^{n} x_i = x_{i-1}).
\]

Next, we have that

\[
(GI \land \bigwedge_{i=2}^{n} B_i) \rightarrow \bigwedge_{i=1}^{n} (x_i = \text{GCD}(a_1, \ldots, a_n)) = B.
\]

Hence, each \( \hat{P}_i \) must be ready to send to its right neighbor its \( x_i \), and alternatively, to receive from its left neighbor the value \( y \) (a copy of \( x_{i-1} \)). A process which receives \( y \) may use it to update the current value of its local variable \( x_i \) according to the following expression:

\[
y \leq x_i \rightarrow \text{if } y \mid x_i \text{ then } x_i + y \\
\text{else } x_i + x_i \mod y.
\]

If \( y \mid x_i \) then \( \text{lequal}_i \) is assigned the value \text{true} since \( y = x_i \) after the update of \( x_i \). Observe that \( x_i \) should not be sent more than once before being changed. A local Boolean variable \( \text{r} \) (\text{ready to send right}) may be employed as a guard to avoid sending \( x_i \) again and again.
An obvious flaw is that whenever \( x_1 > x_2 > \ldots > x_n \) no progress towards obtaining the GCD may occur. Any process must be ready to send his \( x_i \) to both his right and left neighbor. Again, a flag \( rsl \) (ready to send left) will prevent needless communication of \( x_i \) to the left. Note the preservation of GI during communication.

In conclusion, the local predicate for expressing local stability becomes

\[
B_i \equiv (\text{lequal} \land \lnot (rsl \lor rsl))
\]

and when reached, leaves \( P_i \) with the only alternative of being ready to accept communication from either neighbor. We are led to the parallel program \( P'_i ::= [P'_1 \parallel \ldots \parallel P'_n] \) given below. Notice that this program is nonterminating and the program segments \( C_i \) discussed in Section 3 must be installed to transform it into a terminating problem. Therefore, the fact that \( \text{lequal} \) is set but not used in the program should not surprise: it plays an important role in the detection of local stability and is therefore used in the \( C_i \)'s. See [1] for a formal proof of a variant of this program.

\[
P'_1:: \text{rer}:=\text{true}; x:=a_1;
\]

*\[
\quad [ \text{rer} ; P'_2/x \rightarrow \text{rer}:=\text{false} \\
\quad \alpha P'_2/y + [y \geq x \rightarrow \text{skip} \\
\quad \alpha y < x \rightarrow \text{rer}:=\text{true}; \\
\quad \quad [y/x + x:=y \circ y]/x + x:=x \mod y]
\]

]
$P'_n:: rel:=true; lequal:=false; x=c_n$

* [ rel; $P_{n-1}!x + rel:=false$

  $P_{n-1}?y + [ y > x + skip$

  $y = x + lequal:=true$

  $y < x + [ y|x + x:=y; lequal:=true$

  $y|x + x:=x \mod y; lequal:=false; rel:=true$ ]

$P'_i:: \text{comment: This is the program for an internal node } 1 < i < n$

end of comment;

$rel:=true; rer:=true; lequal:=false; x=c_i$

* [ rel; $P_{i-1}!x + rel:=false$

  rer; $P_{i+1}!x + rer:=false$

  $P_{i-1}?y + [ y > x + skip$

  $y = x + lequal:=true$

  $y < x + rer:=true; [ y|x + x:=y; lequal:=true$

  $y|x + x:=x \mod y;$

  lequal:=false; rel:=true$ ]

$P'_{i+1}?y + [ y \geq x + skip$

  $y < x + lequal:=false; rer:=true; rel:=true;$

  $[y|x + x:=y \oplus y|x + x:=x \mod y]$

]}

For the sake of completeness let us demonstrate the effect of installing the program segments $C_i$ in the programs $P'_i$. The first thing to do is to choose some spanning tree. In our case it would be natural to use $P'_1$ as the root and $P_{i+1}$ as the (only) child of $P'_i$, $1 \leq i \leq n$. Next a setting of the variables advance to $true$ should be added to the text of the $P'_i$'s in conjunction with every communication. For the root ($P'_1$) the predicate $B_i$ is the simplest:

$B_i \equiv \sim rer$
and the following program is obtained:

$$P'_i:: \text{comment: The program of } P'_i \text{ after installing } C_i \text{ end of comment:}$$

\begin{verbatim}
rsr:=true; rst:=false; x:=o1; m:=0; γ:=1; send-w2:=false; advance:=true;
[x. rsr; P_2 !x \rightarrow advance:=true; rst:=false
  \dot{\rho} P_2?y \rightarrow \begin{array}{l}
  y \geq x \rightarrow \text{skip}
  \quad \text{[} y | x + x := y \; \dot{\rho} \; y | x + x := x \mod y \text{]}
  
\text{comment: Notice that the setting of } advance \text{ to } true \text{ is done only}
\text{in conjunction with a change in the value of } rsr \text{ end of comment.}
\end{array}
\]
\end{verbatim}

$$C_1:: \begin{array}{l}
P_2 !w13 \rightarrow m:=m+1
(m=γ); \sim rsr \rightarrow \begin{array}{l}
  \sim advance; \sim w13 \rightarrow \text{halt}
  \quad \begin{array}{l}
  \text{advance \lor } w13 + m:=0; advance:=false;
  \quad \text{send-w2:=true}
  \end{array}
  
\text{send-w2; } P_2 !w2( ) + \text{send-w2:=false}
\end{array}
\end{array}$$

6. CONCLUSION

Distributed termination should be considered as a tool for making
distributive programming easier. It is felt that additional such tools
must be developed. The concept of indulgency seems to be useful in other
applications. It reduces the danger of deadlock at the price of increasing
the danger of infinite loops. Global stability has been observed through
a continuous set of tests. It seems that a similar technique may be used
for solving synchronization problems.

ACKNOWLEDGEMENT

The authors wish to acknowledge useful discussions with S. Katz,
A. Segall and U. Vishkin.
REFERENCES


