

Distributed Termination

NISSIM FRANCEZ

University of Southern California, Los Angeles

Discussed is a distributed system based on communication among disjoint processes, where each process is capable of achieving a post-condition of its local space in such a way that the conjunction of local post-conditions implies a global post-condition of the whole system. The system is then augmented with extra control communication in order to achieve distributed termination, without adding new channels of communication. The algorithm is applied to a problem of constructing a sorted partition.

Key Words and Phrases: concurrent programs, distributed processes, disjoint memories, communication, input-output, distributed termination

CR Categories: 4.32, 5.24

1. INTRODUCTION

Recently, Hoare designed a new programming language for concurrent programming called CSP [7], which differs significantly from previously suggested languages such as Concurrent Pascal [1] or Modula [10] in that processes are *disjoint*, i.e., share no kind of variables whatsoever. All communication among concurrent processes is by means of input-output, which also serves for synchronization. A similar kind of communication is employed by Milne and Milner [8], without suggesting any specific language for expressing it. Sintzoff [9] uses similar primitives with a different synchronization based on channel testing. Brinch Hansen [2] suggests procedure calls as a means of communication among otherwise disjoint processes. Thus the subject of *distributed programs* gains more and more interest, in concert with developments in microprocessor technology, the future implementation tool.

Some of the theoretical problems involved in the semantics of distributed programs are discussed in [8], as well as in [6], and we are currently working on proof rules for strong (total) correctness of such programs. One of those questions, the problem of *distributed termination*, is discussed in this paper, and an algorithm for achieving such termination is suggested.

The need for such an algorithm arises because, in general, termination is a property of the global state of a concurrent program. It may be hard to distribute

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

This work was supported in part by the National Science Foundation under Grant MCS 78-673.

A revision of this paper was presented at the International Symposium on Semantics of Concurrent Computations, Evian les Bains, France, July 2-4, 1979.

Author's present address: Department of Computer Science, Technion—Israel Institute of Technology, Haifa, Israel.

© 1980 ACM 0164-0925/80/0100-0042 \$00.75

ACM Transactions on Programming Languages and Systems, Vol. 2, No. 1, January 1980, Pages 42-55.

the decision to terminate to processes that are aware of their local state but have only limited information about the state of other processes with which they are connected along a channel of communication. On the other hand, it may be fairly easy to distribute a global post-condition into (a conjunction of) local post-conditions. We discuss an example, an algorithm for achieving a sorted partition into n subsets, which is a generalization of a problem solved by Dijkstra [5] for $n = 2$.

2. THE PROBLEM

Suppose we want to design a concurrent program P , which has to achieve upon termination a post-condition $B(\bar{y})$, where \bar{y} is the (global) state vector. Suppose also that it is possible to partition \bar{y} into $n(\geq 2)$ *disjoint* substates $\bar{y}_1, \dots, \bar{y}_n$, and to find n predicates (over those substates) $B_i(\bar{y}_i)$, $i = 1, \dots, n$, so that we have the following property.

Property 1: $(\bigwedge_{i=1,n} B_i(\bar{y}_i)) \supset B(\bar{y})$.

Finally, suppose we can design relatively easily n processes P_1, \dots, P_n , which have as their state vectors \bar{y}_i , respectively, and which, by means of some communications with each other, exchanging data, can respectively achieve a state $B_i(\bar{y}_i)$ after a finite amount of time. We call this communication the *basic communication*.

Then, the concurrent program $P :: [P_1 \parallel \dots \parallel P_n]$ would be a natural solution to the original problem, *provided* we can enforce termination as soon as *all* P_i 's achieve their favorable state satisfying B_i . Such states are called *final*. Partial correctness follows from Property 1. Each P_i is repetitive, of the form

$$\begin{array}{l} P_i = * [g_{i_1} \rightarrow S_{i_1} \\ \quad \square \\ \quad \vdots \\ \quad \square \\ \quad g_{i_{k_i}} \rightarrow S_{i_{k_i}} \\ \quad] \end{array}$$

(where g_{i_j} may involve basic communication).¹

When the stable state $\forall i. B_i(\bar{y}_i)$ is reached, all P_i 's are in their outer level, with no guard ready. We assume another natural property.

Property 2: No two processes in final states conduct basic communication.

Since there is no central control which can inspect all P_i 's from the outside and decide when to terminate, the P_i 's have to establish the required fact by means of some extra *control communication*. We call the problem of designing such control communication the problem of *distributed termination*.

Obviously, a solution in which each P_i terminates as soon as it finds its own

¹ In this notation, which follows Hoare [7], $*[\dots]$ is used, rather than the **do...od** of Dijkstra, to denote repetition until all guards are false.

$B_i(\bar{y}_i)$ true is not correct, since there may exist another P_j , which needs additional basic communication with P_i in order to establish the truth of $B_j(\bar{y}_j)$.

The required solution should be, of course, independent of the specific problem P is trying to solve. It should be a communication scheme that could be inserted into any partially correct solution as above with the slightest possible additions to the basic communication part (for interface with the control communication). The solution should, therefore, also be independent of the number of processes involved n and independent of the specific neighborhood relationships holding among the P_i 's (here "neighbors" means processes with which some P_i is connected by means of a communication channel).

Another important requirement from the solution is that *it does not add new neighborhood relationships*, or equivalently, new communication channels. This requirement is important for physical realizations of such a solution. Simpler solutions can be obtained if the requirement is deleted.

3. THE PATTERN OF DISTRIBUTED TERMINATION

Let $P :: [P_1 \parallel \dots \parallel P_n]$ be a concurrent program with processes P_i , $i = 1, \dots, n$. Let \bar{y} denote the global state of P , and \bar{y}_i , $i = 1, \dots, n$, the total state of P_i . Assume that all the processes are *disjoint* and are communicating in some way along channels.

For a process P_i , *endotermination* is termination depending only on the reachability of some final state, determined by some predicate $B_i(\bar{y}_i)$ over the initial local state. Endotermination is similar to the termination of a sequential program, e.g., as described in [3]. *Exotermination* is termination depending on the condition that every member of a prespecified set $T = \{P_{i_1}, \dots, P_{i_k}\}$, $k > 0$, has terminated.

T is called the *termination dependency set* (TDS), and $T = \emptyset$ by convention in the case of endotermination. Let TDS_i denote the termination dependency set of process P_i . If $P_j \in \text{TDS}_i$, P_i is connected to P_j . We avoid here discussion of how the TDS is specified and how the termination of its members is sensed, since there are language-dependent issues.

In general, the kind of termination a process exhibits may depend on its own initial state as well as on the initial state of its companions. If the processes contain also local (internal) nondeterminism [6], then the dependency is on each initial state *and* each computation! As an example, consider a process P_1 , which is ready to accept no more than m messages from its neighbor P_2 , where m is locally determined, and $\text{TDS}_1 = \{P_2\}$. If P_2 terminates before delivering m messages, P_1 will terminate in an exotermination. If P_2 is willing to send m or more messages, P_1 will terminate in an endotermination.²

The concepts of endonontermination and exonontermination are defined analogously. Thus, endonontermination means the unreachability of any final local state. Exonontermination means nontermination of at least one process from the TDS.

Some processes are always either endoterminating or endononterminating, or

² In the sequel, we assume that processes have *no* local nondeterminism in order to simplify the discussion.

are always either exoterminating or exononterminating. ("Always" here means for each tuple of initial states and each computation.) We call such processes *endoprocesses* and *exoprocesses*, respectively. It is a useful language feature to allow syntactically imposed restrictions on a process as to what kind of termination it should exhibit. In CSP [7], one can use a guarded command containing only input-output guards in a way which implies that the process is an exoprocess. Using a mixture of Boolean and input-output guards does not allow such a syntactical distinction.

For each program P as above, we now define its *communication graph* G_P . G_P contains one node for each P_i in P and an edge (P_i, P_j) for each pair of processes P_i, P_j such that a channel of communication exists between P_i and P_j . Note that channels of communication are directed and that the G_P is a directed graph that may contain both (P_i, P_j) and (P_j, P_i) . We assume that G_P has no self-loops, since no process communicates with itself. For convenience, we assume that G_P is weakly connected, i.e., that the underlying undirected graph is connected. Otherwise, one has to discuss properties of the weakly connected components. We also assume that the channels are not created and destroyed dynamically and that G_P can be determined syntactically from P . Various properties of P can be expressed in terms of G_P , which characterizes all potential communications. For example, if G_P is a tree, then P obviously cannot deadlock. In Dijkstra [4] some dynamic properties are expressed with respect to such a graph, again in a context of deadlock freedom characterization.

We are also interested in another graph T_P , derived from G_P and the various TDS's. T_P will reflect all the termination dependencies within P . The nodes of T_P are the same as those of G_P , one for each process $P_i \in P$. However, the edges of T_P are not in general syntactically determined, since they may depend on the initial states. Let $\bar{y} = (\bar{y}_1, \dots, \bar{y}_n)$ be a fixed initial state vector.

For each edge $(P_i, P_j) \in G_P$,

- (1) $(P_i, P_j) \in T_P$ iff $P_i \in \text{TDS}_j$;
- (2) $(P_j, P_i) \in T_P$ iff $P_j \in \text{TDS}_i$.

Thus, T_P contains some of the directed edges of G_P and/or their reversals. All nodes corresponding to endoterminating processes will be sources, having no incoming edges.

DISTRIBUTED TERMINATION PATTERN THEOREM. *P terminates for $(\bar{y}_1, \dots, \bar{y}_n)$ only if T_P is acyclic.*

PROOF. If T_P contains a cycle, then a deadlock situation occurs, since no process whose corresponding node lies on the cycle can possibly terminate. (Obviously all nodes on such a cycle correspond to processes which are exononterminating for \bar{y} .)

The meaning of this theorem is that whenever P terminates for \bar{y} , it has a partial order induced on its processes, which describes a *wave of termination*, where once endoterminating processes (for that \bar{y}) terminate, all processes whose TDS's contain only those endoterminating processes terminate, and so on.

The acyclicity of T_P is of course not sufficient, because of the possibility of dynamic deadlocks and infinite computations. However, even the necessary

condition gives one the general feeling of the way distributed termination is achieved.

Methodologically, it may be easier to impose such a partial ordering syntactically and let the required "termination wave" be apparent from the program text. A similar observation is mentioned by Dijkstra [5], who suggests an algorithm containing one (syntactically) endoterminating process (in our terms) and one (syntactically) exoterminating process. He indicates that an alternative design of the algorithm is possible, involving two endoterminating processes, which would use a different termination pattern but would be harder to verify.

Finally, consider a simple example, expressed by CSP [7] notation. Consider again the process P_1 mentioned above, which is ready to consume no more than m inputs and to terminate if its companion P_2 does so before m messages have been passed. Let P_2 be a process ready to produce no more than k outputs and terminate if its companion P_1 does so before k messages have been passed.

For brevity in the program, we omit variable declarations, as well as the portions of program which process input or generate output without further communication. We let $P :: [P_1 \parallel P_2]$, where³

$$P_1 :: a := 0; \quad *[a < m; P_2 ? x \rightarrow a := a + 1]$$

$$P_2 :: b := 0; \quad *[b < k; P_1 ! y \rightarrow b := b + 1].$$

For this program P we have

$$G_P = \overset{\circ}{P}_1 \overset{\circ}{\longleftarrow} \overset{\circ}{P}_2$$

reflecting the directed channel of communication between P_2 and P_1 , along which P_2 sends outputs to P_1 , and P_1 receives input from P_2 .

For T_P , we have three possible cases.

(1) $k = m$. T_P has no edges at all, since no termination dependencies exist.

$$T_P = \overset{\circ}{P}_1 \quad \overset{\circ}{P}_2$$

(2) $k < m$. In this case, P_2 is endoterminating and P_1 is exoterminating with $\text{TDS}_1 = \{P_2\}$.

$$T_P = \overset{\circ}{P}_1 \overset{\circ}{\longleftarrow} \overset{\circ}{P}_2$$

P_2 will induce termination on P_1 .

(3) $k > m$. This is the symmetric case, and $\text{TDS}_2 = \{P_1\}$.

$$T_P = \overset{\circ}{P}_1 \overset{\circ}{\longrightarrow} \overset{\circ}{P}_2.$$

T_P is acyclic in all three cases, as expected, since P terminates for every initial state (m, k) .

As an example of the insufficiency of the acyclicity of T_P , consider the following

³ In this notation $P_2 ? x$ means that a value is input from process P_2 and assigned to x ; $P_1 ! y$ means that the value of y is output to process P_1 .

program in CSP [7]:

$$P :: [P_1 :: P_2 ? x \parallel P_2 :: P_1 ? y \parallel P_3 :: *[P_1 ? z \rightarrow \text{skip} \square P_2 ? u \rightarrow \text{skip}]]$$

$$T_P = \begin{array}{cc} P_1 & P_2 \\ & \swarrow \quad \searrow \\ & P_3 \end{array} \quad \text{which is acyclic (syntactically!)}$$

However, P_1 and P_2 are engaged in a deadlock, each waiting for an input from the other, thus preventing P_3 from (exo)termination.

Note that in CSP [7] the TDS is implicit, dependencies being determined by means of input-output guards. Exotermination is expressed by making the whole program an input-output guarded loop. We shall extend this convention.

4. A SOLUTION

Our strategy is to arrange termination dependencies among P_1, \dots, P_n so that T_P is acyclic. We assume that in the original P_i 's no dependencies are specified. We want to designate an arbitrary P_{i_0} which will "collect" all the information about the rest of the P_i 's having reached a state satisfying $B_i(\bar{y}_i)$. Having reached the conclusion that *all* P_i 's are in a final state, P_{i_0} will terminate, thus initializing the termination wave, which will eventually reach all P_i 's.

The basic idea in the design of the algorithm is to find a spanning tree in G_P 's underlying undirected graph (thus not adding new channels!). The control communication has the following phases.

(1) When in a final state, the root process initiates a control wave to all its descendants in the spanning tree.

(2) The wave propagates through a node P_j as long as $B_j(\bar{y}_j)$ is true too. The passage of the control wave through a node "freezes" the basic communications of the node (except for communications introduced by this algorithm).

(3) Each node notifies its parent (in the tree) whether all nodes in its subtree have reached final state. If $B_j(\bar{y}_j)$ itself is false, there is no need to propagate the control wave, and an immediate negative notification can be delivered.

(4) If a positive answer (i.e., that all nodes are in a final state) reaches the root, it initiates the implicit termination wave by terminating itself. The TDS's will be induced by the spanning tree so that this wave will spread all the way to the leaves. Note that contrary to the control waves and answers which are part of the algorithm, the termination wave will be derived from the TDS's.

In the case of a negative answer (meaning that some nodes are not in a final state yet), an "unfreezing" wave is propagated, allowing the resumption of basic communication.

(5) One has to take care that at least one basic communication is performed between two consecutive control waves to eliminate the possibility of an endless control loop. Hence a fourth wave is initiated by the leaves and passes each node not yet in final state only after having performed at least one basic communication. When reaching the root, a new control cycle may start.

Let G_P be the communication graph and T_P^* be any spanning tree in the

underlying undirected graph of G_P . We now modify P_1, \dots, P_n to $\bar{P}_1, \dots, \bar{P}_n$. First, we define for each \bar{P}_i its TDS to be the singleton set containing its parent in T_P^* . The root of T_P^* is specified as an endoterminating process. We have thereby made the dependency graph T_P coincide with the spanning tree T_P^* , thus guaranteeing that it is acyclic, in accordance with the pattern theorem.

Next, we add to each P_i a control section C_i to be executed as a set of alternatives to the basic communications, and we add a small interface section to the basic communication part of P_i .

C_i will depend on the relative position of the node i (corresponding to P_i) in T_P^* . We distinguish among three cases: the root, an intermediate node, and a leaf.

In order to describe C_i , we use a liberal extension of the CSP [7] notation. Let $J = \{j_1, \dots, j_k\}$ be an index set.

$$\begin{aligned} \prod_{j \in J} S_j &\stackrel{\text{df}}{=} [S_{j_1} \| S_{j_2} \| \dots \| S_{j_k}], \\ \bigwedge_{j \in J} q_j &\stackrel{\text{df}}{=} q_{j_1} \& \dots \& q_{j_k} \quad (\text{conjunction}). \\ \square_{j \in J} q_j \rightarrow S_j &\stackrel{\text{df}}{=} q_{j_1} \rightarrow S_{j_1} \\ &\quad \square \\ &\quad \vdots \\ &\quad \square \\ &\quad q_{j_k} \rightarrow S_{j_k} \end{aligned}$$

For a node P_i in T_P^* , let $f(i)$ be the index of P_i 's parent and let Γ_i be the set of indices of P_i 's children. We assume that all the variables in C_i are new, not appearing in P_i (except the arguments of the process predicate B_i , which are not stated). Also, we assume that the main loop is exited once all processes *in the TDS* (addressed by some input-output guard in the loop) have terminated. This extends the CSP convention

Case 1. P_i is the root. $\{\text{initially, } newwave = \text{true}, \forall j: \text{ready}(j) = \text{false}\}.$

$$\begin{aligned} C_i :: B_i; newwave \rightarrow \prod_{j \in \Gamma_i} \bar{P}_j ! \text{ok}; \prod_{j \in \Gamma_i} \bar{P}_j ? a(j); \\ r := \bigwedge_{j \in \Gamma_i} a(j); \quad [r \rightarrow \text{halt} \\ \quad \square \\ \quad \sim r \rightarrow newwave := \text{false}; \\ \quad \prod_{j \in \Gamma_i} \bar{P}_j ! \text{resume} \\ \quad] \\ \square_{j \in \Gamma_i} \bar{P}_j ? \text{ready}(j) \rightarrow [\bigwedge_{j \in \Gamma_i} \text{ready}(j) \rightarrow newwave := \text{true}; \\ \quad \prod_{j \in \Gamma_i} \text{ready}(j) := \text{false} \\ \quad \square \\ \quad \sim \bigwedge_{j \in \Gamma_i} \text{ready}(j) \rightarrow \text{skip} \\ \quad] \end{aligned}$$

a is a Boolean array, while **ok** is a new (control) communication signal, of a type different from **boolean** and **integer** (to meet the matching of types required by CSP [7] for input-output).

Thus, whenever \bar{P}_i is in a B_i state, it may initiate a control cycle by sending a control signal **ok** to each of its children in T_P^* (in any order that the children are ready to accept it). Then it waits for each child to "answer" with a Boolean value. If all answers are true, \bar{P}_i halts. Otherwise, it sends each child a **resume** message (to be interpreted by the children as a permission to resume some basic communication, after being "frozen" by the **ok** question) and may itself resume some basic communication or try to initiate another control cycle, once a ready signal has arrived from each child. The meaning of $ready(j)$ is explained in the sequel. The Boolean *newwave* records the arrival of all $ready(j)$ messages.

Case 2. P_i is an intermediate node {initially, $cm = \text{true}$, $\forall j: ready(j) = \text{false}$, *advanced* = false}.

$$\begin{aligned}
 C_i :: & \bar{P}_{f(i)} ? \mathbf{ok} \rightarrow cm := \text{false}; \\
 & [\sim B_i \rightarrow \bar{P}_{f(i)} ! \text{false} \\
 & \quad \square \\
 & \quad B_i \rightarrow \prod_{j \in \Gamma_i} \bar{P}_j ! \mathbf{ok}; \prod_{j \in \Gamma_i} \bar{P}_j ? a(j); \\
 & \quad \quad r := \bigwedge_{j \in \Gamma_i} a(j); \bar{P}_{f(i)} ! r \\
 & \quad] \\
 & \square \\
 & \bar{P}_{f(i)} ? \mathbf{resume} \rightarrow cm := \text{true}; \text{advanced} := \text{false}; \prod_{j \in \Gamma_i} \bar{P}_j ! \mathbf{resume}; \\
 & \square \\
 & \square \bar{P}_j ? ready(j) \rightarrow \text{skip} \\
 & \quad j \in \Gamma_i \\
 & \square \\
 & (B_i \vee \text{advanced}) \ \& \ \bigwedge_{j \in \Gamma_i} ready(j); \bar{P}_{f(i)} ! \text{true} \rightarrow \prod_{j \in \Gamma_i} ready(j) := \text{false}
 \end{aligned}$$

The array a and the **ok** signal are as in Case 1. cm is a Boolean variable, whose interpretation is masking basic communications. The \bar{P}_i basic communication part is augmented with cm as a guard.

Thus as an alternative to its basic communication, \bar{P}_i may accept an **ok** signal from its parent as part of the control wave. Upon receiving such a signal, \bar{P}_i immediately falsifies the basic communication guard cm , which can be set again to true only after a **resume** input from its parent, thus freezing itself. Then, \bar{P}_i checks its local state. If it is not a B_i state, it immediately responds with a false to its parent, thereby breaking the control wave. Otherwise (in a B_i state) it behaves like the root, propagating the control wave, except that instead of using the value of r , the accumulated state of its subtree, it communicates r to its parent.

Another set of alternatives is to receive a $ready(j)$ (equal actually to true) from each child, which means a permission to initiate a new control cycle. This message is passed on to the parent, in case \bar{P}_i is in a B_i state, or some basic communication occurred, an occurrence recorded by the Boolean variable *advanced*.

Case 3. P_i is a leaf {initially, $cm = \text{true}$, $ready = \text{false}$ }

$$\begin{aligned}
 C_i :: & \bar{P}_{f(i)} ? \mathbf{ok} \quad \rightarrow cm := \text{false}; \bar{P}_{f(i)} ! B_i \\
 & \square \\
 & \bar{P}_{f(i)} ? \mathbf{resume} \rightarrow cm := \text{true}; advanced := \text{false}; ready := \text{true} \\
 & \square \\
 & ready \ \& \ (B_i \vee advanced); \bar{P}_{f(i)} ! \text{true} \rightarrow ready := \text{false}
 \end{aligned}$$

Within leaf processes, *ready* is a Boolean variable whose task is to insure that a ready signal to the parent is sent only once per control cycle.

In all three cases, we also augment the basic communication part of each P_i which is not the root (why?), with a statement $advanced := \text{true}$, recording the fact that some advance in basic communication did occur.

Thus, the overall operation of \bar{P} is as follows. Processes are engaged in basic communication as long as possible. Occasionally, the root chooses to initiate an **ok** message, to traverse the tree T_P^* , and to wait for a Boolean result r , which should be true only if $\forall i. B_i(\bar{y}_i) = \text{true}$ holds. Whoever receives this **ok** signal freezes its basic communication and either spreads the **ok** message down the tree or decides that its own state is *not* a B_i state. Eventually, each process delivers an answer r to its parent such that $r = \text{true}$ iff all the processes in its subtree (all frozen) are in a B_i state. Once the root receives its own r , it halts if r is true and otherwise sends a **resume** signal to “unfreeze” the whole tree. Once a process receives this message, it delivers it further down the tree and resumes basic communication. Each process not yet in a B_i state, after doing at least one basic communication, signals that another **ok** wave is possible. When this signal reaches the root, the whole control cycle may start again. This goes on until $\forall i. B_i(\bar{y}) = \text{true}$ is reached (this is assumed to occur!), which will cause the root to halt eventually, and then the required termination wave will spread down T_P^* until it reaches all processes of \bar{P} , since T_P^* is a spanning tree. Thus, distributed termination has been induced on the original P . We remind the reader that the spreading of the wave of termination follows from an extension of the language rules of input-output guards and the construction of T_P^* as a spanning tree. It is not part of the addition. The control communication is used only to determine when the root can terminate!

5. CORRECTNESS

We now want to prove that the preceding algorithm has the required properties, i.e., that the augmented program $\bar{P} = [\bar{P}_1 \parallel \dots \parallel \bar{P}_n]$ terminates, with $\forall i. B_i(\bar{y}_i) = \text{true}$, given that P is such that $\forall i. B_i(\bar{y}_i) = \text{true}$ eventually occurs. The proof is not intended to be formal program verification, but such a proof could be obtained from this sketch once the right proof rules are available.

By our construction, $T_{\bar{P}} = T_P^*$, and so it is acyclic. Since the root process is the only endoterminating process, we have to prove its termination. That proof is sufficient to establish overall termination because of the construction of $T_{\bar{P}}$ as a spanning tree over $\bar{P}_1, \dots, \bar{P}_n$ and the determination of the TDS's accordingly.

Claim 1. Each nonroot \bar{P}_i must eventually be ready for control (**ok**) communication with its parent. Otherwise, it will perform an indefinite number of basic

communications, which is impossible by assumption. Note that this alternative is not conditioned and therefore cannot be blocked. Note, also, that this control communication *may* take place earlier, before it becomes the *only* alternative, depending on the guard scheduling rules.

Claim 2. Whenever some nonroot \bar{P}_i receives the control signal **ok** from its parent, it will eventually be ready to respond with a Boolean r , satisfying

$$r = \begin{cases} \text{true,} & \text{iff for all } P_j \text{ in } P_i\text{'s subtree, } B_j(\bar{y}_j) \text{ holds;} \\ \text{false,} & \text{otherwise.} \end{cases}$$

This claim follows by induction on the height of \bar{P}_i in T_P^* . If \bar{P}_i is a leaf, then \bar{P}_i itself is the whole subtree, and the claim is obvious from the leaf program. Otherwise, if \bar{P}_i detects a non- B_i state and is ready to answer false, the claim is true immediately. If \bar{P}_i detects a B_i state, it will attempt to send **ok** control signals to all its children. By Claim 1 applied to all the children, all of them will eventually receive the **ok** signal, and by the induction hypothesis, since their height is smaller by 1, they will eventually respond with an answer r as above. The claim follows because the conjunction of such r 's has the same property when B_i is known to hold. Note that after the response with r , \bar{P}_i is again at the top level, but unable to perform basic communication, since $cm = \text{false}$, and the next communication will be a **resume** signal.

Claim 3. The state of each \bar{P}_i does not change between its response (r) to its parent and the input of a **resume** signal. Upon receiving **ok**, cm is set to false and thus disables any further basic communication, which might change its state. Only the input of the **resume** signal causes setting cm to true and allows resumption of basic communication.

Claim 4. (a) The root cannot initiate two consecutive control cycles unless some process has performed some basic communication. (b) After each control cycle, either the root terminates or another cycle will follow.

Claim 4(a) follows from the presence of the *newwave* guard, which is set to true only after all the children reported with $\text{ready}(j) = \text{true}$. Each such response of some P_i depends on receiving ready signals from its own children and reporting to the parent. This reporting cannot be blocked and has to occur eventually, again as the only alternative, because either B_i is true or it is false and then some basic communication has to occur and set *advanced* to true. For a leaf, *ready* is set to true upon receiving **resume** and does not depend on any input from other processes. Hence claim 4(b) follows also.

PROPOSITION. \bar{P}_{i_0} (i_0 is the index of the root) terminates, and upon termination $\forall i. B_i(\bar{y}_i)$ holds.

PROOF. By the same argument as in Claim 1 and since *newwave* is initially true, \bar{P}_{i_0} eventually reaches a state where it must send an **ok** signal to all its children. By Claim 1, each child will eventually receive this signal, and by Claim 2, each child will eventually respond with some a . Let $r = \bigwedge_{j \in \Gamma_{i_0}} a(j)$. If $r = \text{true}$, then P_{i_0} terminates, and by Claims 2 and 3, the property $\forall i. B_i(\bar{y}_i) = \text{true}$ follows.

If $r = \text{false}$, P_{i_0} will send **resume** signals to its children, etc. (this signal will be accepted, by the note after Claim 2) until every process can resume basic communication. By Claim 4, some basic communication will necessarily have been performed by some \bar{P}_j , before a new control cycle like this can be repeated. Thus, only a finite number of such control cycles is possible, and again \bar{P}_{i_0} will terminate.

A simple analysis of the suggested algorithm shows that in the best case, where \bar{P}_{i_0} first attempts a communication cycle only when $\forall i. B_i(\bar{y}_i) = \text{true}$ already holds, the number of control communications performed is $2(n - 1)$. Each \bar{P}_i receives once an **ok** signal and responds once with an r answer. In the worst case, a control communication can occur between any two basic communications, and hence the total number of communications is $4b(n - 1)$, where b is the number of basic communications. (The factor 4 is due to the additional unfreezing wave and *ready* response.) This number may grow very fast and is due to the nonmonotonic behavior of the P_i 's with respect to reaching a B_i state. The actual number of control communications will depend on the guard scheduling algorithm. Note that "best" and "worst" in this context refer to quantification over all possible schedulings, and not over initial states. Further complexity analysis is beyond the scope of this paper.

6. AN EXAMPLE: SORTED PARTITION

Let S be a nonempty set (without repetitions) of natural numbers, and let $S = S_1 + S_2 + \dots + S_n$ be a *disjoint* partition of S into $n \geq 2$ nonempty subsets. Also, let $m_i = |S_i|$ be the number of elements in S_i .

Consider the following post-condition $B(S_1, \dots, S_n)$:

$$B(S_1, \dots, S_n) \equiv \forall i, j (1 \leq i < j \leq n) \forall p, q (p \in S_i \ \& \ q \in S_j \supset p < q) \\ \& \ \forall i (1 \leq i \leq n) |S_i| = m_i,$$

i.e., the final state is such that each subset S_i has the same number of elements as it started with, and the partition is sorted in ascending order of the natural numbers.

The only basic communication allowed is sending or receiving a natural number.

This is a generalization of (a slight modification of) a program presented by Dijkstra [5] for $n = 2$, there called a sorting problem.

One can easily verify that an equivalent specification is given by

$$B(S_1, \dots, S_n) \equiv \forall i (1 \leq i < n) \forall p, q (p \in S_i \ \& \ q \in S_{i+1} \supset p < q) \\ \& \ \forall i (1 \leq i \leq n) |S_i| = m_i.$$

Furthermore, if we introduce the functions $\max(S_i)$ and $\min(S_i)$ with the usual meanings, this can be further transformed to

$$B(S_1, \dots, S_n) \equiv \forall i (1 \leq i < n) (\max(S_i) < \min(S_{i+1})) \\ \& \ \forall i (1 \leq i \leq n) |S_i| = m_i.$$

This last specification suggests a program organization in which each process (except P_1 and P_n) has two neighbors with which it communicates, i.e., inter-

changes natural numbers (members of the corresponding sets). For $1 < i < n$, the left neighbor of P_i is P_{i-1} and the right neighbor is P_{i+1} . The end processes in the line will have only one neighbor each.

Now we may introduce an additional variable: lin_i = last input from right neighbor, for $1 \leq i < n$. Then define for $1 \leq i < n$,

$$B_i(S_i, lin_i) \equiv \max(S_i) \leq lin_i \ \& \ |S_i| = m_i; \quad B_n \equiv \text{true}.$$

Notice that for $1 \leq i \leq n$, the last input of P_i from P_{i+1} = the last output of P_{i+1} to P_i , which is implied by the CSP semantics and the program below. With the help of that fact, one can verify that

$$(\bigwedge_{i=1,n} B_i(S_i, lin_i)) \supset B(S_1, \dots, S_n)$$

which fits our requirement as described above.

Next we have to find processes P_i , $i = 1, \dots, n$, which after a finite amount of basic communication reach their B_i states and which conform to Property 2 of Section 2 and to the linear arrangement of the neighborhood relation.

By generalizing Dijkstra's algorithm in [5], one obtains the following for $1 < i < n$.

```

Pi :: update; lin := - ∞;
    * [
        mx > lin; Pi+1 ! mx → Si := Si - {mx}; Pi+1 ? lin;
        □ Si := Si + {lin}; update
        Pi-1 ? l → Si := Si + {l}; update; Pi-1 ! mn;
        □ Si := Si - {mn}; update; Pi-1 ! mn
        □ Pi+1 ? lin → skip
    ]

```

where *update* is defined by ($mx := \max(S_i)$; $mn := \min(S_i)$). Each process P_i has a choice between two alternatives:

(1) If the largest element in the current value of S_i is larger than the last input, send it to the right neighbor, remove it from S_i , and include in S_i an element received from the right neighbor.

(2) Accept any number from the left neighbor, include it in S_i , then send back the smallest member of S_i and remove it from S_i .

(3) Accept a change in lin , after a change in the right neighbor's S .

In general, each process sends "large" numbers to the right, replacing them with "small" numbers, and similarly, receives large numbers from the left, replacing them by small numbers.

For the end processes we have the following.

```

P1 :: update; lin := - ∞;
    * [ mx > lin, P2 ! mx → ... (as before) ...
    □ Pn-1 ! mn
    ]
Pn :: update;
    * [ Pn-1 ? l → ... (as before) ...
    □ Pn-1 ! mn
    ]

```

By a slight generalization of Dijkstra's argument in [5], one indeed shows that after a finite amount of time, each process reaches a B_i state. Furthermore, each output guard is adjoined to a Boolean guard implying $\sim B_i$, and Property 2 in Section 2 also holds.

Note again the nonmonotonic behavior of P_i with respect to B_i . It may be possible that some P_i is in its B_i state, i.e., all its elements are smaller than its right neighbor's and larger than its left neighbor's. However, the right neighbor P_{i+1} , for example, may exchange a number with its right neighbor P_{i+2} and receive an element smaller than $\max(S_{i-1})$. When this element is passed to P_i , B_i is no longer true. The eventual stability is shown as in bubble sort, where the "big" elements float to the right and the "small" ones float to the left.

7. CONCLUSION

We have formulated an algorithm that achieves a joint decision of a group of communicating processes to terminate, where each of them is directly aware only of its own local state. The algorithm is based on a general property of disjoint processes, in which termination can be either achieved directly or induced by other terminating processes. We have shown how to solve a problem of sorted partition using the algorithm.

We feel that this kind of situation will occur often in various applications of distributed programming. Recently, we learned that in an unpublished manuscript, Sintzoff dealt with a similar question and suggested a circular arrangement of the P_i 's instead of our spanning tree. He is not concerned with the problem of avoiding new channels.

Currently a research project (jointly with W.P. de Roever) is attempting to construct a formal system in which the formal counterpart of the sketch proof presented here can be formulated.

ACKNOWLEDGMENTS

Zami Ben-Chorin suggested two-way control communication. Although his solution had some weakness, an improvement led to the current scheme. Conversations with Jorgen Staunstrup helped to clarify the general concept of distributed termination.

Many thanks to Robin Milner, who detected a deadlock possibility in an early version and who helped to improve the presentation, as did an anonymous referee.

REFERENCES

1. BRINCH HANSEN, P. *The Architecture of Concurrent Programming*. Prentice-Hall, Englewood Cliffs, N.J., 1977.
2. BRINCH HANSEN, P. Distributed processes—A concurrent programming concept. *Commun. ACM* 21, 11 (Nov. 1978), 934–941.
3. DE BAKKER, J.W. Semantics and termination of nondeterministic recursive programs. Proc. 4th Conf. on Automata, Languages, and Programming, 1976.
4. DIJKSTRA, E.W. A class of simple communication patterns. EDW-643, 1978.
5. DIJKSTRA, E.W. A correctness proof for communicating processes—A small exercise. EDW-607, 1977.

6. FRANCEZ, N., HOARE, C.A.R., LEHMANN, J.D., DE ROEVER, W.P. Semantics of nondeterminism, concurrency, and communication. To appear in *J. Comput. Syst. Sci.*
7. HOARE, C.A.R. Communicating sequential processes. *Commun. ACM* 21, 8 (Aug. 1978), 666–677.
8. MILNE, G., AND MILNER, R. Concurrent processes and their syntax. *J. ACM* 26, (April 1979), 302–321.
9. SINTZOFF, M. On language design for program construction. Centre de Recherche en Informatique, Jan. 1978.
10. WIRTH, N. Modula—A programming language for modular programming. *Softw. Pract. Experi.* 7, 2 (March 1977).

Received July 1978; revised August 1979