DISTRIBUTED TERMINATION WITH INTERVAL ASSERTIONS

by

N. Franché, M. Roqeh, and M. Sintzoff

Technical Report #186

November 1980

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

** IBM-Israel Scientific Center, Technion City, Haifa, Israel

*** Philips Research Laboratory, Brussels, Belgium
SUMMARY

An efficient solution to the distributed termination problem is presented for a communication graph containing a (Hamiltonian) cycle. A top down development of the solution is given.

Keywords and Phrases: Communicating processes, distributed programs, distributed termination, disjoint variables, concurrency, interval assertions.

CR categories - 4.24.
1. INTRODUCTION

One of the problems that every attempt to construct a distributed program has to face is that of distributed termination, stated and studied in general by [F]. There it is observed that often it is relatively easy to distribute the global post condition \( B(\bar{y}) \), of a required program \( P \), into a conjunction \( \bigwedge_{i=1}^{n-1} B_i(\bar{y}_i) \) of \( n > 1 \) local post conditions (over disjoint portions \( \bar{y}_i \) of the state \( \bar{y} \)) so that:

\[
T1: \quad \bigwedge_{i=0}^{n-1} B_i(\bar{y}_i) \rightarrow B(\bar{y})
\]

T2: Communicating processes \( P_0, \ldots, P_{n-1} \) can be found, so that by means of some finite sequence of communications (to be called basic communications) the parallel composition \( P: = [P_0] \mid \ldots \mid [P_{n-1}] \) reaches a globally stable state, in which each \( P_i \) is locally stable, namely \( B_i(\bar{y}_i) \) holds, and \( P_i \) initiates no communication, but is ready to communicate with other processes that would initiate such a communication.

However, in a globally stable state, even though all processes are in their final local states, the whole program is deadlocked, since no process "knows" that the other processes are in final states, and all wait for each other to initiate communications.

Since the construction of such a program may be much easier than the construction of a terminating program (with the same post condition), the
availability of an automatic transformation (that can be applied by a translator) which transforms every concurrent program with the properties as described above into a terminating program is of much help, since such a transformation relieves the programmer from the (usually hard) problem of combining the overall program design with the distribution of termination, and is suggested as a methodological tool for constructing terminating distributed programs. Using such a tool is in accordance with the general principle of "separation of concerns", where local success is separated from global success.

Designing such a transformation by means of some additional communication (called control communication) was called in [F] the problem of distributed termination. An additional restriction on the acceptable solutions was posed in [F]:

R1. No new communication channels may be added for the sake of control communication.

Thus the transformed program and the resulting program must have the same communication graph.

In [F] the discussion was based on processes as expressed in Hoare's programming language CSP [H], which has a convention to allow distributed termination in some cases. Moreover, that convention was extended and that extension was used to solve the distributed termination problem.

Independently, a similar problem, but in a different setting of communicating processes, was investigated by [S1]. There, the
restriction R1 was not imposed. As a result, in [F] the control communi-

cation takes place along two-way channels forming a spanning tree of the

communication graph of the transformed program. Such a spanning tree
always exists. In the approach of [SI] the control communication takes
place along one-way channels forming a (Hamiltonian) cycle in the commu-
nication graph of the transformed program. Avoiding restriction R1 allows
the addition of a sufficient number of new channels until such a cycle is
created.

Besides this difference in ways of "covering" the nodes of the
communication graph, the two solutions suggested control communication
schemes with some striking similarities:

a. In both schemes, there is one designated process that has to "sense"
the global stability when it occurs, and initiate a termination wave
(the necessity of such a wave was stated in [F] as the theorem of the
pattern of distributed termination).

b. Both communication schemes were based on the idea of a communication
wave spreading (either down or up the tree, or along the cycle) as
"far" as possible, being aborted when "hitting" a process which is
not locally stable (at that moment).

c. In both schemes, each control cycle consists of four waves, which
could be described as:

(i) For each node, if its predecessors are "frozen" and if it
ended locally, then "freeze" it.

(ii) For each node, if its predecessors are "frozen" and if it
ended globally, then "freeze" it.
(ii) When all nodes are frozen, then for each node still locally ended, if its predecessors are "confirmed", then confirm it.

(iii) If the predecessors of a node are frozen but the node itself is either not ended locally or is "unfrozen", then unfreeze its predecessors.

(iv) If the predecessors of a node are confirmed but the node is not ended locally or is unconfirmed, then unconfirm its predecessors.

Thus, the overall communication complexity of both schemes was of the order of \(4nb\), where \(b\) is the number of basic communications needed to achieve global stability, and \(n\) is the number of processes involved.

Both schemes had the drawback of "freezing", i.e. disallowing basic communications while being in a control communication phase, in order to prevent the "behind the back" phenomenon, where a process was marked as locally stable, and by means of some basic communication that took place after this marking, became locally unstable without notifying this change in state.

In a more recent work [FR], an improved control communication scheme, also using a spanning tree, was presented.

Two aspects were improved: the communication complexity was reduced to \(2nb\), and, more important, the communication scheme was relieved of "freezing". The following ideas contributed to the improvement:

i. By distinguishing between the instance in which a communication is enabled and the instance in which it actually takes place, the
The concept of indulgent communication was formed. An indulgent communication scheme allowed any number of basic communications between two control communications. It was also observed that using indulgent communications reduces the deadlock possibilities.

ii. Instead of aborting a wave when it attempts to pass a process not locally stable, such a wave is delayed at that node until the node becomes locally stable. Thus, the arrival of a wave carries more information, which has still to be confirmed.

iii. Observing that in case a process that was marked as locally stable communicates, then either it is the case that its partner was not marked as such, or the partner itself communicated with an unmarked (yet) process, etc. (such a chain is finite). Hence establishing that at some instant all processes were marked and did not communicate after their being marked, is sufficient for deducing global stability. Establishing this fact can be done without freezing.

In the improved algorithm, each control cycle consists of two waves only: The first wave is initiated by the leaves, whereas the second wave is initiated by the root.

Two other works containing ideas connected to distributed termination are [DS] and [MS]. The purpose of this paper is to combine the ideas mentioned above to the case where a Hamiltonian cycle is present, as in [S1]. Moreover, we present a systematic derivation of the transformation using successive refinements, which clarifies the observation (iii) above. This derivation is presented in Section 2. Section 3 contains a case study exemplifying the usage of the suggested methodology, and Section 4 ends with a general discussion.
2. A SYSTEMATIC DESIGN OF DISTRIBUTED TERMINATION WITH CYCLES

In this section the method of interval assertions is first explained and then applied to solve the problem of distributed termination.

Assume that we follow up the execution of a distributed program

\[ P = [P_1, \ldots, P_n]. \]

We wish to observe that a global predicate

\[ \mathcal{E} = \bigwedge_{i \leq n} \mathcal{E}_i \] (where \( \mathcal{E}_i \) is defined in terms of the variables local to \( P_i \))

holds, without introducing centralization. Since there is no way to assure simultaneity, interval assertions are required:

**Definition:** Let \( t \leq \bar{t} \) be time instances and \( \mathcal{E} \) a global (local) predicate. Then \( \alpha = (t, \bar{t}, \mathcal{E}) \) is a global (local) interval assertion which is true if \( \mathcal{E} \) holds during the entire time interval \([t, \bar{t}]\) and false otherwise.

**Definition:** A set \( \mathcal{A} = \{\alpha_i = (t_, \bar{t}_i, \mathcal{E}_i) \mid 0 \leq i < n\} \) of local interval assertions covers the global interval assertion \( \alpha = (t, \bar{t}, \mathcal{E}) \) if:

1. \( t_0 \leq t, \bar{t}_i \leq \bar{t} \) for all \( 0 \leq i < n \)
2. \( \forall i \leq n \) \( \mathcal{E}_i \rightarrow \mathcal{E} \)

Thus, to show that a predicate \( \mathcal{B} \) is true sometimes it suffices to prove the existence of a global interval assertion \( \alpha = (t, \bar{t}, \mathcal{E}) \) which is covered by some set of true local interval assertions. Notice that if a global interval assertion \( \alpha = (t, \bar{t}, \mathcal{E}) \) is true then a covering set of true local interval assertions necessarily exists. Also note that interval assertions are not directly connected with verification of the program.

To implement the method of interval assertions, a sequence \( T_1, T_2, \ldots \)
of sets of time instances is dynamically constructed such that for each
1 \leq k, T^k = \{t^k_i \mid 0 \leq i < n\} and \(\max_{0 \leq i < n} t^k_i < \min_{0 \leq i < n} t^{k+1}_i\). At time \(t^{k+1}_i\) the i-th
process \(P_i\) makes a local test in order to verify the interval assertion
\(d^k = (t^k_i, t^{k+1}_i, B_i)\). The truth values of the \(d^k\)'s for the various i's
are accumulated in some process, say \(P_0\), which is thereby capable of de-
ducing that all the \(d^k\)'s are true. In that case, \(\{d^k \mid 0 \leq i < n\}\) is a set
of true local interval assertions which covers \(\max_{0 \leq i < n} t^k_i, \min_{0 \leq i < n} t^{k+1}_i, B\).

If some of the \(d^k\)'s are false then \(P_0\) distributes a message among the
processes to start a new set of tests.

Let us turn now to develop the distributed termination scheme in
which the interval assertions method is employed. Note that since global
stability is eventually reached, there exists a time instance t at which
\(B\) becomes true. Moreover, \(B\) may not become false after being true once.
Thus, if the above tests are added to the life cycle of each process, there
will be some k such that \(\{d^k \mid 0 \leq i < n\}\) will all be true and
thus global stability will be detected by \(P_0\). Care must be taken here so
as to assure that the processes do not devote all their time testing the
\(d^k\) rather than doing some useful activity.

Let us start with the given distributed program:

**Version 0:** \(P::[P_o] \ldots [P_{n-1}]\)

where for each \(0 \leq i < n\), \(P_i::[S_i]\) and \(P\) is assumed to have the two prop-
erties T1 and T2 stated in Section 1. Thus, when \(B_i\) holds \(P_i\) waits at the
top level loop, and \(S_i\) contains guards which wait for possible commu-
nication.
We wish to augment each $P_i$ so as to include a test for the value of an interval assertions concerning $B_i$ at time instances $t_i^k$, $k=1,2,...$. The results of the tests will be accumulated in $P_o$ which is taken as responsible for the detection of $B$. Once $B$ is detected, $P_o$ will initiate a wave of termination throughout all the $P_i$'s, whereby $P$ will terminate, as required.

Refinement 1: To implement the test of the local interval assertions, changes in the values of the $B_i$'s must be followed. The date of the last time instance in which $B_i$ became true could be used to decide whether it occurred in the time interval $[t_i^k, t_i^{k+1}]$ or earlier. However capturing the exact time instance is both virtually impossible and not required; it suffices to attribute a date to some later test for the value of $B_i$ in which $B_i$ is found to be true. To this end we introduce local Boolean variables $b_i$, initially false, in which changes in the values of the $B_i$'s are recorded. The program segments $S_i$ are modified so as to set $b_i$ to false in conjunction with every communication. (This is based on property T2 which assumes that if $P_i$ is locally stable then it can become active again only as a result of some communication.) Let $S'_i$ denote the modified program segments. We get:

**Version 1:**

$p_i:: b_i:: false$

```plaintext
[ S'_i

$B_i; \neg b_i \land b_i:: true.$

].
```
Refinement 2: Next a mechanism for accumulating the values of the \( b_i \)'s is to be developed. The accumulation process starts at \( P_0 \), propagates along the cycle and ends at \( P_0 \). When a process \( P_i \) receives the accumulating wave it can transmit it immediately. However, \( P_i \) may as well postpone the propagation until \( b_i \) becomes true. In this way there is no need to deal with Boolean messages but rather with a constant message (a pebble). A local Boolean variable \( send_i \) is used to indicate the reception and the transmission of the constant message. As a side effect we obtain an indulgent mode of communication \([FR]\) and thereby avoid freezing the basic activity of the algorithm in favor of control activity. When \( P_0 \) gets the pebble back, it has to initiate the additional termination wave. We shall ignore the details of this last wave, and let \( P_0 \) halt instead.

**Version 2:**

\[
P_i :: b_i = \text{false}; \quad send_i = \text{if } i=0 \text{ then true else false};
\]

\[
\quad S_i ^i
\]

\[
\begin{align*}
& b_i \rightarrow \neg b_i & b_i = \text{true} \\
& P_{i-1} \text{ ? pebble( )} & [ i=0 \rightarrow \text{Halt} \\
& \quad \text{if } i\neq 0 \rightarrow send_i := \text{true}] \\
& send_i ; b_i ; P_{i+1} \text{ ? pebble( )} & send_i := \text{false}
\end{align*}
\]

Here \( i+1 \) and \( i-1 \) and taken modulo \( n \). We let \( P_0 \) halt when the pebble returns to it, and do not deal here with the details of the final wave of termination.

If \( B_i \) is monotonic (that is, once it becomes true, it is not changed to false) then Version 2 is correct. However, if some of the \( B_i \)'s are not monotonic Version 2 may halt in a nonstable state. The reason is that
\( F_i \) may set \( b_i \) to true, transmit the pebble and then participate in a communication with some \( P_j \) at which the pebble has not yet arrived, thereby falsifying \( B_j \). To detect this type of "behind the back" communication, the method of interval assertions will be used. A more detailed information than that carried by the pebble must be propagated by each wave. Let us use the Boolean variable \( s_i \) to store the received message of the augmented wave, and let \( C_i \) denote the contribution to the accumulated information \( a_i \) of the process \( E_i \) to the accumulated information \( a \). The exact meaning of \( C_i \) will become clear in the next version.

Version 3: \( F_i : \ b_i := \text{false}; \ send_i := \text{false} \)

\[ \begin{array}{c}
S_i \\
L_i : B_i ; \neg b_i \\
P_{i-1} ? s_i \\
send_i \ L_i ; P_{i+1}(s_i, \neg C_i) \\
\end{array} \]

The program of \( F_i \) should be somewhat different. This will be pointed out later.

To use the method of interval assertions, dates must be attached to the changes in the value of \( B_i \). It suffices to attach a date to the assignment \( b_i := \text{true} \) of line \( L_i \). The variable \( d_i \) (initially 0) will be used as a clock in \( F_i \). Each time a message is sent, \( d_i \) will be increased by one. Another variable, \( d_i \) will record the value of \( d_i \) when the assignment at line \( L_i \) is performed. The resulting program (for all processes other than \( F_i \)) is:
Version 4: $P_i: b_i := \text{false}; \text{send}_i := \text{false}; d_i := 0_i$

\[ S_i \]

- $b_i := \neg b_i \land b_i := \text{true}; \text{db} := d_i$
- $P_{i-1} ? \sigma_i + \text{send}_i := \text{true}$
- $M_i := \text{send}_i; b_i := P_{i+1} (a_i | C_i) + \text{send}_i := \text{false}; d_i := d_i + 1$

The time instance in which the assignments at line $M_i$ are performed is to be considered as the $t^k_i$ of the interval assertions method. Notice that in that time instance $b_i$ necessarily holds: $b_i = \text{true}$ being one of the guards and the consistency with $b_i$ is assured by the structure of $S_i$. Thus the role of $C_i$ is to assure that a change in the value of $B_i$ has not occurred since $t^k_i$. To this end, we use:

$$C_i \equiv \text{db} < d_i$$

As for the structure of $P_o$: The first wave is sent when $b_i$ is true. When $P_o$ receives the wave back it checks the Boolean value arriving. If it is true then $P_o$ may halt (or, more precisely, initiate a terminating wave). The reason is that the set $\{(t^k_{i-1}, t^k_i, B_i)| 0 < i < n\}$ is a set of local interval assertions which are all true, and $\max t^k_{i-1} - \min t^k_i$. To see this latter fact notice that $P_o$ receives the $k-1$ wave before initiating a new wave of tests. The program for $P_o$ is therefore:
In [FR], a slight optimization in the implementation of the interval assertions method was achieved. The use of dates was avoided, and instead a flag (called advanced) was used, which was set in conjunction with each basic communication, and served as the basis of noticing a (potential) change in the value of $B$. 

```plaintext
Version 5: $P_0 \iff b_0 := \text{false}; \text{send}_0 := \text{true}; d_0 = 0; \text{d}_0 = 0$

$\text{[ } \begin{array}{l}
\s_0 \\
\text{P}_0 \not\iff b_0 + b_0 := \text{true}; \text{d}_0 = d_0 \\
\text{P}_{n-1} \not\iff \s_0 \implies \begin{array}{l}
\begin{array}{l}
\text{e}_0 := \text{halt}\end{array} \\
\begin{array}{l}
\text{e}_0 := \text{send}_0 := \text{true} \end{array}
\end{array} \\
\text{send}_0 \not\iff b_0; \text{P}_1 ((d_0 < d_0) \not\implies \text{send}_0 := \text{false}; \text{d}_0 = d_0 + 1)
\end{array} \text{ ]}
```
3. A CASE STUDY

We present an example, which both exemplifies the transformation and shows how a moderately tricky algorithm becomes almost trivial once the distributed termination aspect is isolated and left to the care of our transformation.

We did not present here some of our 'heavier' examples, in order not to detract too much of the attention to the details of the examples on account of the details of the distributed termination itself.

We consider a terminating distributed program, solving the 'Dutch national flag' problem [D]. Given are three processes R(ed), W(hite) and B(lue), arranged circularly:

Each process contains a loop colored red, white or blue.

pebbles having a "foreign" color, one at a time. Upon termination, each process must contain all and only pebbles of its own color.

In [D], three buffer processes are introduced, and a clever sending strategy is used, where each process first sends away pebbles of the color of the furthest process (in the cyclic order); also, dynamic channels
(obeying a usual block structure nesting discipline) and negations of i/o guards are introduced, all for the sake of having a terminating program.

If we ignore termination for a while and try to achieve the more modest goal of global stability when each process already contains all and only pebbles of its own color, we end up with the following program, using the same cyclical arrangement (but without dynamic channels or negation of i/o guards);

\[
\begin{align*}
R &:: \text{own, foreign := "initial values"; \{own contains red pebbles, and foreign white and blue pebbles\}} \\
&\quad \text{\[foreign\]} \quad o+x=:\text{any (foreign)}; W!x; foreign:=foreign-\{x\} \\
&\quad \quad B?y + \text{red(y)}; \text{own:=own U \{y\}} \\
&\quad \quad \quad \neg\text{red(y)}; foreign:=foreign'U \{y\} \\
&\end{align*}
\]

\[W\] and \[B\] are defined similarly. A similar version of the program, but using a lower level primitive of channel inspection, appears in Sintzoff [S2]. Note that \[foreign\] does not have to be kept as a union of \[w\] and \[b\] as in Dijkstra's solution. This partition will appear only in the proof that global stability is indeed achieved. In the proof, we index each local variable by the process name to which it belongs.

We define for each pebble in some process its distance from its home process. These distances will be used for a variant function to show the
absence of infinite computations. A similar proof appears in [D2].

\[
\text{Distance}_R(x) = \begin{cases} 
0 & x \in \text{own}_R \quad (\text{I.e. } x \text{ is red}) \\
1 & x \in \text{foreign}^R \land x \text{ is white} \\
2 & x \in \text{foreign}^R \land x \text{ is blue}
\end{cases}
\]

Distance_W and Distance_B are similarly defined.

Now, define

\[
D = \sum x \in \text{foreign}^R \text{Distance}_R(x) + \sum x \in \text{foreign}^W \text{Distance}_W(x) + \sum x \in \text{foreign}^B \text{Distance}_B(x)
\]

Each time a loop is traversed by a process (thereby, because of the I/O synchronization, another process does the same) a pebble changes place. There are two possibilities:

i. A pebble is moved from a foreign bag to an own bag, whereby its distance is not counted anymore in D;

or

ii. A pebble is moved from a foreign bag to another foreign bag, whereby its distance was reduced by 1.

In both cases, D decreases, and since it is non-negative, the absence of infinite computations has been established.

Also, \(D = 0\) implies \(|\text{foreign}| = 0\) in all three processes, thus each process contains all and only pebbles of its own color, and since the Boolean guard in each process is false, no process will try to output a pebble, and all processes are ready to input (in their y) a pebble. Thus, global stability has been achieved when \(D = 0\).
Hence, the transformation described above can be applied to obtain a terminating program to the same effect. Indeed, the transformed program appears to be more complicated than the one in [D]. However, a careful analysis of this specific program shows that the number of control cycles is constant, at most 2, independently of the number of pebbles involved. Whichever process is selected as designated (the root) may reach a locally stable state of \( |\text{foreign}| = 0 \) and initiate a control cycle, and possibly receive a foreign pebble afterwards. However, when the first wave passes the other two processes, it is delayed until each of them becomes locally stable, and one can verify that this time their job is done. Thus, the root may at most initiate a second control cycle, which will report global stability.
4. DISCUSSION

It is a nice property of our transformation, that it keeps the control overhead small whenever possible. Note, again, that the whole transformation may be made invisible to the programmer, which will have to deal with the simple program described above.

The price of these extra (not needed in principle) control communications is not prohibitive in many cases. One has to take into account that the dynamic creation and destruction of channels used in [6] is also equivalent to extra communications, as can be seen if we try to code them in [CSP].

As a concluding remark, we would like to draw attention to the fact that efficiency gained by having a (Hamiltonian) cycle can be gained at least partially for an arbitrary communication graph, in case the (R1) restriction (of not adding new channels for control communication) has to be observed.

Every connected graph can be covered by local cycles, to obtain a "generalized tree", whose "nodes" are such cycles. Pictorially, one would draw any graph in the following form:
The dotted edges are used for basic communications only, whereas
the solid edges are used also for control communication. The two
extreme covers are:

i. Each cycle degenerates to a (single) node; thus we get the
previously used spanning tree.

ii. All nodes participate in a (single) cycle; thus, we get the
Hamiltonian cycle discussed here.

Given such a cover one can combine the algorithms of [FR] and the
one presented in this paper to obtain a control communication scheme that
will act like in [FR] on solid edges not on (local) cycles, and will act
like suggested above within a cycle.

If k processes correspond to nodes located on cycles and n-k
processes correspond to nodes located outside cycles, then the complexity
of the combined communication is \( n + \alpha + b \), where

\[ n \leq \alpha = 2n - k \leq 2n \]

It must be noted that in general finding an optimal cover (with k max-
imal) is NP-complete, since it will find a Hamiltonian cycle if such is
present.

We believe that the method of interval assertions deserves more
study, and can be used to design a solution to other problems requiring
synchronization among disjoint processes. For example, to implement a
procedure using distributed processes, a similar problem arises, where
instead of terminating, the next call may be accepted.

Other variations on the underlying network topology could be tackled,
in a similar way.
REFERENCES


[S1]  M. Sintzoff: Three problems in the design of distributed programs, working note presented at IFIP W.G. 2.3, April 1978.