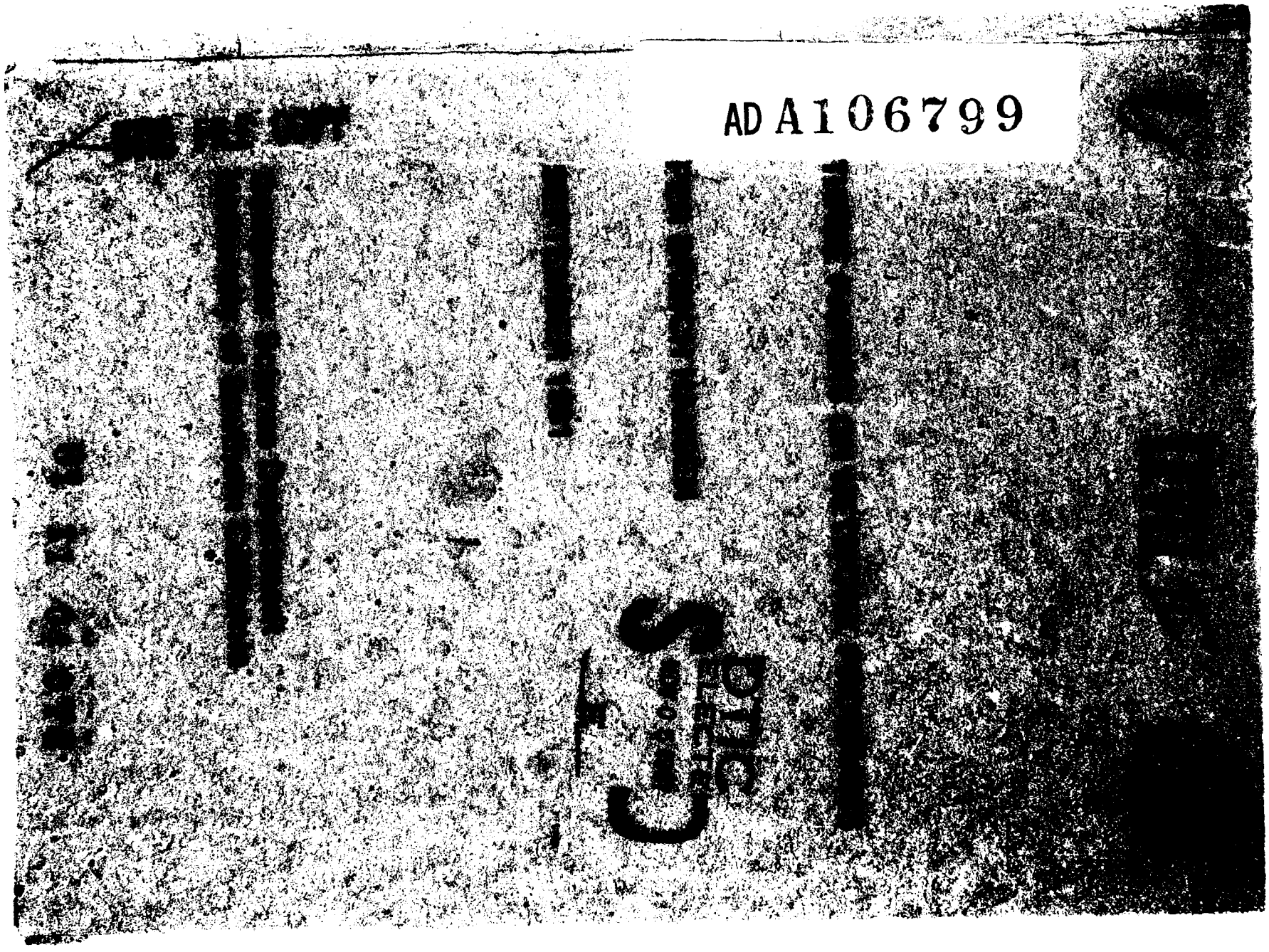
AD-A106 799 UNIVERSITY OF SOUTHERN CALIFORNIA LOS ANGELES DEPT OF--ETC F/G 9/2
THREE PAPERS ON NETWORK PROGRAMS.(U)
SEP 81 P BRINCH HANSEN N00014-77-C-0714
UNCLASSIFIED NL

END
DATE FILMED
12-81
DTIC



SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A106799	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) THREE PAPERS ON NETWORK PROGRAMS.	5. TYPE OF REPORT & PERIOD COVERED Technical Report
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Per Brinch Hansen	8. CONTRACT OR GRANT NUMBER(s) N00014-77-C-0714
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Science Department University of Southern California Los Angeles, California 90007	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS NR 048-647
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research Arlington, Virginia 22217	12. REPORT DATE Sep[redacted]81
	13. NUMBER OF PAGES 90
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report) Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE Not applicable

16. DISTRIBUTION STATEMENT (of this Report)

Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

distributed computing, computer networks, programming languages, program correctness, deadlock prevention, termination

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

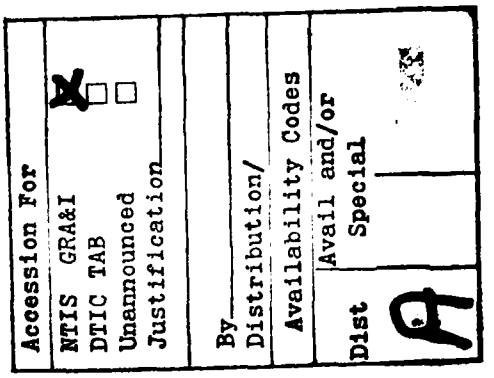
This report contains three papers on distributed computing entitled:

Basic concepts of network programs,
Language notation for network programs,
Proof rules for network programs,

Together these papers outline a systematic method for the programming of computer networks.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)



THREE PAPERS ON NETWORK PROGRAMS

Per Brinch Hansen

Computer Science Department
University of Southern California
Los Angeles, California 90007, U.S.A.

September 1981

Summary

This report contains three papers on distributed computing entitled

Basic concepts of network programs

Language notation for network programs

Proof rules for network programs

Together, these papers outline a systematic method for the programming of a class of computer networks known as coincidence networks.

This research was supported by the Office of Naval Research under Contract NR 048-647.

BASIC CONCEPTS OF NETWORK PROGRAMS

Per Brinch Hansen

Computer Science Department
University of Southern California
Los Angeles, California 90007, U.S.A.

August 1981
Revised September 1981

Summary

This paper defines a class of computer networks known as coincidence networks. The nodes of these networks are assumed to be responsive to their neighbors in a well-defined sense. Coincidence networks are shown to be free of local deadlocks if they are acyclic, or if they communicate according to fixed or cyclic priority schemes. Such networks will always terminate, if they only process data sequences of finite length. If the networks are fair, none of their nodes can remain blocked forever.

This research was supported by the Office of Naval Research under Contract NR 048-647.

1. Introduction

This is the first of three papers on distributed computing.

The present paper defines a class of computer networks known as coincidence networks and gives sufficient conditions for the prevention of local deadlocks and the termination of such networks.

The second paper introduces a formal notation in which programs for such networks can be written [1]. Efficient implementations of these concepts are suggested for existing computers.

The third paper presents proof rules for establishing the partial correctness of network programs written in the proposed notation [2].

Together these papers outline a systematic method for the programming of coincidence networks.

The original inspiration came from Hoare's paper on communicating sequential processes [3]. The key idea of the present work is that all reasoning about the structure of networks, their function and termination is expressed in terms of the data sequences exchanged by the nodes of a network. In the CSP notation, data sequences cannot be declared as named objects. Consequently, assertions about them cannot be expressed in the notation as it stands.

2. Coincidence networks

A coincidence network consists of a finite number of sequential machines, called nodes, and a finite number of data channels, called links. Each link connects exactly two nodes and enables the nodes to exchange a data sequence.

Figure 1 shows an example of a network. The nodes and links are represented by dots and lines.

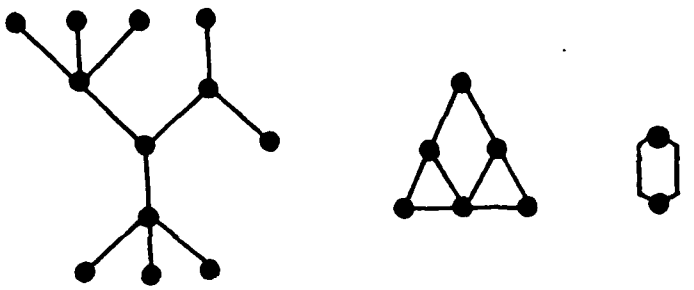


Fig. 1. A network

From this figure several observations can be made:

(1) A network may consist of several unconnected pieces.

(2) Some of the links may form cycles.

(3) Two nodes may be directly connected to each other by more than one link.

The transfer of a data value from one node to another is called a communication. It takes place when one of the nodes is ready to output a data value through a link and

the other node is ready to input a data value from the same link. Two nodes that are able to communicate are called neighboring nodes.

The coincidence of input and output is achieved by delaying a node that is ready to communicate with one of its neighbors until both of them are ready to communicate with each other. A delayed node is said to be blocked.

A link can transmit data in either direction between two nodes. An input operation performed by one of the nodes must, however, be matched by an output operation performed by the other node on the same link.

If two nodes simultaneously attempt to input (or output) through the same link, then both nodes (and the whole network) will fail.

If a link is used to transmit data in one direction only, it can be declared as an output link of one node and as an input link of another node. A compilation check can then ensure that the two nodes always perform matching input/output operations on the link.

From now on, we will assume that neighboring nodes always perform matching input/output operations. (If they fail to do so, the network must detect the failure and stop.)

The method for reasoning about networks will focus on the entire sequence of data values that has been transmitted through a link, independent of the direction in which each value has been sent.

3. Termination

The nodes and links of a network are created simultaneously.

In a temporary network, each node performs a finite sequence of operations and ceases to exist. The nodes (and the network) are then said to have terminated.

In a permanent network, some nodes perform endless cycles of operations and never terminate.

The data sequences may be unbounded in a permanent network, but can only be of finite length in a temporary network.

When two nodes have exchanged all the data values of a finite sequence through a link, one of the nodes must close the link, which then ceases to exist.

Before a node attempts to input a data value from a link, it may evaluate a boolean function named more to determine whether or not another data value will be output by the neighbor attached to the same link. The evaluation takes place either when the neighbor is ready to output to the link (in which case, more is true) or when the link has been closed (in which case, more is false).

From now on, the word network will be used to denote a coincidence network in which all the nodes are responsive in the following sense:

(1) A responsive node continues to communicate with some of its neighbors at a finite rate (> 0) as long as they are reachable and the node is not blocked forever.

(2) A responsive node terminates if, and only if, none of its neighbors are reachable.

Two neighbors are said to be reachable to each other as long as they can communicate through at least one link that is not closed.

Lemma 1: A network with two or more nodes remains a network when one of its nodes terminates.

Proof: When a node terminates, all its links have already been closed. The removal of the node and its links leaves a structure consisting of a finite number of nodes and links. As long as a link exists, none of the nodes attached to it will be removed. So each of the remaining links still connects exactly two existing nodes. Consequently, the structure is still a network.

When all the nodes of a network have terminated, all the nodes and links have ceased to exist.

It may seem more natural to only consider networks in which all the nodes are connected by some paths. But, since the termination of nodes eventually may break a network into disconnected pieces, it is easier to permit any network to be like that to begin with. Otherwise, Lemma 1 is invalid.

A network can only be expected to terminate if each of

its nodes operates on sequences of finite length. This restriction is captured by the following definition:

Limited network: A network in which each link only transmits a data sequence of finite length.

In a permanent network, a node only serves a useful purpose if its neighbors continue to pay attention to it. Otherwise, the node can remain blocked forever. This notion of fairness is defined as follows:

Fair node: A responsive node that continues to communicate with each of its neighbors at a finite rate (> 0) as long as they are reachable and the node is not blocked forever.

Fair network: A network in which each node is fair.

4. Local deadlocks

A path in a network is a sequence of distinct links L1, L2, ... , Lk which connect a sequence of neighbors n1, n2, ... , nk+1 as shown in Fig. 2.

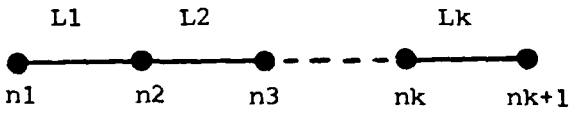


Fig. 2. A path

A cycle is a path that begins and ends at the same node

(see Fig. 3).

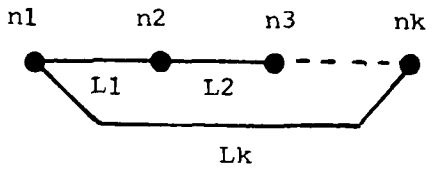


Fig. 3. A cycle

A local deadlock is a situation in which each of the nodes in a cycle is ready to communicate with a node that follows it in the cycle, but none of them are ready to communicate with each other. Consequently, the nodes will remain blocked forever.

A network is totally deadlocked when all its nodes are blocked forever.

In the following, we will only discuss networks in which local deadlocks cannot occur.

Lemma 2: A network that is free of local deadlocks is also free of total deadlocks.

Proof: Consider a node n1 that is waiting to communicate with a neighbor n2. In a total deadlock, n2 cannot be waiting for n1. For, in that case, a communication would take place between them. So n2 must be waiting to communicate with another neighbor n3, and so forth. Consequently, we end up with a sequence of distinct

neighbors n1, n2, ... , nk each waiting to communicate with the next one. Since the number of nodes in the network is finite, so is the sequence of blocked nodes. One of the nodes nk is therefore forced to wait for one of the previous nodes in the sequence in order for them to remain blocked. But this contradicts the assumption that the network is free of local deadlocks. Consequently, a total deadlock cannot occur under that assumption.

Theorem 1: If a limited network is free of local deadlocks then it terminates.

Proof: If there are no local deadlocks, there are no total deadlocks either (Lemma 2). So, among the nodes that can reach one another, there is always at least one pair that is able to communicate. The network will therefore continue to transmit some data elements at a finite rate (> 0). Since the number of links is finite, and since each of them must transfer a sequence of finite length only, the network will eventually reach the end of all the sequences and close them. At that point, the network terminates (because the nodes are responsive).

Theorem 2: If a fair network is free of local deadlocks then none of its nodes can be blocked forever.

Proof: Assume that a node n1 is waiting forever to communicate with a neighbor n2. Since n2 is fair, it can only be prevented from eventually communicating with n1 if it ends up waiting forever to communicate with another

neighbor n3, and so forth. So, again we end up with a cycle of blocked processes, which contradicts the assumption that local deadlocks do not occur. Consequently, a node cannot be blocked forever.

5. Acyclic networks

A network that contains no cycles is called acyclic.

Figure 4 shows an example of an acyclic network consisting of two trees.

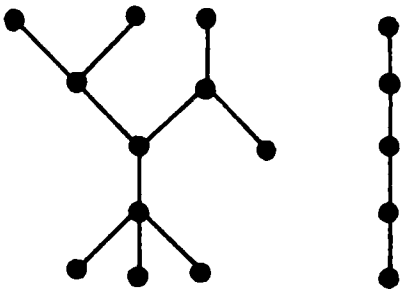


Fig. 4. An acyclic network

Lemma 3: An acyclic network remains acyclic when one of its nodes terminates.

Proof: obvious. (If a network has no cycles to begin with, the removal of a node and the links attached to it cannot create new cycles.)

Theorem 3: An acyclic network is free of local deadlocks.

Proof: In a network without cycles a local deadlock can (per definition) not occur.

When this result is combined with theorems 1 and 2, we obtain:

Theorem 4: If an acyclic network is limited then it terminates.

Theorem 5: If an acyclic network is fair then none of its nodes can be blocked forever.

6. Priority networks

The only cyclic networks we will consider are known as priority networks. A priority network is a network constrained as follows:

(1) Each link L is assigned a number #L, called the priority of L.

(2) For each node, the links attached to it have distinct priorities.

(3) Among the links attached to it, each node will always select the one that has the smallest number and attempt to communicate through that link within a finite time.

(4) After a communication through a link, the priority of the link may be changed as long as (2) is satisfied.

Figure 5 shows an example of a priority network.

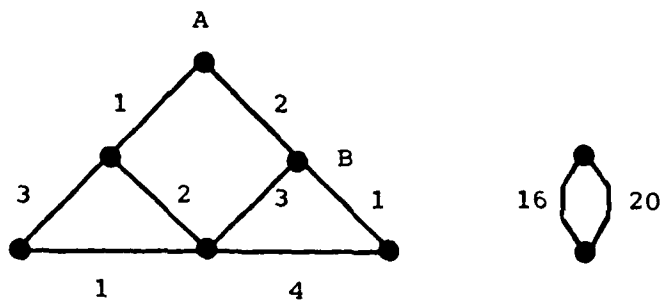


Fig. 5. A priority network

Lemma 4: A priority network remains a priority network when one of its nodes terminates.

Proof: obvious.

Theorem 6: A priority network is free of local deadlocks.

Proof: Assume that a priority network includes a cycle of nodes n1, n2, ... , nk (see Fig. 3) that are blocked forever. In that case,

n1 is waiting for n2 because #Lk > #L1
n2 is waiting for n3 because #L1 > #L2
...
nk is waiting for n1 because #Lk-1 > #Lk

So, we have #Lk > #L1 > ... #Lk-1 > #Lk, or #Lk > #Lk, which is a contradiction. Consequently, local deadlocks are impossible.

When this theorem is combined with theorems 1 and 2, we obtain the following results:

Theorem 7: If a priority network is limited then it terminates.

Theorem 8: If a priority network is fair then none of its nodes can be blocked forever.

In order for a priority network to be fair, the link priorities must change in time; otherwise, each node would continue to communicate through the same link (as long as it remains open).

As an example of a fair priority network, we will consider a clockwork net. This is a network in which the links of each node have fixed numeric indices and, in which each node communicates with its reachable neighbors in a fixed cyclical order. In each cycle, a node communicates through its links in the increasing order determined by their indices.

The network shown in Fig. 5 behaves like a clockwork net, if the node A repeats the communication cycle 1, 2, ... while the node B follows the pattern 1, 2, 3, ... , and so on.

A clockwork net C behaves exactly like a fair priority network C' defined as follows:

(1) C' consists of the same nodes and links as C. The initial priorities of the links in C' are equal to the indices of the corresponding links in C.

(2) Since the links of each node in C have distinct indices, the links of each node in C' have distinct initial priorities.

(3) Among the links attached to it, each node in C' will attempt to communicate through the one that has the highest priority (i.e. smallest number).

(4) After each communication through a link in C', the priority of the link is increased by a network constant which is greater than the difference between the largest and smallest indices used anywhere in C. Since the priorities of each node in C' are distinct to begin with, they will remain distinct after each communication.

Each node in C' will obviously communicate with its neighbors in exactly the same cyclical order as the corresponding node in C. Consequently, if C' is free of local deadlocks and permanent blocking, then so is C. And, if C' terminates, then C also terminates.

In short, theorems 6, 7, and 8 also apply to clockwork nets.

A moments reflection will make it clear that these theorems also apply to a wider class of periodic networks in which each link has one or more indices, as long as the same index never is associated with two or more links attached to the same node.

Figure 6 shows a periodic network in which the nodes go through the following cycles:

A: 1, 2, 3, 4, 5, ...

B: 1, 5, 9, ...

C: 2, 4, 8, 11, ...

D: 3, 7, 10, ...

E: 7, 8, 9, 10, 11, ...

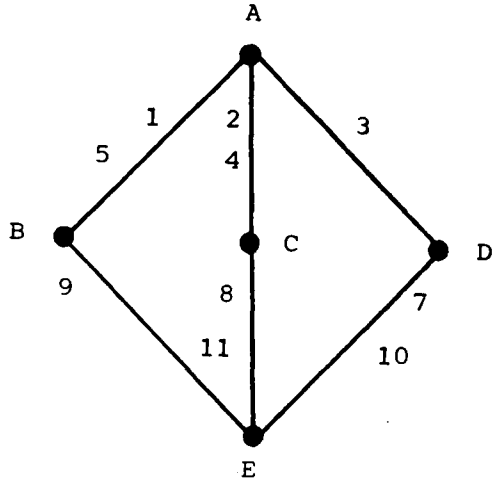


Fig. 6. A periodic network

More precisely, a <u>periodic network</u> is a network in which

(1) Each link is assigned a non-empty set of distinct numbers, called the <u>priority set</u> of the link.

(2) For each node, the priority sets of the links attached to the node are disjoint.

(3) Each node communicates cyclically with its neighbors

in the numerical order determined by the union of the priority sets of the links attached to the node.

7. Final remarks

This paper has defined a class of computer networks, called coincidence networks. In a coincidence network, each node continues to communicate with some of its neighbors as long as they are reachable. Coincidence networks are free of local deadlocks if they are acyclic, or if they communicate according to fixed or cyclic priority schemes. Such networks will always terminate, if they only process sequences of finite length. If the networks are fair, none of their nodes will remain blocked forever.

The limitations of the present work are three fold:

(1) The paper only discusses a particular kind of cyclic network (priority networks).

(2) The paper does not consider non-deterministic networks in which some nodes may be ready to communicate with any one of several neighbors as soon as one of the neighbors is ready to do so.

(3) The paper does not consider networks in which additional nodes and links can be created dynamically.

The programming notation introduced in [2] can describe more general networks that are cyclic, non-deterministic, or dynamic in the sense defined in (1), (2), and (3).

Acknowledgement

The comments of David Jefferson, Peter Lyngbaek and Tom Mowbray have been extremely helpful.

References

1. Brinch Hansen, P.: Language notation for network programs. Computer Science Dept., Univ. of Southern California, Los Angeles, CA, Sep. 1981.

2. Brinch Hansen, P.: Proof rules for network programs. Computer Science Dept., Univ. of Southern California, Los Angeles, CA, Sep. 1981.

3. Hoare, C. A. R.: Communicating, sequential processes, CACM 21, 666-677, (Aug. 1978).

LANGUAGE NOTATION FOR NETWORK PROGRAMS

Per Brinch Hansen

Computer Science Department
University of Southern California
Los Angeles, California 90007, U.S.A.

August 1981
Revised September 1981

Abstract

This paper introduces a set of programming concepts for computer networks in which all communication takes place by means of typed data sequences. These concepts are illustrated by a variety of programming examples using a notation that is suitable for a programming language. An implementation is described for existing computers.

Index Terms - programming languages, distributed computing, computer networks, input/output, concurrency

This research was supported by the Office of Naval Research under Contract NR 048-647.

INTRODUCTION

This paper develops a set of programming concepts for a class of computer networks known as coincidence networks [1]. A coincidence network consists of a finite number of nodes and links. Each node is either a sequential machine or a network composed of other nodes and links. Each link is a data channel that connects exactly two nodes and enables them to exchange a data sequence. The output of a data value by one of the nodes coincides with the input of the same value by the other node. The nodes and links of a network are created simultaneously. When a node has accepted all its input and has produced all its output, it terminates. A network terminates when all its nodes have terminated.

Coincidence networks borrow two key ideas from Hoare's communicating, sequential processes [2]:

(1) The notation for input/output operations.

(2) The idea of a repetitive input statement controlled both by the availability of input data and by the values of boolean expressions.

Apart from that, the ideas presented here are very different from CSP:

(1) In the CSP notation, each communication appears as an isolated event that has no apparent relation to other communications. The entire program text of a process must

be read to determine the set of processes with which a given process communicates.

The language notation for coincidence networks clearly groups messages into distinct data sequences. The data sequences that connect a node to other nodes are explicitly declared in a parameter list associated with the node.

(2) In CSP, a process may send messages of different types to another process.

In this paper, each data sequence consists of elements of the same type. This restriction can be removed by an obvious extension of the notation.

(3) In CSP, an input loop performed by a process terminates when the producers of the input cease to exist. A recent paper argues that this form of termination complicates the semantics and suggests instead that input loops should terminate in the traditional manner when the values of boolean expressions are false [3].

The input loops used in coincidence networks simply terminate when the nodes reach the end of their input sequences.

(4) The original CSP notation is a concise mathematical notation, but is not suitable for a programming language as it stands: The control statements do not begin with unique symbols, and the type compatibility of operands must be checked during execution.

The notation for coincidence networks is based on the

Pascal-like notation of the programming language Edison [4]. The syntax and semantics of network programs expressed in this notation can be checked systematically during compilation.

(5) The CSP paper does not discuss the problems of implementing input/output operations efficiently.

This paper proposes a straightforward implementation of the proposed programming concepts on existing computers.

(6) Proof rules for CSP appeared in a recent paper [3].

Proof rules for establishing the partial correctness of network programs are defined in [5]. Sufficient conditions for the absence of local deadlocks and the termination of network programs are presented in [1].

The basic language concepts proposed for coincidence networks are surprisingly simple and are well-known from sequential programming languages that include sequential files as data types.

A variety of programming examples will be used to motivate a gradual extension of the basic concepts to their most general (and most complicated) form.

SEQUENCE TYPES

Our starting point is the concept of a data sequence. A type declaration of the form

```
seq T (E)
```

introduces a name T to denote sequences consisting of elements of some known type E. The element type E can either be an elementary data type (such as integer, boolean, or character) or a record, array, or set type. Sequences of other sequences cannot be defined.

In the programming examples, we will use sequences of these types:

```
seq numbers (int)
seq signals (bool)
seq string (char)
seq lines (line)
```

A declaration of the form

```
var x : T
```

introduces a variable x of a sequence type T, for example

```
var stream: numbers
```

Sequences may be combined into arrays by means of a type declaration, such as

array group [1:n] (string)

The declaration

var chain: group

introduces a variable named chain of type group. It consists of n indexed variables denoted

chain[1], chain[2], ... , chain[n]

Each of the indexed variables is a sequence of type string. Sequences may also be combined into records.

INPUT/OUTPUT STATEMENTS

In a network program, each link is described as a sequence variable x. The variable must be initialized by performing an open operation

open(x)

before it can be used to transmit data between two nodes.

A transfer of a data value from one node to another is called a communication. It takes place when one of the nodes is ready to output a data value on a link and the

other node is ready to input a data value from the same link. A node that is ready to communicate through a link is delayed (or "blocked") until another node is ready to communicate through the same link. Two nodes that are able to communicate are called neighboring nodes.

An output statement

x!e

describes the output of a data value to a sequence x. The value is obtained by evaluating an expression e that must be of the same type as the elements of x.

An input statement

x?v

describes the input of a data value from a sequence x. The value is assigned to a variable v that must be of the same type as the elements of x.

Since the input and output of the same data value coincide in time, the net effect of a communication is to perform the assignment

v := e

A data sequence can be used to transmit data in either

direction between two nodes. An input operation performed by one of the nodes must, however, be matched by an output operation performed by the other node on the same sequence. If two nodes simultaneously attempt to input (or output) from the same sequence, both nodes (and the whole network) will fail.

If a sequence is used to transmit data in one direction only, it can be declared as an output sequence of one of the nodes and as an input sequence of the other node. A compilation check can then ensure that the two nodes always perform matching input/output operations on the sequence. To simplify the presentation, all sequences described in this paper will be of this kind.

If two nodes exchange a sequence x of finite length, one of the nodes must perform a close operation

close(x)

when the last element of the sequence has been output.

When a sequence is closed, further input/output operations on the sequence are undefined. When all the sequences that connect two neighboring nodes are closed, the neighbors are said to be unreachable to each other.

A node may evaluate a boolean function named more

more(x)

to determine whether or not another data value will be output to the sequence x. The evaluation is delayed until another node is ready to output to the sequence (in which case, more is true), or until the sequence has been closed (in which case, more is false).

Notice, that the function more does not test the present availability of a data element in the sequence. It is a time-independent function that establishes whether the entire sequence has been output.

The only operations defined on sequences are the ones described so far: open, close, input, output, and more. A repetitive input statement will be added later but will be defined in terms of the five elementary operations on sequences.

NODE PROCEDURES

The actions performed by a node are described by a procedure of the form

```
node Procedure name ( Parameter list)
Declarations
begin Statement list end
```

The parameter list describes parameters of two kinds:

(1) The constant parameters denote fixed values that are computed when the procedure is called. These parameters cannot contain sequences as components.

(2) The input/output parameters denote distinct sequences that are selected when the procedure is called.

The possible operations on an input/output parameter may be restricted to more and input (or to output and close) by prefixing the parameter declaration with the word in (or out).

A node procedure may include declarations of local entities which are either constants, data types, variables, or procedures. A node procedure acts as a block. The named entities described by the parameter list and the declarations are only known within the procedure body.

A node procedure can use constants, data types, and node procedures that are declared in surrounding blocks. But it cannot use variables and general procedures declared in the surrounding blocks.

The above restrictions ensure that the only global variables used by a node procedure are the sequences that are bound to its input/output parameters when the procedure is called.

A node procedure includes a statement list to be executed when the procedure is called.

The first programming example is a procedure executed by a node that outputs the sequence of natural numbers 1, 2,

... , n and terminates.

```
node counter(n: int; out x: numbers)
var i: int
begin i := 0;
  while i < n do
    i := i + 1; x!i
  end;
  close(x)
end
```

The next procedure describes a node that inputs a sequence of numbers, adds them, and outputs the sum.

```
node adder(in x: numbers; out y: numbers)
var a, b: int
begin a := 0;
  while more(x) do
    x?b; a := a + b
  end;
  y!a; close(y)
end
```

Both of these nodes are responsive to their neighbors in the sense defined in [1]:

(1) A responsive node continues to communicate with some

of its neighbors at a finite rate (> 0) as long as they are reachable and the node is not blocked forever.

(2) A responsive node terminates if, and only if, none of its neighbors are reachable.

In a coincidence network, all nodes are assumed to be responsive.

A network can be described by means of nested node procedures, for example

```
node sum(max: int; out total: numbers)
  node counter( ... ) begin ... end
  node adder ( ... ) begin ... end
var stream: numbers
begin ... end
```

Here the counter and adder procedures are imbedded in a node procedure that outputs the sum of the numbers 1, 2, ... , max when it is called. A local sequence named stream connects the counter and adder nodes during their execution.

A node procedure call has the form

Procedure name (Argument list)

The argument list consists of expressions that are evaluated to obtain the values of the constant parameters,

and of variable symbols that are evaluated to select the distinct sequences denoted by the input/output parameters of the procedure.

A node procedure call is executed in four steps:

(1) A fresh instance of the procedure parameters is created by evaluating the arguments one at a time in the order written.

(2) A fresh instance of the local variables of the procedure is created with undefined initial values.

(3) The statement list of the procedure is executed.

(4) The parameter and variable instances created in steps (1) and (2) cease to exist.

Some examples of node procedure calls are:

```
counter(max, stream)
adder(stream, total)
```

A network program consists of a node procedure that is not contained in any other procedure.

CONCURRENT STATEMENTS

A concurrent statement of the form

```
cobegin P1(a1) // P2(a2) // ... // Pn(an) end
```

describes simultaneous execution of the node procedure calls P1(a1), P2(a2), ... , Pn(an). The execution of the concurrent statement terminates when the execution of all the node procedure calls have terminated.

The sum procedure outlined earlier includes the following statement list.

```
begin open(stream);
  cobegin counter(max, stream) // adder(stream, total)
  end
end
```

When this statement list is executed, a local sequence named stream is opened, and the counter and adder procedures are then executed simultaneously. The stream is used as an output sequence by the counter procedure and as an input sequence by the adder procedure.

This is an example of a limited network - a network in which the nodes only exchange sequences of finite length [1]. Furthermore, it is an acyclic network since the links do not form closed paths among the nodes. A theorem in [1] states that an acyclic, limited network always terminates.

A node procedure that includes sequential statements only describes a process. The counter and adder procedure are of this kind.

A node procedure that includes concurrent statements

describes a network. The sum procedure is such a procedure.

The neutral term "node" was deliberately chosen instead of the more familiar term "process" to describe an abstract machine that may perform its input/output operations either one at a time or simultaneously. A node is simply a consumer and producer of a fixed number of data sequences.

IMPLEMENTATION

On a computer with a common store, the implementation of data sequences can be described by a module written in the programming language Edison.

A sequence type T is represented by a record type

```
record T (state: phase; element: E)
```

with two fields defining the current state of the sequence and the value of the last element (if any) output to the sequence.

An enumeration type

```
enum phase (openx, morex, closed))
```

defines the possible states of the sequence:

openx: the value of more is undefined.

morex: the value of more is true.

closed: the value of more is false.

The indivisible operations on a sequence will be described by means of the when statement of Edison:

```
when B1 do SL1
else B2 do SL2
  ...
else Bn do SLn end
```

A process executes a when statement in two phases, called the synchronizing phase and the critical phase:

(1) In the synchronizing phase, the process is delayed until no other process is in a critical phase.

(2) In the critical phase, the process evaluates the boolean expressions B1, B2, ... , Bn one at a time until the value true is obtained from one of them or until all of them have been found false. In the latter case, the process returns to the synchronizing phase. But, if the value of an expression Bi is true, the statement list SLi that follows the expression is executed. This completes the execution of the when statement.

The scheduling of processes that are waiting to enter

critical phases one at a time is fair.

A sequence x of type T is opened by calling the following procedure:

```
proc open(var x: T)
begin when true do x.state := openx end
end
```

An output operation consists of assigning an element value e to the sequence x and then waiting until the value has been input:

```
proc output(var x: T; e: E)
begin
  when x.state = openx do
    x.element := e; x.state := morex
  else x.state = closed do fail end;
  when x.state = openx do skip end
end
```

The output operation fails if the sequence is closed.

An input operation consists of waiting until an element of the sequence x is available and then assigning the element value to a variable v:

```
proc input(var x: T; var v: E)
begin
  when x.state = morex do
    v := x.element; x.state := openx
  else x.state = closed do fail end
end
```

The input operation fails if the sequence is closed.

A sequence x is closed as follows:

```
proc close(var x: T)
begin when true do x.state := closed end
end
```

Finally, we have

```
proc more(x: T): bool
begin
  when x.state = morex do val more := true
  else x.state = closed do val more := false end
end
```

At the machine level of programming, it may be more efficient to store the address of a data element rather than its value.

In the general case, where a sequence may be used to

transmit values in both directions between two nodes, more states are needed to detect communication failures, but the details are very similar.

Node procedures and concurrent statements can be represented by the procedures and concurrent statements of the Edison language.

FOR STATEMENTS

The input and processing of a whole sequence x by means of a loop of the form

```
while more(x) do x?v; SL end
```

where SL is a statement list, can be expressed more concisely by a repetitive input statement of the form

```
for v in x do SL end
```

which means "for each remaining element v (if any) in the sequence x execute the statement list SL."

The for statement serves as a declaration of the variable v, which per definition is of the same type as the elements of x and is local to the statement list SL.

The while statement described above may be considered an implementation of the for statement.

The following version of the adder procedure illustrates the use of the for statement:

```
node adder(in x: numbers; out y: numbers)
var a: int
begin a := 0;
  for b in x do a := a + b end;
  y!a; close(y)
end
```

The next example taken from [6] is a node that inputs a deck of punched cards and outputs them as a string of characters. The node deletes spaces at the end of each card and terminates it by a newline character in the output.

```
node reader(in deck: lines; out text: string)
var m, n: int
begin
  for card in deck do
    if card <> blankcard do
      m := 1; n := 80;
      while card[n] = space do n := n - 1 end;
      while m <= n do
        text!card[m]; m := m + 1
      end
    end;
    text!newline
  end;
  close(text)
end
```

A node that serves as a single-slot character buffer between a predecessor and a successor node is defined as follows:

```
node slot(in pred: string; out succ: string)
begin for value in pred do succ!value end;
  close(succ)
end
```

A group of these nodes can be connected to form a

multislot buffer:

```
node buffer(in pred: string; out succ: string)
  node slot( ... ) begin ... end
var chain: group
begin
  cobegin slot(pred, chain[1])
    // all i [2:n] slot(chain[i - 1], chain[i])
    // slot(chain[n], succ)
  end
end
```

A quantified procedure call of the form

all i [m:n] P

is an abbreviation for the list of procedure calls

P(i/m) // P(i/m+1) // ... // P(i/n)

where P(i/e) stands for the call obtained by replacing all free ocurrences of the variable i in the call P by the expression e.

The buffer slots are fair in the sense defined in [1]:

Fair node: A responsive node that continues to communicate with each of its neighbors at a finite rate (>

0) as long as they are reachable and the node is not blocked forever.

Since the buffer network is also acyclic, it has the following properties [1]: None of the buffer slots can be blocked forever. If the input to the network is a finite sequence, the network terminates.

A similar network that inputs a sequence of numbers at one end and outputs them in increasing order at the other end can be built out of nodes like the following

```
node sort(in pred: numbers; out succ: numbers)
var x: int
begin
  if more(pred) do
    pred?x;
    for y in pred do
      succ!min(x, y); x := max(x, y)
    end;
    succ!x
  end;
  close(succ)
end
```

Consider now a cyclic network consisting of two nodes playing the game of Nim. From a pile of 20 coins, the players take turns picking one, two, or three coins from

the pile. The player forced to pick up the last coin loses the game.

The network is defined as follows

```
var x1, x2: numbers
begin open(x1); open(x2);
  cobegin player(20, x2, x1) // player(0, x1, x2) end
end
```

where

```
node player(pile: int; in x: numbers;
  out y: numbers)
var loser: bool
begin if pile > 0 do y!pile end;
  loser := false;
  for pile in x do
    if pile = 1 do close(y); loser := true
    else true do y!reduced(pile) end
  end;
  if not loser do close(y) end
end
```

To prove that this network terminates, we make the following observations:

(1) Each player continues to communicate with the other

player until the input sequence x is closed. The for statement maintains the invariant: loser => not more(y). When the for statement terminates, we have

not more(x) and (loser => not more(y))

When a player terminates, after executing the final if statement, the assertion not more(x) and not more(y) holds. So each player is a responsive node.

(2) Each player alternates between its two sequences in the cyclical order given by their indices

player 1: x1, x2, ...
player 2: x1, x2, ...

Together they form a clockwork net as defined in [1].

(3) If the input sequence x of a player is finite, the player terminates after closing its output sequence y. This can be expressed as follows: finite(x) => finite(y), or for each player

player 1: finite(x2) => finite(x1)
player 2: finite(x1) => finite(x2)

Consequently, we have finite(x1) = finite(x2), that is, either both of the sequences are finite, or none of them

are.

(4) According to a theorem in [1], none of the nodes in a clockwork net can be blocked forever. The players will therefore continue to reduce the pile until one of them loses and closes its output sequence. Since that sequence is finite then so is the other one as shown above. So the network is also limited. Another theorem in [1] states that a limited clockwork net always terminates.

This termination proof is also valid for a group of n nodes which play the game in cyclical order.

NON-DETERMINISTIC INPUT

A node that adds the elements of two sequences x and y can be described by two for statements which will be executed one at a time:

```
a := 0;
for b in x do a := a + b end;
for b in y do a := a + b end
```

Although this solution is correct, it imposes an unnecessary constraint on the node and its neighbors, since the elements of the two sequences may be added in any order.

In real-time applications, it is desirable to take advantage of such freedom of sequencing and enable a node to input the values of a fixed number of sequences as soon as they become available. This can be done by introducing a non-deterministic statement of the form

```
for v1 in x1 do SL1
else v2 in x2 do SL2
  ...
else vn in xn do SLn end
```

This statement is executed by examining the sequences x1, x2, ... , xn cyclically until one of the following situations arise:

(1) If more(xi) is true for a sequence xi, a data element is input from xi and assigned to a variable vi. Following this, the statement list SLi associated with vi is executed. When the execution of SLi terminates, the above process is repeated.

(2) When more(xi) is false for each sequence xi, the execution of the for statement terminates.

Each of the variables vi is local to the corresponding statement list SLi.

The addition of two sequences can now be expressed as follows

```
a := 0;
for b in x do a := a + b
else b in y do a := a + b end
```

Although the precise interleaving of the elements of the two sequences is unpredictable, the final result of executing the for statement is completely predictable.

Since the elements of the sequences may be interleaved in any order, the for statement may, for example, be implemented as follows:

```
while more(x1) do x1?v1; SL1
else more(x2) do x2?v2; SL2
  ...
else more(xn) do xn?vn; SLn end
```

This form of while statement (borrowed from Edison) is executed by evaluating the boolean expressions one at a time in the order written until the value true is obtained from one of them or until all of them have been found false. In the latter case, the execution of the while statement terminates. But, if the value of an expression is true, the statements that follow the expression are executed and, afterwards, the above process is repeated.

Although this algorithm is a correct implementation, it is not efficient, since the evaluation of the function

more(xi) delays a node until the next element of xi is ready to be output or until xi has been closed.

The previous implementation scheme must therefore be extended with a boolean function called more now. The value of this function is true if the function more is known to be true now and is false if more is either false now or cannot be evaluated yet.

```
proc morenow(x: T): bool
begin
  when true do val morenow := x.state = morex end
end
```

We will also need a similar function to determine if a sequence is closed now:

```
proc closednow(x: T): bool
begin
  when true do val closednow := x.state = closed end
end
```

The non-deterministic for statement can be implemented as follows:

```
while morenow(x1) do x1?v1; SL1
else morenow(x2) do x2?v2; SL2
  ...
else morenow(xn) do xn?vn; SLn
else not (and i) closednow(xi) do skip end
```

The abbreviation (and i) B, where B is a boolean expression stands for B(i/1) and B(i/2) ... and B(i/n).

In contrast to the function more, these two functions are time-dependent. They are, however, only used to implement the non-deterministic for statement and are not part of the abstract programming concepts.

The next example is a printer node connected to a group of n user nodes. The printer must input and print one text file from each of these nodes but may do so in any order.

```
node printer(in user: group)
begin
  for all i [1:n] x in user[i] do
    print(x);
    for x in user[i] do print(x) end;
    formfeed
  end
end
```

The notation

```
all i [m:n] x in y do SL
```

stands for

```
x in y(i/m) do SL(i/m)
else x in y(i/m+1) do SL(i/m+1)
  ...
else x in y(i/n) do SL(i/n)
```

CONDITIONAL INPUT

When the ability of a node to interleave the elements of several input sequences depends on the internal state of the node, we need a conditional for statement of the form

```
for v1 in x1 when B1 do SL1
  ...
else vn in xn when Bn do SLn end
```

The meaning of this statement is that the input of a value vi from a sequence xi and the execution of a statement list SLi takes place only if more(xi) and a boolean expression Bi both have the value true.

The for statement terminates when the predicate not (more(xi) and Bi) holds for each of the pairs (x1, B1),

... , (xn, Bn).

Each variable vi is local to the corresponding statement list SLi and cannot occur in the boolean expression Bi. The evaluation of Bi must not have any side-effects.

The conditional for statement is implemented as follows:

```
while morenow(x1) and B1 do x1?v1; SL1
  ...
else morenow(xn) and Bn do xn?vn; SLn
else not (and i) closednow(xi) or not Bi do
  skip
end
```

The conditional for statement is used in the following procedure that describes the merging of two ordered sequences x and y into a single, ordered sequence z.

```
node merge(in x, y: numbers; out z: numbers)
var a, b: int
begin
  if more(x) and more(y) do
    x?a; y?b;
    for c in x when a <= b do z!a; a := c
    else c in y when a > b do z!b; b := c end;
    z!min(a, b); z!max(a, b)
  end;
  for c in x do z!c
  else c in y do z!c end;
  close(z)
end
```

When the conditional loop terminates, either x or y must be closed. For, if we assume that both sequences are still open, the loop must continue since either a <= b or a > b. Consequently, the assertion not more(x) or not more(y) holds after the if statement.

The final loop copies the remaining sequence (if any). When this loop terminates, we have not more(x) and not more(y). When the node terminates, all its sequences have been closed. So the node is responsive.

The next example is a node that behaves like a general semaphore [7]:

```
node semaphore(in p, v: signals)
var n: int
begin n := 0;
  for x in p when n > 0 do n := n - 1
  else x in v when true do n := n + 1 end
end
```

This node is not intended to terminate. If it does, the sequence p may not be closed, since the assertion

```
(not more(p) or (n = 0)) and not more(v)
```

holds. Consequently, the semaphore is not a responsive node.

In CSP, a process acting as a multislot buffer between two other processes requires a signal from the receiver before an element is output [2]. A similar node can be programmed using the conditional for statement. Such a solution is, however, not nearly as elegant as the buffer network described earlier in this paper. A network is also better suited to a future technology that may provide a large number of identical processors on a single chip.

The conditional for statement is similar to the repetitive command of CSP (although the termination condition is different in CSP):

```
*[v1: integer; B1; P1?v1 -> SL1|
   ...
  vn: integer; Bn; Pn?vn -> SLn]
```

I have doubts about the wisdom of introducing such a complicated statement in a programming language. For this reason, it has been presented here as a complex extension of much simpler programming concepts.

FINAL REMARKS

The development of these ideas was motivated by the expectation that distributed computing would turn out in principle to be similar to the processing of sequential files. This viewpoint led to the use of typed data sequences as the only means of communication in a computer network. The idea of terminating input loops at the end of sequences followed naturally.

In applications where several data sequences may be processed in arbitrarily interleaved order, the freedom of interleaving is used to improve the real-time performance. But the use of nondeterminism takes place on a microscopic time scale only and must eventually produce a deterministic result on a macroscopic time scale. This form of manageable nondeterminism is also advocated in [8].

Although sequences of mixed types give the programmer

additional freedom of expression, the programming examples in this paper demonstrates the surprising power of uniform sequences.

Finally, it should be emphasized that the language concepts proposed here have not yet been implemented. Although it seems obvious how to do this on a single processor, the problem of designing an appropriate architecture for a network of microprocessors remains unsolved.

ACKNOWLEDGEMENT

It is a pleasure to acknowledge the comments of David Jefferson, Peter Lyngbaek and Tom Mowbray.

REFERENCES

[1] P. Brinch Hansen, Basic concepts of network programs. Comput, Sci. Dep., Univ. Southern California, Los Angeles, CA, Sep. 1981.

[2] C. A. R. Hoare, "Communicating sequential processes," Commun. Ass. Comput. Mach., vol. 21, pp. 666-677, Aug. 1978.

[3] G. M. Levin and D. Gries, "A proof technique for communicating sequential processes," Acta Informatica, vol. 15, no. 3, pp. 281-302, 1981.

[4] P. Brinch Hansen, "Edison - a multiprocessor language," Software - Practice and Experience, vol. 11, no. 4, pp. 325-361, April 1981.

[5] P. Brinch Hansen, Proof rules for network programs. Comput. Sci. Dep., Univ. Southern California, Los Angeles, CA, Sep. 1981.

[6] P. Brinch Hansen, "Distributed processes: a concurrent programming concept," Commun. Ass. Comput. Mach., vol. 21, no. 11, pp. 934-941, Nov. 1978.

[7] E. W. Dijkstra, "Cooperating sequential processes," Programming Languages, New York, NY: Academic Press, 1968.

[8] ___, "Hierarchical ordering of sequential processes," Operating Systems Techniques. New York, NY: Academic Press, 1972.

APPENDIX: SYNTAX SUMMARY

The following defines the syntax of the programming language in which the examples of this paper are written. The syntax is defined by means of the extended BNF notation used in the Edison report [4]. The notation [E] stands for zero or one occurrence of the syntactic form E, while [E]* denotes zero or more occurrences of E. Alternative forms are separated by the character #, and the basic symbols of the language are enclosed in quotes.

```
Constant symbol:
  Constant name # Numeral
Range symbol:
  '[' Constant symbol ':' Constant symbol ']'
Array type declaration:
  'array' Type name Range symbol '(' Type name ')'
Sequence type declaration:
  'seq' Type name '(' Type name ')'
Type declaration:
  Array type declaration # Sequence type declaration
Variable group:
  Variable name [ ',' Variable name ]* ':' Type name
Variable declaration list:
  'var' Variable group [ ';' Variable group ]*
Parameter group:
  [ 'in' # 'out' ] Variable group
Parameter list:
  Parameter group [ ';' Parameter group ]*
Procedure heading:
  'node' Procedure name [ '(' Parameter list ')' ]
Procedure body:
  [ Declaration ]* 'begin' Statement list 'end'
Procedure declaration:
  Procedure heading Procedure body
Declaration:
  Type declaration # Variable declaration list #
  Procedure declaration
Variable symbol:
  Variable name # Variable symbol '[' Expression ']'
Factor:
  Constant symbol # Variable symbol # 'not' Factor #
  '(' Expression ')'
Multiplying operator:
  '*' # 'div' # 'mod' # 'and'
Term:
  Factor [ Multiplying operator Factor ]*
Adding operator:
  '+' # '-' # 'or'
Simple expression:
  [ '+' # '-' ] Term [ Adding operator Term ]*
Relational operator:
 '=' # '<>' # '<' # '<=' # '>' # '>='
Expression:
  Simple expression
    [ Relational operator Simple expression ]
Assignment statement:
  Variable symbol ':=' Expression
Conditional statement:
  Expression 'do' Statement list
Conditional statement list:
  Conditional statement [ 'else' Conditional statement ]*
```

```
If statement:
  'if' Conditional statement list 'end'
While statement:
  'while' Conditional statement list 'end'
Input statement:
  Variable symbol '?' Variable symbol
Output statement:
  Variable symbol '!' Expression
Quantifier:
  'all' Variable name Range symbol
Input clause:
  [ Quantifier ] Variable name 'in' Variable symbol
    [ 'when' Expression ]
Conditional input statement:
  Input clause 'do' Statement list
Conditional input list:
  Conditional input statement
    [ 'else' Conditional input statement ]*
For statement:
  'for' Conditional input list 'end'
Argument:
  Expression # Variable symbol
Argument list:
  Argument [ ',' Argument ]*
Procedure call:
  Procedure name [ '(' Argument list ')' ]
Quantified procedure call:
  [ Quantifier ] Procedure call
Procedure call list:
  Quantified procedure call
    [ '//' Quantified procedure call ]*
Concurrent statement:
  'cobegin' Procedure call list 'end'
Statement:
  'skip' # Assignment statement # If statement #
  While statement # For statement # Concurrent statement
Statement list:
  Statement [ ';' Statement ]*
Program:
  Procedure declaration
```

PROOF RULES FOR NETWORK PROGRAMS

Per Brinch Hansen

Computer Science Department
University of Southern California
Los Angeles, California 90007, U.S.A.

August 1981
Revised September 1981

Summary

This paper introduces rules for proving the partial correctness of concurrent programs written for a class of computer networks known as coincidence networks. The use of the proof rules is illustrated by means of annotated program examples.

Key Words and Phrases: program correctness, proof rules, distributed computing, computer networks, input/output, concurrency

CR Categories: 4.2, 4.22, 5.24

This research was supported by the Office of Naval Research under Contract NR 048-647.

1. INTRODUCTION

A previous paper defines a class of computer networks known as coincidence networks and gives sufficient conditions for the absence of local deadlocks and the termination of such networks [1]. A formal notation in which programs for coincidence networks can be written is described in [2].

This paper introduces rules for proving the partial correctness of concurrent programs written for coincidence networks. The use of these proof rules is illustrated by annotated program examples.

2. SEQUENCE TYPES

A coincidence network consists of a finite number of abstract machines, called nodes, and a finite number of data channels, called links. Each link connects exactly two nodes and enables the nodes to exchange a data sequence. The transfer of a data value from one node to another is called a communication. It takes place when one of the nodes is ready to output a data value to a sequence and the other node is ready to input a data value from the same sequence. Two nodes that are able to communicate are called neighboring nodes.

A type declaration of the form

seq T (E)

introduces a name T to denote sequences consisting of elements of some known type E.

Examples:

seq numbers (int)

seq signals (bool)

The sequences of type T are defined as follows [3]:

(1) The empty T sequence is denoted <>.

(2) If x is a T sequence and e is an E value then x <e> denotes the T sequence obtained by appending e to x.

(3) The only T sequences are those given by (1) and (2).

The abbreviation <e1, e2, ... , en> stands for the sequence <> <e1> <e2> ... <en>. Example: <1, 2, 3>.

The last element of a non-empty sequence x is denoted last(x), while the previcus elements of the sequence are called prev(x), that is

x = prev(x) <last(x)>

Example:

last(<1, 2, 3>) = 3

prev(<1, 2, 3>) = <1, 2>

<1, 2, 3> = <1, 2> <3>

The last element of a non-empty sequence is defined by the rule

$$last(x\ \langle e \rangle) = e$$

The length of a sequence x is denoted $|x|$ and is defined as follows

$$|\langle\rangle| = 0$$
$$|x\ \langle e \rangle| = |x| + 1$$

The sequence constructor < ... > and the functions prev, last, and $|x|$ are not part of the programming notation but are used to make assertions about sequences.

3. INPUT/OUTPUT STATEMENTS

The proof rules will be explained in Hoare's notation [4]

$$P\ \{\ S\ \}\ R$$

which means the following: If the assertion P is true before the execution of the statement S, and if the execution of S terminates, then the assertion R will be true afterwards.

In particular, we will use the familiar rule of assignment:

$$P(x/e)\ \{\ x := e\ \}\ P$$

where P(x/e) denotes the assertion obtained from P by replacing all free occurrences of the variable x by the expression e. This is known as backward substitution.

In a network program, a sequence x of type T is declared as a variable

var x : T

In program assertions, the name of a sequence denotes the entire sequence of values output so far. This idea greatly simplifies the proof rules to be described.

Initially, a sequence x must be initialized by performing an open operation

open(x)

An output statement

x!e

describes the output of a data value e to a sequence x. The

value e becomes conceptually the last element of x.

An input statement

x?v

describes the input of a data value from a sequence x. The value is assigned to a variable v. Since the output and input of the same value coincide in time, the input value is also the last element of x.

If two nodes exchange a sequence x of finite length, one of the nodes must perform a close operation

close(x)

when all the elements of the sequence have been output. Further input/output operations on x are now undefined.

A node may evaluate a boolean function named more

more(x)

to determine whether or not another data value will be output to a sequence x. The evaluation takes place either when another node is ready to output to the sequence (in which case, more is true) or when the sequence has been closed (in which case, more is false).

In the usual mathematical sense, more is not a sequence

function, since it is meaningless to ask what the value of more is for a given sequence, say more(<1, 2, 3>). This function is only defined in the context of program execution. It enables one node to determine whether another node has finished the computation of a sequence. In reasoning about partial correctness, we are not concerned about termination. Consequently, more does not appear in the proof rules to be described.

After this summary of the input/output statements, we are ready to introduce proof rules.

The initial statement open(x) corresponds to the assignment

$$x := \langle\rangle$$

and is defined by the following rule of opening:

$$P(x/\langle\rangle) \{ \text{open}(x) \} P$$

The first annotated program example illustrates a trivial use of this rule:

"max $\geq$ 0, total = <>" open(stream)
"max $\geq$ 0, total = <>, stream = <>"

The assertions are written as program comments enclosed in quotes using commas instead of and operators.

An output statement corresponds to the assignment

$$x := x \langle e \rangle$$

and is defined by the following rule of output:

$$P(x/x\langle e\rangle)\ \{\ x!e\ \}\ P$$

In the next program example, the notation $\langle 1:i \rangle$ stands for $\langle\rangle$, if $i = 0$, and for $\langle 1, 2, \ldots, i \rangle$, if $i > 0$:

"$0 < i \leq n$, $x = \langle 1:i-1 \rangle$" $x!i$
"$0 < i \leq n$, $x = \langle 1:i \rangle$"

If we replace x by $x \langle i \rangle$ in the postcondition, we obtain

$$0 < i \leq n,\ x \langle i \rangle = \langle 1:i \rangle$$

which is equivalent to the precondition used.

An input statement corresponds to the assignments

$$x := x \langle e \rangle;\ v := e$$

Now, the expression e is unknown within a receiving node. So, if we use backward substitutions based on the rule of assignment, we obtain the following rule of input:

(all e) Q(v/e, x/x<e>) { x?v } Q

where Q(v/e, x/x<e>)) denotes the assertion obtained from Q by simultaneously replacing all free occurrences of the distinct variables v and x by the expressions e and x<e>, respectively. Since the expression e is unknown, any assertion made about the sequence x must be true for any e. This requirement is expressed by the universal quantifier (all e).

The next example is taken from a program that adds a sequence of numbers according to the following rules

sum(<>) = 0
sum(x <e>) = sum(x) + e

The program includes the following input statement

"a = sum(x)" x?b "a + b = sum(x)"

The postcondition Q of this statement is: a + b = sum(x). The input statement is correct, if the precondition used implies (all e) Q(b/e, x/x<e>), that is, if

a = sum(x) => (all e) a + e = sum(x<e>)

which indeed is true.

The "satisfaction proof" required in [5] consists of establishing the equivalence between every matching pair of input/output operations and a corresponding assignment. Such a proof is unnecessary for the verification method presented here, because the effect of input/output operations is defined in terms of equivalent assignments to begin with.

Since data elements are input as fast as they are output, there is no need to store them in a buffer between neighboring nodes. The complete sequence of data values communicated through a variable x may, however, be referred to in assertions about a network program. This is the only kind of auxiliary variable used in program assertions.

All assertions about a network node are expressed in terms of the local variables of the node and the input/output sequences used by the node to communicate with its neighbors. Since these variables only can be changed when the node performs an operation, the operations of other nodes cannot change the assertions about the state of the given node at unpredictable times. So there is no need to prove that the nodes are "interference free" either as required in [5].

The rule of closing is simply

$$P \{ \text{close}(x) \} P$$

Although it is irrelevant for partial correctness, not more(x) is also true after a close operation.

Example:

"n $\geq$ 0, x = <1:n>" close(x) "n $\geq$ 0, x = <1:n>"

An earlier paper describes a node that merges two ordered sequences into a single, ordered sequence [2]. A verification of this node depends on the assumption that the input sequences are ordered. Whether or not this invariant is satisfied cannot be established by studying the merging node, since it depends on the nodes that output these sequences.

The following rule of invariants makes it possible to use a sequence invariant established by one node in the proof of another node:

If the opening of a sequence x establishes an assertion I about x, and, if every output operation on x maintains I, then I is also true before and after each input operation on x. The local variables of a node cannot occur in I.

4. FOR STATEMENTS

The repetitive input statement

for v in x do SL end

means "for each remaining element v (if any) in the sequence x, execute the statement list SL." This is an abbreviation for the following while statement

while more(x) do x?v; SL end

The rule of repetitive input is

$$\frac{P \Rightarrow (\text{all } e)\ Q(v/e,\ x/x\langle e\rangle),\quad Q\ \{\ SL\ \}\ P}{P\ \{\ \text{for } v \text{ in } x \text{ do } SL \text{ end}\ \}\ P}$$

The variable v must not occur free in P.

This proof rule is derived as follows from the corresponding while statement:

(1) Since the for statement is equivalent to a while statement, an invariant P must be true before and after the for statement, before the implied input of v, and after the statement list SL.

(2) Since SL is executed after the input of v, the precondition of SL must be the postcondition Q of an input operation.

(3) The invariant P must therefore imply the precondition

of the input operation, which is obtained from Q by simultaneous substitution in accordance with the rule of input.

When the for statement terminates, not more(x) and P holds, but, as explained earlier, more(x) has no meaning in a proof rule for partial correctness.

It seems startling only to assume that a loop maintains an invariant without establishing a stronger postcondition. Later, we shall see why this proof rule nevertheless works for network programs (Section 6).

Example:

```
"a = sum(x)"
for b in x do
  "a + b = sum(x)" a := a + b "a = sum(x)"
end
"a = sum(x)"
```

In this example, the loop invariant P is: a = sum(x). According to the rule of assignment, the desired precondition Q of the assignment statement is: a + b = sum(x). As shown earlier, P => (all e) Q(b/e, x/x<e>). So the hypothesis of the rule of repetitive input is satisfied. Consequently, the invariant P is also true after the execution of the for statement.

5. NODE PROCEDURES

The actions performed by a node are described by a node procedure of the form

node P(x) D begin SL end

It consists of a procedure name P, a parameter list x, some declarations D of local entities, and a statement list SL.

The parameter list describes parameters of two kinds:

(1) The constant parameters denote fixed values that are computed when the procedure is called.

(2) The input/output parameters denote sequences that are selected when the procedure is called. The sequences are distinct and none of them are contained in the constant arguments of the call.

A node procedure can call global node procedures but cannot refer to other kinds of global procedures or to global variables. To simplify matters, we will only consider non-recursive node procedures in this paper.

The following example is an annotated procedure executed by a node that outputs the sequence of natural numbers 1, 2, ... , n and terminates.

```
node counter(n: int; out x: numbers)
var i: int
begin "n ≥ 0, x = <>" i := 0;
  "0 ≤ i ≤ n, x = <1:i>"
  while i < n do
    "0 ≤ i < n, x = <1:i>" i := i + 1;
    "0 < i ≤ n, x = <1:i-1>" x!i
    "0 < i ≤ n, x = <1:i>"
  end;
  "n ≥ 0, x = <1:n>" close(x)
  "n ≥ 0, x = <1:n>"
end
```

The next example describes a node that inputs a sequence of numbers, adds them, and outputs the sum.

```
node adder(in x: numbers; out y: numbers)
var a: int
begin "x = <>, y = <>" a := 0;
  "a = sum(x), y = <>"
  for b in x do
    "a + b = sum(x), y = <>" a := a + b
    "a = sum(x), y = <>"
  end;
  "a = sum(x), y = <>" y!a; close(y)
  "y = <sum(x)>"
end
```

A node procedure call P(a) denotes execution of the node procedure P with an argument list a.

For our purposes, it is sufficient to use the following rule of invocation for non-recursive procedures [6]:

$$\frac{\text{node } P(x)\ D\ \text{begin } SL\ \text{end},\ Q\ \{\ SL\ \}\ R}{Q(x/a)\ \{\ P(a)\ \}\ R(x/a)}$$

This rule states that if the precondition of a node procedure call is obtained from the precondition of the procedure body by replacing each parameter in the list x by the corresponding argument in the list a, then the postcondition of the call is obtained from the

postcondition of the body by a similar substitution. The local entities in D must not occur in the assertions Q and R about the statement list SL.

Example:

"max $\geq$ 0, stream = <>" counter(max, stream)

"max $\geq$ 0, stream = <1:max>"

Example:

"stream = <>, total = <>" adder(stream, total)

"total = <sum(stream)>"

6. CONCURRENT STATEMENTS

A concurrent statement of the form

cobegin P1(a1) // ... // Pn(an) end

describes simultaneous execution of the node procedure calls P1(a1), ... , Pn(an). The execution of the concurrent statement terminates when the execution of all the node procedure calls have terminated.

The rule of concurrency is similar to the rule used for concurrent processes with shared variables [7]:

$$\frac{(\underline{\text{all}}\ i)\ Q_i\ \{\ P_i(a_i)\ \}\ R_i}{\begin{array}{l}(\underline{\text{and}}\ i)\ Q_i \\ \quad \{\ \underline{\text{cobegin}}\ P_1(a_1)\ //\ \ldots\ //\ P_n(a_n)\ \underline{\text{end}}\ \} \\ (\underline{\text{and}}\ i)\ R_i\end{array}}$$

It expresses the obvious: If the precondition of each procedure call holds before the execution of the concurrent statement, and if the execution of each procedure call terminates, then the postcondition of each call will hold after the execution of the concurrent statement.

The procedure calls must satisfy the restrictions imposed on node procedure calls (Section 5). In particular, the only variables that can be shared by the nodes are the sequence variables that occur in the argument lists of the calls. Furthermore, each sequence variable must be shared by exactly two of the nodes (Section 2). And, finally, the sequences used by a given node must be distinct. Under these assumptions, the nodes are always interference free as mentioned earlier (Section 3).

In the following example, this rule is used to annotate a network consisting of a counter node that outputs a sequence of numbers to an adder node:

```
node sum(max: int; out total: numbers)
  node counter( ... ) begin ... end
  node adder( ... ) begin ... end
var stream: numbers
begin "max ≥ 0, total = <>" open(stream);
  "max ≥ 0, total = <>, stream = <>"
  cobegin
    "max ≥ 0, stream = <>" counter(max, stream)
    "max ≥ 0, stream = <1:max>"
    //
    "stream = <>, total = <>" adder(stream, total)
    "total = <sum(stream)>"
  end
  "max ≥ 0, total = <sum(<1:max>)>"
end
```

The local variable named stream has been eliminated in the postcondition, because it is unknown outside the sum procedure.

Although the adder node only maintains an invariant: total = <sum(stream)>, the counter node establishes a postcondition: stream = <1:max>, that defines the elements of the stream. The conjunction of these assertions implies the desired postcondition: total = <sum(<1:max>)>. That is why the rule of repetitive input works (Section 4).

The termination of this network is proved in [2].

7. CONDITIONAL INPUT STATEMENTS

The conditional input statement

```
for v1 in x1 when B1 do SL1
else v2 in x2 when B2 do SL2
  ...
else vn in xn when Bn do SLn end
```

enables a node to interleave the input of several sequences x1, x2, ... , xn in some arbitrary order. An element vi of a sequence xi can be input as soon as another node is ready to output it, provided a boolean expression Bi has the value true. The variable vi is local to the statement list SLi and cannot occur in Bi. The evaluation of Bi must not have side-effects.

The rule of conditional input is an obvious extension of the earlier rule of repetitive input:

$$\frac{\begin{array}{l}(\underline{\text{all}}\ i)\\ \quad P\ \underline{\text{and}}\ Bi \Rightarrow (\underline{\text{all}}\ e)\ Qi(vi/e,\ xi/xi\langle e\rangle),\\ \qquad\qquad Qi\ \{SLi\}\ P\end{array}}{\begin{array}{l}P\ \{\ \underline{\text{for}}\ v1\ \underline{\text{in}}\ x1\ \underline{\text{when}}\ B1\ \underline{\text{do}}\ SL1\\ \quad \ldots\\ \quad \underline{\text{else}}\ vn\ \underline{\text{in}}\ xn\ \underline{\text{when}}\ Bn\ \underline{\text{do}}\ SLn\ \underline{\text{end}}\ \}\ P\end{array}}$$

The local variables vi cannot occur free in the invariant P or in the corresponding expressions Bi.

Since the input of the sequences may be interleaved in arbitrary order, the only assumption that can be made before the input of an element vi is that the invariant P and the corresponding expression Bi hold. That is what the proof rule expresses.

When the conditional input statement terminates, the assertion

$$\underline{\text{not}}\ (\text{more}(xi)\ \underline{\text{and}}\ Bi)$$

is true for each pair (xi, Bi). But this postcondition is omitted in the proof rule for partial correctness.

In the following example, this rule is used to annotate a node that acts as a general semaphore:

```
node semaphore(in p, v: signals)
var n: int
begin "|p| = 0, |v| = 0" n := 0;
  "|p| + n = |v|, n ≥ 0"
  for x in p when n > 0 do
    "|p| + n - 1 = |v|, n > 0" n := n - 1
    "|p| + n = |v|, n ≥ 0"
  else x in v when true do
    "|p| + n + 1 = |v|, n ≥ 0" n := n + 1
    "|p| + n = |v|, n > 0"
  end
  "|p| + n = |v|, n ≥ 0"
end
```

The loop invariant P is

$$|p| + n = |v|,\ n \geq 0$$

It shows that the number of p operations cannot exceed the number of v operations.

From P we derive the precondition Q of the p operation using the rule of assignment:

$$Q:\ |p| + n - 1 = |v|,\ n - 1 \geq 0$$

We must now show that

$$P \text{ and } (n > 0) \Rightarrow (\text{all } e)\ Q(x/e,\ p/p\langle e\rangle)$$

that is

$$|p| + n = |v|,\ n > 0 \Rightarrow (\text{all } e)\ |p\langle e\rangle| + n - 1 = |v|,\ n > 0$$

or

$$|p| + n = |v|,\ n > 0 \Rightarrow |p| + 1 + n - 1 = |v|,\ n > 0$$

which certainly is true. So, the hypothesis of the proof rule is satisfied for the p operation. The hypothesis of the v operation is verified by a similar argument.

8. FINAL REMARKS

We have introduced proof rules for establishing the partial correctness of network programs. All assertions about a network node are expressed in terms of the local variables of the node and the data sequences which the node exchanges with its neighbors. A "satisfaction proof" of the form required in [5] is unnecessary, and the nodes are always "interference free."

It is encouraging that the proof rules support the

intuitive meaning of the programming concepts in an obvious way. The trivial nature of the examples may not appeal to most programmers. They do, however, demonstrate that the concepts have a solid mathematical foundation.

This paper has only considered the partial correctness of computer networks. Another paper [1] gives sufficient conditions for the termination of a large class of networks based on minimal assumptions about the communication patterns followed by their nodes. An economical argument of this form seems preferable to me compared to finding a separate termination proof for each network program based on detailed assertions.

ACKNOWLEDGEMENT

This paper has been improved considerably by the advice of David Jefferson. I am also grateful to Peter Lyngbaek and Tom Mowbray for their constructive comments.

REFERENCES

1. BRINCH HANSEN, P., Basic concepts of network programs. Computer Science Department, University of Southern California, Los Angeles, CA, Sep. 1981.

2. BRINCH HANSEN, P., Language notation for network programs. Computer Science Department, University of Southern California, Los Angeles, CA, Sep. 1981.

3. HOARE, C. A. R., "Notes on data structuring," Structured Programming. Academic Press, New York, NY, (1972), 83-174.

4. HOARE, C. A. R., "An axiomatic basis for computer programming," Comm. ACM 12, 10 (Oct. 1969), 576-580, 583.

5. LEVIN, G. M., AND GRIES, D., "A proof technique for communicating sequential processes," Acta Informatica 15, 3 (1981), 281-302.

6. HOARE, C. A. R., "Procedures and parameters: an axiomatic approach," Lecture Notes in Mathematics 188, Springer-Verlag, New York, NY, 1971.

7. HOARE, C. A. R., "Towards a theory of parallel programming," Operating Systems Techniques, Academic Press, New York, NY, (1972), 61-71.

DISTRIBUTION LIST

FOR THE TECHNICAL, ANNUAL, AND FINAL REPORTS

FOR CONTRACT N00014-77-C-0714

Defense Documentation Center Cameron Station Alexandria, VA 22314	12 copies
Office of Naval Research Information Systems Program Code 437 Arlington, VA 22217	2 copies
Office of Naval Research Code 715LD Arlington, VA 22217	6 copies
Office of Naval Research Code 200 Arlington, VA 22217	1 copy
Office of Naval Research Code 455 Arlington, VA 22217	1 copy
Office of Naval Research Code 458 Arlington, VA 22217	1 copy
Office of Naval Research Branch Office, Boston 495 Summer Street Boston, MA 02210	1 copy
Office of Naval Research Branch Office, Chicago 536 South Clark Street Chicago, Illinois 60605	1 copy
Office of Naval Research Branch Office, Pasadena 1030 East Green Street Pasadena, CA 91106	1 copy
Office of Naval Research New York Area Office 715 Broadway - 5th Floor New York, NY 10003	1 copy

Naval Research Laboratory — 6 copies
Technical Information Division
Code 2627
Washington, D. C. 20375

Dr. A. L. Slafkosky — 1 copy
Scientific Advisor
Commandant of the Marine Corps
(Code RD-1)
Washington, D. C. 20380

Naval Ocean Systems Center — 1 copy
Advanced Software Technology Division
Code 5200
San Diego, CA 92152

Mr. E. H. Gleissner — 1 copy
Naval Ship Research & Development
Center
Computation and Mathematics Department
Bethesda, MD 20084

Captain Grace M. Hopper — 1 copy
NAICOM/MIS Planning Branch (OP-916D)
Office of Chief of Naval Operations
Washington, D. C. 20350

Mr. Kin B. Thompson — 1 copy
NAVDAC 33
Washington Navy Yard
Washington, D.C. 20374


FILME