Proofs of Networks of Processes

JAYADEV MISRA, MEMBER, IEEE, AND K. MANI CHANDY

Abstract—We present a proof method for networks of processes in which component processes communicate exclusively through messages. We show how to construct proofs of invariant properties which hold at all times during network computation, and terminal properties which hold upon termination of network computation, if network computation terminates. The proof method is based upon specifying a process by a pair of assertions, analogous to pre- and post-conditions in sequential program proving. The correctness of network specification is proven by applying inference rules to the specifications of component processes. Several examples are proved using this technique.

Index Terms—Communication networks, distributed systems, message passing systems, program proofs.

I. Introduction

We propose a proof technique for networks of processes in which component processes communicate exclusively through messages, as in Hoare [8]. The technique is based upon specification of a process h by a pair of assertions r and s, analogous to pre- and post-conditions in sequential program proving. The specification is denoted by $r|h|s$, which means: 1) s holds initially in h and 2) if r holds at all times prior to any message transmission of h, then s holds at all times prior to and immediately following that message transmission, where a message transmission of process h could be either h sending or h receiving a message.

The proof technique is built around a few inference rules. These rules allow us to deduce the specification of a network from the specifications of its component processes. The advantages of such a proof technique are the following.

1) A network specification is obtained solely from component process specifications and not from the details of process implementation.

2) The proof technique supports the hierarchical decomposition of networks. Starting with R, S for network H, we construct r_i, s_i of component processes h_i's, such that the component process specifications yield the desired network specifications. The h_i's may in turn be networks themselves, in which case decomposition of h_i's into component processes proceeds hierarchically in the same manner.

We give several examples which demonstrate the power and convenience of using r, s to specify a process. Our inference rules are built upon Hoare's theory of *traces* [9].

Manuscript received May 23, 1979; revised August 18, 1980. This work was supported by the National Science Foundation under Grant MCS79-25383 and ARPA Grant Systems Performance Modeling Part II N00039-78-G-0080.

The authors are with the Department of Computer Sciences, University of Texas, Austin, TX 78712.

II. A Model of a Network of Processes

We are not concerned with the definition of an entire programming language in this paper. We are concerned only with proving properties about message communication among processes. We consider the message communication mechanism proposed by Hoare [8]. The example programs will be written in Hoare's *CSP* with the following minor differences. *CSP* uses an explicit process addressing mechanism in message communication. For instance, process A may have commands of the form $B?x$ to receive a message from process B and put its content in local variable x; similarly $B!x$ denotes transmission of a message to process B, where the content of the message is the value of x. For autonomous proofs it is preferable to avoid explicit process naming. Hence, we will only allow process A to communicate using commands of the form $C?x$ or $C!x$, where C denotes a channel (see Section II-B). As Hoare has noted, addressing via channels is semantically equivalent to explicit process addressing.

We expect the reader to know *CSP* because our model is derived from it. We briefly summarize below concepts related to message transmission that we use in this paper.

A. Process

A process communicates only by sending or receiving *messages.* A process is either a *sequential process*, i.e., a sequential program with commands for message transmission, or a *network of processes*, as described next.

B. Network

A *network* consists of a collection of *processes* and *channels* where processes transmit messages via channels. A channel C is directed from exactly one process h_i to exactly one other process h_j; C is then said to be *incident* on h_i and h_j. Process h_i (or h_j) is said to be *waiting on* C if h_i (or h_j) is waiting as in *CSP*, to send (or receive) a message along C. Messages can only be transmitted along the direction of a channel. If both h_i and h_j are within a network H, then C is said to be *internal* to H. If h_i is within H and h_j is not, C is said to be *incident* on H and directed away from H; in this case H is said to be waiting (to send a message) on C if h_i is waiting on C. Similarly, C is incident on and directed towards H if h_j is within H but h_i is not; H is said to be waiting on C if h_j is.

Consider channel C directed from process h_i to h_j. A message may be transmitted along C only if h_i is waiting to execute a statement of the form $C!x$ and h_j is waiting to execute a statement of the form $C?y$. The effect of transmission of the message is to assign x to y.

C. Hierarchical Process Construction

A process h within a network H may be implemented either as a sequential process or as a network of processes. However, processes (other than h itself) do not know the implementation of h; these processes are only concerned with h's interaction with other processes in H and not with interactions (if any) among component processes of h. The *external specification* of h will specify the manner of interaction of h with processes external to it. The external specification is independent of the implementation of h.

If h is implemented as a network of processes, it is often convenient to have an *internal specification* of h from which the external specification may be abstracted. The internal specification will specify the manner of interaction between component processes of h, as well as between component processes of h and processes external to h.

The proof mechanism is as follows. The internal and external specifications of a sequential process are identical. The internal specification of a network h is derived by using the inference rules from external specifications of its component processes. We next abstract the external specification of h from its internal specification, by using the inference rules.

D. Trace

The theory of traces of communicating sequential processes is due to Hoare [9]; we use slightly different notation and adapt a small fragment of that theory for this paper. Work on traces of sequential programs was first carried out by Bartussek and Parnas[1] and McLean.[2]

The *external trace* of a process h at any point in a computation is a sequence of tuples $<(C_1, v_1), (C_2, v_2), \cdots, (C_n, v_n)>$, where in that computation the ith message sent or received by h is along channel C_i *incident* on h and has value $v_i, i = 1, \cdots, n$.

The *internal trace* of h at any point in a computation is a sequence of tuples $<(C_1, v_1), \cdots, (C_n, v_n)>$, where in that computation the ith message transmitted on all channels *incident on or internal to* h, is transmitted along channel C_i and has value v_i, $i = 1, \cdots, n$. Initially all traces, internal and external, are null sequences and will be denoted by *empty*.

All assertions r and s will be on traces exclusively; thus, the entire proof technique deals only with propositions on traces. A trace may be considered to be an "auxiliary variable" for proof purposes. The notion of auxiliary or mythical variables was first introduced by Clint [2].

Let $t = <(C_1, v_1), \cdots, (C_i, v_i), \cdots, (C_n, v_n)>$ be a trace. We say that the assertion r holds up to the kth point in t if r holds for all traces $<(C_1, v_1), \cdots, (C_i, v_i)>$, where $0 \leqslant i \leqslant k$ and $i \leqslant n$; note that r holds for the *empty* trace in this case.

III. Inference Rules

An external specification of process h is given by a pair of assertions r, s on the external traces of h, where r states what is assumed by h and s states what h establishes under this assumption; we denote this specification by $r|h|s$. Formally, $r|h|s$ denotes that: 1) s holds initially in h and 2) if r holds up to the kth point in *any* external trace of h, then s holds up to the $(k + 1)$th point in that trace, for all $k \geqslant 0$.

An internal specification of process h is defined similarly on the internal traces of h and is denoted $r[h]s$.

A. Notation

It is often convenient to consider only the sequence of messages transmitted along a channel C in a trace; this is denoted by ZC. We adopt the following notations for sequences. In the following Z, Z_1, Z_2 denote sequences.

Z denotes the length of Z.

$Z_1 \underline{\propto} Z_2$ denotes that Z_1 is an initial substring of Z_2. Note: $Z \underline{\propto} Z$ for every Z.

$Z_1 \equiv Z_2$ denotes that Z_1 and Z_2 are identical sequences.

empty denotes the sequence having no element. Note: *empty* $\underline{\propto} Z$, for any Z.

$Z_1 | Z_2$ denotes the sequence obtained by concatenating Z_2 at the end of Z_1.

$<e_1, e_2, \cdots, e_n>$ denotes a sequence having elements e_1, $e_2 \cdots e_n$ in this order.

B. Example: Specification of a Buffer [8]

A buffer process h has two incident channels: *in* directed towards h and *out* directed away from it. Buffers may be specified easily using $r|h|s$. A common specification is: the sequence of messages output by the buffer must be an initial subsequence of the sequence of messages received by the buffer. This is written as *true* $|h|$ *Zout* $\underline{\propto}$ *Zin*.

Note that this is only one specification of a buffer; more detailed specifications can be obtained if desired.

[1]W. Bartussek and D. Parnas, "Using traces to write abstract specifications for software modules," Dep. Comput. Sci., Univ. of North Carolina at Chapel Hill, Chapel Hill, NC.

[2]J. McLean, "A formal foundation for trace specification," in *Information Processing Systems*, Code 7592, NRL, Washington, DC 20375.

C. Example: Merge

A process *merge* has two input channels, in_1, in_2 and an output channel *out*. *merge* expects to receive positive monotone increasing integers along each of in_1 and in_2; *merge* performs an on-the-fly merge of the received sequences and sends the resulting monotone increasing sequence along *out*.

1) merge:

```
$[in_1?x \| in_2?y]$; {wait to receive in parallel along $in_1$ and $in_2$}
$*[x < y \rightarrow out!x; in_1?x$
 $\Box y < x \rightarrow out!y; in_2?y$
 $\Box x = y \rightarrow out!x; [in_1?x \| in_2?y]$
 ]
```

Let $mi(Z)$ denote that Z is a monotone increasing sequence. A specification for *merge* is the following.

2) $mi(Zin_1),\ \ mi(Zin_2)|merge|mi(Zout),\ \ Zout \subseteq Zin_1 \cup Zin_2$, where $Zout \subseteq Zin_1 \cup Zin_2$ means that the set of values of the output sequence is a subset of the inputs. This specification states that if inputs along both channels are monotone increasing, then the output is also monotone increasing and is a subset of the inputs.

D. Inference Rules

1) Rule of Network Composition: This rule allows us to deduce the internal specification of a network H from the external specification of its component processes h_i, $i = 1, 2, \cdots$

$$\frac{r_i|h_i|s_i, \ \text{all } i}{(\underset{i}{\text{and}}\ r_i)\ [H]\ (\underset{i}{\text{and}}\ s_i)}.$$

The validity of this rule follows easily by considering any network trace where $(\underset{i}{\text{and}}\ r_i)$ holds up to the kth point in the trace and therefore from the hypothesis, $(\underset{i}{\text{and}}\ s_i)$ holds up to the $(k + 1)$th point, for every $k \geqslant 0$.

2) Rule of Inductive Consequence: This consists of two parts relating to the implication on precondition and implication on postcondition.

$$\frac{(s \text{ and } r) \Rightarrow r',\ \ r'[h]s}{r[h]s} \tag{1}$$

$$\frac{r[h]s',\ \ s' \Rightarrow s}{r[h]s}. \tag{2}$$

The validity of (2) is easy to see. Equation (1) can be established by induction on the length of the internal trace: we show that s holds up to the $(k + 1)$th point in any trace provided r holds up to the kth point of the trace, $k \geqslant 0$. s holds initially from $r'[h]s$. Inductively, if (s and r) hold up to the kth point of any trace, then r' holds up to that point and hence it follows from $r'[h]s$ that s holds up to the $(k + 1)$th point of the trace.

3) Rule of Abstraction: These rules allow us to construct an external specification from an internal specification and vice versa. These rules are easily established.

$$\frac{r|h|s}{r[h]s} \tag{3}$$

$$\frac{r[h]s;\ \ r, s \text{ are assertions on external traces of } h}{r|h|s}. \tag{4}$$

Equation (3) states that any external specification of h can be treated as an internal specification. Equation (4) states that any internal specification can be treated as an external specification provided that the internal specification does not specify transmission along internal channels. In such a case all internal communications of the network are hidden in the same way that a value of a local variable is hidden outside a block.

E. Theorem of Hierarchy

This theorem allows us to go directly from the external specifications of component processes h_i's of a network H to the external specification of H. The theorem follows directly by the application of the inference rules. This theorem has been used extensively in proving the examples in this paper.

Theorem of Hierarchy:

$$\frac{\text{For all } i, r_i|h_i|s_i;\ \ (S \text{ and } R_0) \Rightarrow R, S \Rightarrow S_0}{R_0|H|S_0}$$

where R_0, S_0 are assertions on the external trace of H, and R, S, denote $\underset{i}{\text{and}}\ r_i$, $\underset{i}{\text{and}}\ s_i$, respectively.

IV. Examples

A. Network of Merge Processes

Consider a network H of *merge* processes (see Section III-C), h_1, h_2, h_3, as shown in Fig. 1.

We are given the following specification for each *merge* process with input channels in_1 and in_2 and output channel *out*.

$$mi(Zin_1), mi(Zin_2)|merge|mi(Zout), Zout \subseteq Zin_1 \cup Zin_2.$$

We wish to prove $R_0|H|S_0$ for the network H, where

$$R_0 = mi(Zin_{k,j}), \quad \text{all } k, j = 1, 2$$

$$S_0 = mi(Zout_3), Zout_3 \subseteq \bigcup_{k,j=1,2} Zin_{k,j}.$$

From the definition

$$S = mi(Zout_1),\ mi(Zout_2),\ mi(Zout_3),\ Zout_1 \subseteq Zin_{1,1} \cup Zin_{2,1},\ Zout_2 \subseteq Zin_{1,2} \cup Zin_{2,2},\ Zout_3 \subseteq Zout_1 \cup Zout_2$$

$$R = mi(Zin_{k,j}), k, j = 1, 2;\ \ mi(Zout_1),\ \ mi(Zout_2)$$

it is obvious that

$$S \text{ and } R_0 \Rightarrow R, S \Rightarrow S_0.$$

Hence, we have $R_0|H|S_0$, from the theorem of hierarchy.

B. Computing Odd Primes (Adapted From Hoare [8])

1) Description of the Network: The network H consists of one input channel m_1 from the environment and one output channel *output* to the environment. H receives the se-

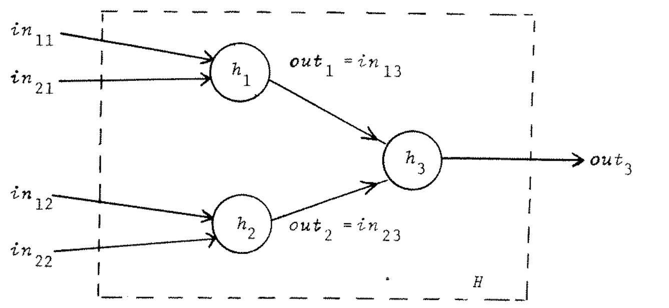


Fig. 1. A network of *merge* processes.

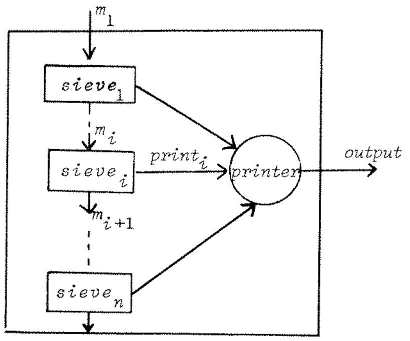


Fig. 2. Schematic representation of the network H to compute odd primes.

quence of all odd integers greater than or equal to 3 in increasing order; H outputs the sequence of primes greater than or equal to 3. Hence, it is required to show the following.

a) $Zm_1 \underline{\propto} odd|H|Zoutput \underline{\propto} oddprime$

where

odd is the sequence of all odd integers $\geqslant 3$ in increasing order,

oddprime is the sequence of all primes $\geqslant 3$ in increasing order.

The structure of the network is shown schematically in Fig. 2.

H consists of two types of processes: *sieve* and *printer*. In order to simplify description, we assume that there are an infinite number of *sieve* processes, designated by $sieve_1$, $sieve_2 \cdots sieve_i \cdots$. $sieve_i$ has one input channel m_i by which it receives input from $sieve_{i-1}$ (environment, if $i = 1$). Channels m_i, $i = 2, 3 \cdots$ are internal to H, but m_1 is incident on and directed towards H. $sieve_i$ has two output channels m_{i+1} and $print_i$. The latter is directed toward a *printer* process of which there is exactly one in H. The *printer* process has one output channel *output*, which is the only output channel incident on H.

2) Description of $sieve_i$: The very first message p received by $sieve_i$ is sent on to the *printer* process. Every subsequent message x received by $sieve_i$ is checked to see if it is a multiple of p (see Hoare [8] for a clever algorithm due to Gries to implement this); if x is not a multiple of p it is sent on to $sieve_{i+1}$, if x is a multiple it is discarded. $sieve_i$ assumes that it receives a monotone increasing sequence of positive integers relatively prime to the first i primes. Given this assumption $sieve_i$ asserts that it outputs a monotone increasing sequence of positive integers, relatively prime to the first $(i + 1)$ primes. See the Appendix for a detailed description and proof of $sieve_i$.

Note that the first item p received along m_i is the ith prime greater than or equal to 3, under the given assumption.

3) Specification of $sieve_i$: Let seq_j be the monotone increasing sequence of positive integers relatively prime to the first j primes (i.e., relatively prime to the first $(j - 1)$ odd primes).

Let $Z^{(i)}$ denote the ith element of sequence Z.

For $sieve_i$ we have: $r|sieve_i|s$, where

r: $Zm_i \underline{\propto} seq_i$

s: 1) $Zm_{i+1} \underline{\propto} seq_{i+1}$

2) $Zprint_i \equiv empty$ or $Zprint_i \equiv \langle oddprime^{(i)} \rangle$

3) $\overline{Zm_i} = 0 \Rightarrow Zprint_i \equiv empty$

4) $\overline{Zm_{i+1}} > 0 \Rightarrow Zprint_i \equiv \langle oddprime^{(i)} \rangle$

$sieve_i$ assumes r, i.e., its input is a monotone increasing sequence of positive integers relatively prime to the first i primes. It establishes that: 1) its output is a monotone increasing sequence of positive integers, relatively prime to the first $(i + 1)$ primes, 2) at most one message is sent to the *printer*, which is the ith odd prime, and 3), 4) upon receiving the first message, $sieve_i$ will send that message on to the *printer* and sends subsequent messages (if any) to $sieve_{i+1}$.

4) Description of Printer: The *printer* process waits to receive input along all input channels. Upon receiving an input, it sends the received value along *output*. This continues indefinitely.

5) A Specification of the Printer: The specification uses assertions on the external trace $\langle (C_1, v_1), \cdots (C_i, v_i) \cdots \rangle$ of the printer.

r: *true*

s: 1) $C_{2i-1} = print_j$,	for some j,	$i = 1, 2, \cdots$
2) $C_{2i} = output$,		$i = 1, 2, \cdots$
3) $v_{2i-1} = v_{2i}$,		$i = 1, 2, \cdots$.

6) Network Proof: We wish to show $R_0|H|S_0$, where R_0 is $Zm_1 \underline{\propto} odd$, S_0 is $Zoutput \underline{\propto} oddprime$. Let $rsieve_i$, $ssieve_i$ denote the r, s associated with $sieve_i$.

a) S and $R_0 \Rightarrow R$, since $R_0 \Rightarrow rsieve_1$; $ssieve_{i-1}$

$\Rightarrow rsieve_i, i > 1$.

b) We next show that $S \Rightarrow S_0$.

Given S, $Zprint_i \equiv empty \Rightarrow \overline{Zm_{i+1}} = 0$, from $ssieve_i$.

$\overline{Zm_{i+1}} = 0 \Rightarrow Zprint_{i+1} \equiv empty$, from $ssieve_{i+1}$.

Therefore, $Zprint_i \equiv empty \Rightarrow Zprint_j \equiv empty, j > i$. For any trace $\langle (C_1, v_1), (C_2, v_2) \cdots (C_i, v_i) \cdots \rangle$ of the *printer*, we have therefore

$\langle v_{2i-1} \rangle \equiv Zprint_i \equiv \langle oddprime^{(i)} \rangle$,

from $ssieve_i$ and above observation,

$v_{2i-1} = v_{2i}$, from s of *printer*,

$Zoutput \equiv \langle v_2, v_4, v_6 \cdots \rangle \underline{\propto}\, oddprime.$

The required proof follows from a) and b) using the theorem of hierarchy.

C. A Network for Computing Factorial Streams: Demonstrating Hierarchical Decomposition

Consider a process H, having one incident input channel x_1 and one incident output channel w_1. The process receives a stream of nonnegative integers along x_1. The process delivers a stream of factorials of the input sequence along w_1.

1) Description of the Network: H is a network consisting of an infinite number of processes $CP_1 \cdots CP_i \cdots$, as shown in Fig. 3. The assumption of an infinite number of processes is for brevity in exposition—it is sufficient to assume that there are more processes than the largest integer received by H.

The process CP_i has two input channels x_i, w_{i+1} and two output channels x_{i+1}, w_i. CP_i receives a stream of nonnegative integers (one at a time) along x_i. It delegates the responsibility of computing the factorial of the next lower number to CP_{i+1} (if the number is positive) by sending the next lower number to CP_{i+1} along x_{i+1}. It receives the response from CP_{i+1} via w_{i+1} and produces its own output along w_i. The operations are asynchronous in that many inputs may be read before any output is produced. There is parallelism in this computation since various CP's may be working on computing the factorials of different inputs.

2) Proof of the Network—Notations: If Z is a sequence of integers

$0 \leqslant Z$ denotes that each element of Z is nonnegative,

$Z!$ denotes the sequence in which each element is the factorial of the corresponding element of Z,

$red(Z)$ denotes the sequences obtained by deleting all 0's from Z and decrementing all other numbers by 1.

The external specification of H is

$$0 \leqslant Zx_1 | H | Zw_1 \underline{\propto} Zx_1!$$

Let $r_i | CP_i | s_i$ be the external specification of CP_i, $i \geqslant 1$, where

$$r_i :: 0 \leqslant Zx_i, Zw_{i+1} \underline{\propto} Zx_{i+1}!$$

$$s_i :: Zx_{i+1} \underline{\propto} red(Zx_i), Zw_i \underline{\propto} Zx_i!$$

The proof of H follows from the theorem of hierarchy and the external specifications of the CP_i: (S **and** $0 \leqslant Zx_1) \Rightarrow R$, $S \Rightarrow (Zw_1 \underline{\propto} Zx_1!)$.

3) Description of CP_i—The Next Refinement Step: CP_i is again a network of processes, schematically depicted in Fig. 4.

For notational convenience, we drop the subscripts in x_i, w_i, x_{i+1}, w_{i+1} and refer to them as x, w, t, e, as shown in Fig. 4.

CP consists of five processes: *in*, *out*, *ba*, *bd*, *bu*. Each of *ba*, *bd*, *bu* is a buffer process of the type described in Section III-B. The process *in* executes the following loop indefinitely. It receives a nonnegative integer dx along channel x. If dx is positive send $dx - 1$ along channel y to buffer *bd*. Next send dx along channel r to buffer *ba*.

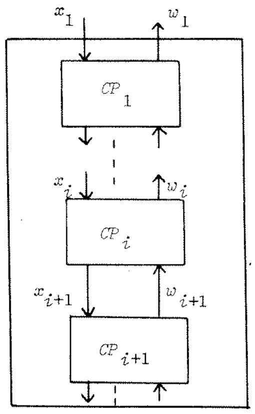


Fig. 3. A schematic representation of the network H.

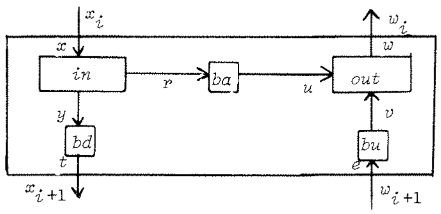


Fig. 4. A schematic of CP_i.

The process *out* executes the following loop indefinitely. It receives a nonnegative integer du along channel u from buffer *ba*. If du is 0, next send 1 (one) along channel w, completing the loop. If du is positive, next wait to receive an integer dv along channel v, and after receiving dv send out the product of du and dv along channel w, thus completing the loop.

It is unfortunate that *CSP* uses "!" as a symbol for sends, as this conflicts with the use of the same symbol for factorial function. It should be obvious from the context which usage is meant; for instance "!" stands for *sends* in the following program.

The process *in* is described by the following program.

```
in ::
   *[true → {loop forever}
      x?dx;
      [dx = 0 → skip
      □ dx ≠ 0 → y!(dx-1)
      ];
      r!dx
   ]
```

The process *out* is described by the following program.

```
out ::
   *[true → {loop forever}
      u?du;
      [du = 0 → w!1
      □ du ≠ 0 → v?dv; w!du*dv
      ]
   ]
```

4) Proof of CP: It is required to show $R_0|CP|S_0$, i.e.,

$$(0 \leqslant Zx, Ze \underline{\propto} Zt!)|CP|(Zt \underline{\propto} red(Zx), Zw \underline{\propto} Zx!).$$

We can show the following:

$r_i|h_i|s_i$, for all i, where r_i, h_i, s_i are as given in Table I.

Then S **and** $R_0 \Rightarrow R$ (shown in Section IV-C5).

$S \Rightarrow S_0$, trivially.

The result follows from the theorem of hierarchy.

The only nontrivial proof for component sequential processes of a *CP* is for process *out*, which appears in Section VII-C. We leave the proofs of other component processes to the reader. We now show that S **and** $R_0 \Rightarrow R$.

5) Proof of S and $R_0 \Rightarrow R$:

S **and** $R_0 \Rightarrow 0 \leqslant Zx$, trivially.

We next show S **and** $R_0 \Rightarrow Zv \underline{\propto} red\ (Zu)!$ **and** $0 \leqslant Zu$.

$Zv \underline{\propto} Ze, Ze \underline{\propto} Zt!, Zt \underline{\propto} Zy \underline{\propto} red(Zx)$ (from S **and** R_0).

Hence,

1) $Zv \underline{\propto} red(Zx)!$,
2) $Zu \underline{\propto} Zr \underline{\propto} Zx$ (from S),
3) since $0 \leqslant Zx$, using 2) we have $0 \leqslant Zu$,
4) also from 2), $red(Zu)! \underline{\propto} red(Zx)!$,
5) $\overline{red(Zu)} \geqslant \overline{Zv}$ (from S),
6) from 1), 4), 5), $Zv \underline{\propto} red(Zu)!$

The result follows from 3) and 6).

V. A Model of Communication with Parameters

We present a model in which a process may communicate with its environment by parameters as well as by messages. We show how parameters are passed to the process at process instantiation and how results are returned at termination. As Hoare has shown [8], the notion of a message communicating process is very general in that it can be used to model a number of well-known programming language constructs. We show that augmenting a process by the addition of parameter passing capability naturally extends Hoare's parallel command to include procedure calls and to allow recursion. The previous sections are restricted to the case where no parameter is passed; we now show simple extensions to the inference rules to handle parameter passing. Details of this model may be found in [4]; we present an abbreviated version here, merely to explain the new inference rules.

A. Procedure and Process

A process which communicates only via parameters is a *procedure*. It is invoked by a *call* and passed parameters. Its invoker is suspended until the procedure terminates. At termination the results of the procedure's computation are returned via parameters to its invoker.

A procedure which is implemented as a sequential process is well understood. A procedure which is implemented as a process which is a network is also *called* and passed parameters. The call results in the *instantiation* of the process. A process

TABLE I
Specification of Component Processes of *CP*

r	h	s
$0 \leqslant Zx$	*in*	$Zr \underline{\propto} Zx, Zy \underline{\propto} red(Zx)$
$Zv \underline{\propto} red(Zu)!, 0 \leqslant Zu$	*out*	$\overline{red(Zu)} \geqslant \overline{Zv}, Zw \underline{\propto} Zu!$
true	*ba*	$Zu \underline{\propto} Zr$
true	*bd*	$Zt \underline{\propto} Zy$
true	*bu*	$Zv \underline{\propto} Ze$

implemented as a network is instantiated by instantiating its component processes and passing them parameters as follows. We only permit *value* and *result* parameters. Value parameters cannot be altered by any process; thus, a process which is passed a value parameter may pass it on to an arbitrary number of component processes as value parameters. Result parameters are *partitioned* among component processes, i.e., each result parameter passed to a process must be passed to one and only one component process. A sequential process is instantiated by creating a fresh copy of it and passing it parameters.

During the lifetime of a process, it communicates exclusively via messages; it treats value parameters as constants and result parameters as local variables.

A sequential process terminates in conventional manner by executing the commands up to the end of its program; a network terminates when all its component processes terminate. Upon termination of a procedure, its result parameters are passed back to its invoker. Upon termination of a process which is a component of a network, the result parameters are passed back to the network. The network cannot alter these result parameters; it simply passes them on to its instantiator upon termination.

A procedure may be written using recursion even though it is implemented as a network of processes. For example, a procedure H may consist of processes $h_1, \cdots, h_n$ and any component process h_i may call H in turn, resulting in a fresh instantiation of procedure H.

B. Specification Mechanism and Inference Rules

1) External Specification: We use four assertions to specify each process: two assertions p, q corresponding to parameters as in sequential program proving and two assertions r, s corresponding to message transmission, as described earlier in this paper.

r, s name only the external trace of h and constants (including value parameters) and p, q name only the external trace, result parameters and constants (including value parameters).

$\{r; p\}\ |h|\ \{q; s\}$ denotes that:

1) if p holds initially in h then s holds initially in h,

2) if p holds initially in h and r holds up to the kth point in any trace of h, then s holds up to the $(k+1)$th point of that trace for all $k \geqslant 0$, and

3) if p holds initially in h and r holds at all times during the life of h and h terminates, then q holds on termination.

The reader may verify the following specializations. If r and s are absent (or ***true***), we can drop them and write

$$\{p\}\ h\ \{q\}$$

which then has its usual meaning as in sequential program proving [7]. If p, q are absent (or ***true***), we can drop those and write

$$r|h|s$$

which has the meaning described earlier in this paper.

2) Internal Specification: The internal specification for a process h, denoted by $\{r;p\}\ [h]\ \{q;s\}$, is identical to that given in Section V-B1 except that r, s, p, q refer to the internal trace.

3) Inference Rules: The inference rules and the theorem of hierarchy for the general model are the obvious extensions of those given in Section III-D, using the rule of consequence from sequential program proving for component assertions p, q. We show below one inference rule and the theorem of hierarchy.

Rule of Network Composition: Let H be a network with component processes $h_1 \cdots h_i \cdots$

$$\frac{\{r_i;p_i\}\ |h_i|\ \{q_i;s_i\},\ \text{for all } i}{\{R;P\}\ [H]\ \{Q;S\}}$$

where R, P, Q, S denote $(\underset{i}{\text{and}}\ r_i)$, $(\underset{i}{\text{and}}\ p_i)$, $(\underset{i}{\text{and}}\ q_i)$, $(\underset{i}{\text{and}}\ s_i)$, respectively.

Theorem of Hierarchy: For a network H, with component processes $h_i, i = 1, 2, \cdots$

$$\frac{\text{for all } i,\ \{r_i;p_i\}\ |h_i|\ \{q_i;s_i\}; S \text{ and } R_0 \Rightarrow R; S \Rightarrow S_0; P_0 \Rightarrow P; Q \Rightarrow Q_0}{\{R_0;P_0\}\ |H|\ \{Q_0;S_0\}}$$

where R, P, Q, S denote $(\underset{i}{\text{and}}\ r_i)$, $(\underset{i}{\text{and}}\ p_i)$, $(\underset{i}{\text{and}}\ q_i)$, $(\underset{i}{\text{and}}\ s_i)$, respectively.

VI. An Example Employing the General Model

The following problem and its solution first appeared in Dijkstra [5]; a formal proof of the solution also appears in [1].

A. Problem Description

It is required to design a procedure *partition* (u_0, v_0, u, v), where u_0, v_0 are value parameters, u, v are result parameters and all parameters are sets of integers. It is given that u_0 and v_0 have no common element. *Partition* is required to compute u, v such that $\bar{u} = \bar{u}_0$, $\bar{v} = \bar{v}_0$ (where $\bar{A}$ denotes the size of set A), $u \cup v = u_0 \cup v_0$ and $max(u) < min(v)$. Formally, the specification for *partition* is

1) $\{u_0 \cap v_0 = \emptyset\}$ *partition* (u_0, v_0, u, v) $\{\bar{u} = \bar{u}_0, \bar{v} = \bar{v}_0$
$u \cup v = u_0 \cup v_0, max(u) < min(v)\}$.

The solution to this problem employs two processes, *small* and *large*, where *small* is passed u_0 as a value parameter and returns u as the result; *large* is passed v_0 as a value parameter and returns v as the result. *small* and *large* communicate via two channels *hi* and *lo*, as shown in Fig. 5. The programs for *small* and *large* are shown next.

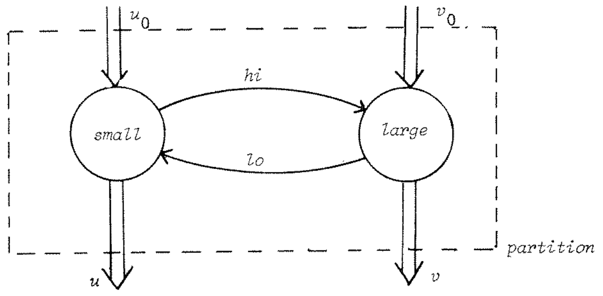


Fig. 5. Schematic of *partition.*

2)

small (u_0, u)	*large* (v_0, v)
$u := u_0;$	$v := v_0;$
$mn := -\infty; mx := +\infty;$	$mn := -\infty; mx := +\infty;$
$*[mn < mx \rightarrow$	$*[mn < mx \rightarrow$
$mx := max(u);$	$hi\ ?\ mx;$
$hi\,!\,mx;$	$v := v \cup \{mx\};$
$u := u - \{mx\};$	$mn := min(v);$
$lo\ ?\ mn;$	$lo\,!\,mn;$
$u := u \cup \{mn\}$	$v := v - \{mn\}$
$]$	$]$

B. Proof of Partition

We use $last(Z)$ to denote the last element of a sequence Z; it is undefined if Z is *empty*. *Zlo* and *Zhi* would be used both as sets and sequences; the usage will be evident from the context.

1) Specification of small: *small* assumes that the numbers in the sequence *Zlo* are from $u_0 \cup v_0$ and that u_0, v_0 are disjoint. It establishes that all numbers in its output sequence *Zhi* are from $u_0 \cup v_0$, and if it terminates, the last number received along *lo* before termination is $max(u)$. Formally, we can establish $\{r;p\}|small|\ \{q;s\}$, where

r :: $Zlo \subseteq u_0 \cup v_0$

s :: $Zhi \subseteq u_0 \cup v_0$

p :: $u_0 \cap v_0 = \emptyset$

q :: $last(Zlo) = max(u), \bar{u} = \bar{u}_0, u \subseteq u_0 \cup v_0$.

2) Specification of large: Similarly, for *large*

r :: $Zhi \subseteq u_0 \cup v_0$

s :: $Zlo \subseteq u_0 \cup v_0$

p :: $u_0 \cap v_0 = \emptyset$

q :: $last(Zlo) < min(v), \bar{v} = \bar{v}_0, v \subseteq u_0 \cup v_0$.

3) Proof of Correctness and Termination: The desired result follows trivially from the theorem of hierarchy.

We next show how termination can be proven for the *individual* processes. We define a metric $\|w\|$ for a set of integers w as

$$\|w\| = \sum_{x \in w} x.$$

Note that $\|u\|$ and $\|v\|$ are both bounded above and below, since their elements are obtained from the finite sets u_0 and v_0. It is trivially shown in *small* (*large*) that $\|u\|$ ($\|v\|$) increases (decreases) in every iteration but the last by showing that if $\|u'\|$ ($\|v'\|$) is the value of $\|u\|$ ($\|v\|$) at the start of an iteration, then $\|u'\| + mn - mx$ ($\|v'\| - mn + mx$) is the value of $\|u\|$ ($\|v\|$) at the end of that iteration, and $mn < mx$ except for the last iteration.

VII. Proving $r|h|s$ for Sequential Process h

A. *Proof Rules for Message Transmission Commands*

Let $tr(h)$ denote trace of h. Initially, $tr(h) = empty$.

1) The message transmission command $e!x$ in h, has the same semantics as

$$tr(h) := tr(h) | <(e, x)>.$$

2) The command $e?x$ in h, has the following semantics.

a) $\{tr(h) = tr^0\}\ e?x\ \{tr(h) = tr^0 | <(e, x)>\}$ and

b) $\{p\}\ e?x\ \{p\}$, provided p has no free occurrence of $tr(h)$ or x.

Note that in case b) if p is an assertion containing a free occurrence of x, we cannot assert anything following the receipt of x.

B. *Proving $r|h|s$*

For a sequential process h, $r|h|s$ may be shown by using only sequential program proving ideas. An *annotated program* is one in which the program text is interleaved with assertions at appropriate places. Every statement T in an annotated program has a unique precondition $pre(T)$ and a unique postcondition $post(T)$, such that

$$\{pre(T)\}\ T\ \{post(T)\}$$

is deducible in the axiom system under consideration. If there are two adjacent assertions P_1, P_2 in the annotated program, then P_1 must imply P_2. We use the system of Hoare [7] augmented with the proof rules in Section VII-A for message transmission.

The reader is referred to Owicki and Gries [13] for a clear description of sequential program annotation.

In order to prove $r|h|s$, we need more than a conventional annotation of h; we need to introduce the assertion r at proper program points.

1) Proof Steps:

a) Initially assert that the trace of h is *empty* (hence all incident sequences on h are *empty*).

b) Show that s holds initially in h.

c) Assert r at all program points. (Note: Since r is affected only by message transmission commands, it is sufficient to assert r prior to and following every message transmission command.)

d) Construct an annotation of h.

e) For every message transmission command M prove that

$$\{pre(M)\}\ M\ \{s\}$$

where $pre(M)$ is the precondition of M in the annotation. Note that $pre(M)$ is obtained from the annotation constructed in step d, however, this proof step cannot assert s from the annotation of step d), s must be proven using the proof rules of Section VII-A.

Intuitively, for any statement A, $pre(A)$ constructed by the above procedure denotes the condition that holds immediately before execution of A assuming that: 1) the trace of h is initially *empty* and 2) r holds at all points in computation *prior* to the execution of A.

2) Proving $\{r; p\}\ |h|\ \{q; s\}$: The procedure for proving $\{r; p\}\ |h|\ \{q; s\}$ is similar to the procedure outlined in Section VII-B1. Step a) will be replaced by a$'$) and step f) is added at the end.

a$'$) Initially assert that the trace of h is *empty* and p holds.

f) Show that q holds if and when h terminates.

C. *Example (Program out, Section IV-C3)*

For the program *out* of Section IV-C3, we show $r|out|s$, where

$$r :: Zv \underline{\propto} red(Zu)!, \qquad 0 \leqslant Zu$$

$$s :: \overline{red(Zu)} \geqslant \overline{Zv}, \qquad Zw \underline{\propto} Zu!$$

s holds initially since all sequences are *empty*. We show an annotated program below where only the program points, following receipt of a message, have been annotated with r since these are the only points where the assertion is needed.

1) Annotated Program:

```
  $\{Zu \equiv empty, Zv \equiv empty, Zw \equiv empty\}$
  $\{Zv \equiv red(Zu)!, Zw \equiv Zu!$: vacuously$\}$
 $*[true \rightarrow$
  $\{Zw = Zu!, Zv \equiv red(Zu)!\}$
  $u?du;$
  $\{Zw|du! = Zu!, du = 0 \Rightarrow Zv \equiv red(Zu)!,$
   $du > 0 \Rightarrow Zv|(du-1)! \equiv red(Zu)!$
    : from $r$ we can deduce here that $0 \leqslant du\}$
  $[du = 0 \rightarrow \{Zw|0! \equiv Zu!, Zv \equiv red(Zu)!\}$
             $w!1$
             $\{Zw \equiv Zu!, Zv \equiv red(Zu)!\}$
  $\Box du \neq 0 \rightarrow \{du > 0, Zw|du! \equiv Zu!,$
              $Zv|(du-1)! \equiv red(Zu)!\}$
              $v?dv;$
             $\{Zw|du! \equiv Zu!, Zv \underline{\propto} red(Zu)!,$  from $r$,
              therefore, from precondition of this
               statement, $dv = (du-1)!$
              and $Zv \equiv red(Zu)!\}$
             $w!du*dv$
            $\{Zw \equiv Zu!, Zv \equiv red(Zu)!\}$
   ]
 $\{Zw \equiv Zu!, Zv \equiv red(Zu)!\}$
]
```

We next have to show for every message transmission command M, $\{pre(M)\}\ M\ \{s\}$, i.e., we have to prove the following four steps:

$$\{Zv \equiv red(Zu)!, Zw \equiv Zu!\}\ u?du\ \{\overline{red(Zu)} \geqslant \overline{Zv}, Zw \underline{\propto} Zu!\},$$
$$\{Zw|0! \equiv Zu!, Zv \equiv red(Zu)!\}\ w!1\ \{\overline{red(Zu)} \geqslant \overline{Zv}, Zw \underline{\propto} Zu!\},$$
$$\{du > 0, Zw|du! \equiv Zu!, Zv|(du-1)! \equiv red(Zu)!\}\ v?dv\ \{\overline{red(Zu)} \geqslant \overline{Zv}, Zw \underline{\propto} Zu!\},$$
$$\{Zv \equiv red(Zu)!, Zw|du! \equiv Zu!, dv = (du-1)!\}\ w!du*dv\ \{\overline{red(Zu)} \geqslant \overline{Zv}, Zw \underline{\propto} Zu!\}.$$

The proof of each one of these is straightforward using proof rules for message transmission commands of Section VII-A.

VIII. Related Work

Apt, Francez, and de Roever [1] and Levin [12] have proposed techniques for proofs of communicating sequential processes described in *CSP*. Both techniques associate *arbitrary* pre- and post-condition with every message transmission command in every process. If $\{P\}\ A?x\ \{Q\}$ has been asserted in process B and $\{R\}\ B!e\ \{S\}$ has been asserted in process A and *if these two commands lead to a communication*, then it must be shown that

$$\{P \textbf{ and } R\}\ x := e\ \{Q \textbf{ and } S\}.$$

The proof system would be too weak if it is necessary to prove this for *every pair of commands* as above. It must be realized that certain pairs of commands cannot lead to a communication.

Levin introduces auxiliary variables common to all processes so that if (P **and** R) is *false*, then the pair of commands cannot lead to a communication. It becomes necessary then in Levin's system to construct a noninterference proof so that the assertions made about auxiliary variables in one process may not be falsified in another process.

Apt, Francez, and de Roever handle this problem by introducing a global invariant (which must be shown to be an invariant). The global invariant can then be used to eliminate those pairs of commands that cannot lead to a communication. Our proof method differs in the following ways from these schemes.

1) We insist upon a specification mechanism for processes and autonomous process proofs.

2) We do not allow arbitrary pre- and post-conditions for message transmission commands. We *prove* $r|h|s$ for every process h autonomously.

3) We permit no network-wide auxiliary variables or global invariants. Every process can use its own trace as an auxiliary variable, which by definition cannot be modified by another process without cooperation of this process.

Keller [11] has proposed proving an invariant property of a network by showing that the property holds initially and following each "firing," i.e., following each message transmission in our model. We also use induction on the number of message transmissions as the basis of our proof technique; however, our emphasis is on autonomous process proofs, which for component processes are combined to yield a network proof.

Trace is an important auxiliary variable in our proof technique; uses of "history sequence" have appeared explicitly in [6] and [10]. The word "trace" seems to have originated with the work of Parnas in connection with sequential program verification.

IX. Discussion

Our proof system has the following features.

A. Autonomy

A process is an independent entity much like a procedure and therefore should have an independent specification. A process in a network should be replaceable by another process having the same specification. It should only be necessary to prove an implementation by showing that it is consistent with its specification. This approach then requires us to use the specifications (but not the code) of the component processes in proving a network.

B. Hierarchy

A hierarchy of processes is constructed in the following manner. The leaves of the hierarchy tree, i.e., the lowest nodes, are sequential processes. The internal and external specifications for a sequential process h are identical, i.e., $r|h|s = r[h]s$. A network of processes can be specified either with an internal or an external specification. The internal specification considers the network trace and may be used to state invariant properties of the network; the external specification restricts attention to the external trace, i.e., the external behavior of the network viewed as a single process. This external specification is the only specification that can be used in proofs at the next higher level in the hierarchy. The proof system supports the hierarchy in a natural manner.

C. Compatibility with Sequential Program Proving Techniques

The technique proposed is a natural extension of the axiomatic system of Hoare [7] for sequential programs. For instance, in the absence of message communication in a process h, h reduces to a procedure and proof of $\{r;p\}\ |h|\ \{q;s\}$ reduces to a conventional sequential proof of $\{p\}\ h\ \{q\}$. Hence, the external specification of h uses only p, q in the traditional manner, although h may be implemented by a network of processes.

D. Limitations

One limitation of our proposed system, much like Hoare's axiomatic system for sequential programs, is the inability to prove temporal properties such as eventual deadlock, or eventual termination, etc., within the proof system directly. We suspect that proofs of temporal properties can never be achieved *directly* using only the notion of invariants; instead a metric must be associated with the program which is shown to decrease as the computation progresses. Examples of such metrics may be found in [3], which can be used in conjunction with methods of this paper to prove absence of deadlock.

Appendix

Description and Proof of $sieve_i$ (See Sections IV-B2 and IV-B3)

It is required to show $r_i | sieve_i | s_i$, where

1) $r_i :: Zm_i \underline{\propto} seq_i$

$s_i :: Zm_{i+1} \underline{\propto} seq_{i+1}, Zprint_i \equiv empty$

or $Zprint_i \equiv \langle oddprime^{(i)} \rangle$

$\overline{Zm_i} = 0 \Rightarrow Zprint_i \equiv empty$

$\overline{Zm_{i+1}} > 0 \Rightarrow Zprint_i \equiv \langle oddprime^{(i)} \rangle$

```
2) sieve_i ::
  m_i?p; print_i!p;
  mp := p;
    *[true →
      m_i?x;
        *[x > mp → mp := mp + p];
         [x = mp → skip
         □x < mp → m_{i+1}!x
         ]
    ]
```

The proof can be accomplished by using the following invariants. In the following, $cut(Zm_i)$ denotes the sequence obtained by removing all mulitples of $Zm_i^{(1)}$ from Zm_i.

$I :: Zm_i \underline{\propto} seq_i, cut(Zm_i) \equiv Zm_{i+1}, Zprint_i \equiv \langle oddprime^{(i)} \rangle,$

p *divides* $mp, mp - p < last(Zm_i)$

$I' :: x = last(Zm_i), Zm_i \underline{\propto} seq_i,$

p *divides* $x \Rightarrow cut(Zm_i) \equiv Zm_{i+1},$

not p *divides* $x \Rightarrow cut(Zm_i) \equiv Zm_{i+1} | \langle x \rangle,$

$Zprint_i \equiv \langle oddprime^{(i)} \rangle,$

p *divides* $mp, mp - p < x$

The annotation uses I as the outer loop invariant and I' as the inner loop invariant.

Acknowledgment

The authors gratefully acknowledge the helpful comments of Profs. S. Owicki and G. Levin. They are deeply grateful to Prof. C. A. R. Hoare for his encouragement. Prof. D. Parnas has kindly brought some important early work on trace to their attention.

References

[1] K. R. Apt, N. Francez, and W. P. de Roever, "A proof system for communicating sequential processes," *TOPLAS*, vol. 2, July 1980.

[2] M. Clint, "Program proving: Coroutines," *Acta Inform.*, vol. 2, 1973.

[3] K. M. Chandy and J. Misra, "Deadlock absence proofs for networks of communicating processes," *Inform. Process. Lett.*, vol. 9, no. 4, 1974.

[4] —, "A simple model of distributed programs based on implementation hiding and process autonomy," *SIGPLAN Notices*, July 1980.

[5] E. W. Dijkstra, "A correctness proof for communicating processes: A small exercise," EWD607, The Netherlands.

[6] D. I. Good, R. M. Cohen, and J. Keeton-Williams, "Principles of proving concurrent programs in GYPSY," in *Conf. Rec. 6th Annu. ACM Symp. on Principles of Programming Lang.*, San Antonio, TX, Jan. 1979.

[7] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. Ass. Comput. Mach.*, vol. 12, 1969.

[8] —, "Communicating sequential processes," *Commun. Ass. Comput. Mach.*, vol. 21, no. 8, 1978.

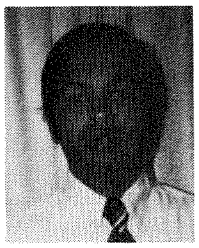
[9] —, "A model for communicating sequential processes," Comput. Lab., Oxford Univ., Dec. 1978.

[10] J. H. Howard, "Proving monitors," *Commun. Ass. Comput. Mach.*, vol. 19, no. 5, 1976.

[11] R. M. Keller, "Formal verification of parallel programs," *Commun. Ass. Comput. Mach.*, vol. 19, no. 7, 1976.

[12] G. M. Levin, "A proof technique for communicating sequential processes (with an example)," TR 79-401, Dep. Comput. Sci., Cornell Univ., Ithaca, NY, 1979.

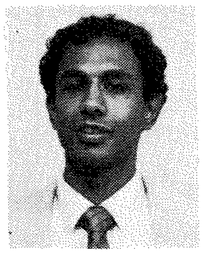
[13] S. Owicki and D. Gries, "An axiomatic proof technique for parallel programs," *Acta Inform.*, vol. 6, 1976.



Jayadev Misra (S'71–M'72) received the B.Tech. degree in electrical engineering from the Indian Institute of Technology, Kanpur, in 1969, and the Ph.D. degree in computer science from The Johns Hopkins University, Baltimore, MD, in 1972.

He worked for IBM, Federal System Division, from January 1973 to August 1974. He is currently an Associate Professor with the Department of Computer Sciences, University of Texas, Austin.

Dr. Misra is a member of the Association for Computing Machinery.



K. Mani Chandy received the B.Tech. degree in electrical engineering from the Indian Institute of Technology, Madras, India, the M.S. degree in electrical engineering from the Polytechnic Institute of Brooklyn, Brooklyn, NY, and the Ph.D. degree in operations research from the Massachusetts Institute of Technology, Cambridge, in 1965, 1966, and 1969, respectively.

From 1966 to 1967 he was an Associate Engineer with Honeywell EDP and from 1969 to 1970 he worked as a Staff Member at the IBM Cambridge Scientific Research Center. He has also been a consultant to the Computer Sciences Department, Thomas J. Watson Research Center. He is presently Professor of Computer Sciences and Electrical Engineering, University of Texas, Austin, and a Consultant for Information Research Associates, Inc., Austin, TX. His current research interests include modeling of computer systems, networks, and reliability.

Dr. Chandy is a member of the Association for Computing Machinery.