A methodology for developing distributed programs

Citation for published version (APA):

Ramesh, S., & Mehndiratta, S. L. (1987). A methodology for developing distributed programs. *IEEE Transactions on Software Engineering*, 13(8), 967-976. <https://doi.org/10.1109/TSE.1987.233514>

DOI:

[10.1109/TSE.1987.233514](https://doi.org/10.1109/TSE.1987.233514)

Document status and date:

Published: 01/01/1987

Document Version:

Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

- A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
- The final author version and the galley proof are versions of the publication after peer review.
- The final published version features the final layout of the paper including the volume, issue and page numbers.

[Link to publication](#)

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy

If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

A Methodology for Developing Distributed Programs

S. RAMESH AND S. L. MEHNDIRATTA

Abstract—A methodology, different from the existing ones, for constructing distributed programs is presented. It is based on the well-known idea of developing distributed programs via synchronous and centralized programs. The distinguishing features of the methodology are: 1) specification include process structure information and distributed programs are developed taking this information into account, 2) a new class of programs, called PPSA's, is used in the development process, and 3) a transformational approach is suggested to solve the problems inherent in the method of developing distributed programs through synchronous and centralized programs. The methodology is illustrated with an example.

Index Terms—Communicating processes, decentralization, distributed programming, programming methodology, program transformation.

I. INTRODUCTION

THE difficulty of programming for distributed systems, due to their characteristic property of process autonomy, demands a systematic methodology for constructing distributed programs. In recent times, many approaches have been suggested for such a systematic development [1], [5], [12], [13], [17]. A divide and conquer method has been proposed in [5] that consists of developing a collection of layers from a specification of the distributed program to be developed and composing these layers, into the required distributed program, preserving their communication closeness. In [1], [12], [13], distributed programs are derived by transforming sequential programs/centralized action systems; this derivation is accomplished either by first partitioning the variables of the sequential programs and then transforming them [12], [13] or by first decentralizing the centralized action systems, through introduction of control variables and control actions, and then transforming them [1]. The basic idea behind these approaches is to develop distributed programs for synchronous and centralized programs. This idea has been found to be extremely useful as it renders the process of developing distributed programs a two-step activity. The first step involves deriving synchronous and centralized programs for appropriate specifications and the second step entails translating these derived programs into the required distributed programs.

Certain comments are in order on some of the existing methodologies.

In [5], the specifications of distributed programs are informal and no method has been suggested for systematic

cally constructing layers. Derivation of layers *directly* from specification does not seem to be easy as layers are distributed programs involving disjoint process codes and communications and hence their construction requires design of *separate* process codes that when put together, satisfy their specifications.

Like [3], we believe that the distributed solutions of problems are required *not* for getting efficient solutions but for programming distributed systems whose process structures and topologies are decided by physical considerations such as distribution of data, resources, etc. The methodologies discussed in [12], [13] are not suitable for developing distributed programs with prespecified topologies because 1) they decide the process structure and the topology of a distributed program, being developed, based on performance considerations and 2) constructing sequential programs and partitioning their variables to derive distributed programs with prespecified topologies are not straightforward.

Hence, it would be desirable to have a methodology that will *systematically* develop distributed programs from their specifications that *include* process structure and topology informations. It is the purpose of this paper to demonstrate that based on the same idea [1], [5], [12], [13] of developing distributed programs via synchronous and centralized programs, such a methodology can be devised.

Accordingly, this paper presents a methodology for developing distributed programs from their specifications. The specification of any program consists of 1) its process structure specification that specifies the number of processes, their variables and their interconnections (i.e., the topology of the underlying network) and 2) a pair of pre- and postconditions (on the variables of its processes) in the form of first order logical formulae. Given such a specification, the method develops a high level distributed program called PPSA—Program with Processes involving Shared Actions, which can be implemented in terms of processes that communicate via messages.

A PPSA is a program consisting of a fixed number of processes. Each process has a set of variables disjoint from those of other processes. These processes *interact* with each other via *shared actions*. Shared actions are a multiprocess generalization of CSP (communicating sequential processes [8]) communication primitives as they allow *arbitrary number* of processes to communicate with each other. Furthermore, shared actions are similar to joint actions [1] in that they involve local variables from different processes. Shared actions are, however, more gen-

Manuscript received January 31, 1985; revised July 31, 1985.

The authors are with the Department of Computer Science and Engineering, Indian Institute of Technology, Bombay 400076, India.

IEEE Log Number 8715546.

eral than the joint actions: joint actions are *always* guarded by enabling conditions that may involve variables from *all* the processes in the system whereas shared actions can be *simple shared actions* or *Boolean shared actions* and they involve variables from *only those* processes that share the actions. Also, PPSA's are process-oriented as they specify explicitly the control structure of processes and they model noncyclic programs. On the other hand, joint action systems are modeled after the Manna-Pnueli model [10] and hence they are action-oriented, do not specify explicitly the control structure of processes and model deadlocking/nonterminating programs. It may perhaps be noted that the idea of sharing common actions among processes is also made use of in [6].

The method of developing PPSA's is based on the same divide-and-conquer principle as employed in [5]. But it is more systematic as it makes use of 1) the WP-calculus [4], [7] approach and 2) a transformational approach for developing efficient PPSA's. We shall see in Section II that PPSA's involve only sequential segments and hence their construction and manipulation are easier and more systematic. The method of implementing PPSA's in terms of communicating processes is similar to the method given in [12] for translating sequential programs into CSP programs.

Since PPSA's are based on a model different from that of joint action systems, the proposed methodology obviously proceeds along different lines from those discussed in [1]. Furthermore, our methodology also differs significantly from the one discussed in [17].

The organization of this paper is as follows. Section II elaborates further on PPSA's. Section III discusses the proposed methodology while Section IV illustrates the methodology. Section V contains some concluding remarks.

In what follows, we shall make use of GCL (Guarded Command Language [4]), WP-calculus [4], [7], and CSP [8].

II. PPSA'S—PROGRAMS WITH PROCESSES INVOLVING SHARED ACTIONS

A PPSA is a high level distributed program consisting of a fixed number of processes. Each process has its own set of variables (disjoint from those of other processes) and a process code. Each process code may involve four kinds of actions: *Simple shared actions*, *Boolean shared actions*, *alternate actions*, and *iterative actions*. A simple shared action involves a command in GCL while a Boolean shared action involves a Boolean expression. A shared action (simple or Boolean) may occur in the process codes of more than one process and is said to be *shared by* all those processes in whose process codes it occurs; trivially the set of processes sharing an action could be a singleton. A shared action may involve the variables of a process that shares it. Also whenever a variable of a process occurs in a shared action, the action should be shared by that process.

An alternate action (iterative action) is of the form **if** $ga_1 \square \dots \square ga_k$ **fi** (**do** $ga_1 \square \dots \square ga_k$ **od**) where for each $i \in k = \{1, \dots, k\}$, ga_i is a *guarded action* of the form $g \rightarrow S$; g is a guard and S is a sequence of actions involving simple shared/alternate/iterative actions. The general form of g is $b; b'$ where b is a Boolean shared action shared by *only one process* and b' is a Boolean shared action shared by *more than one process*. The actions b and b' are, respectively, called the *local* and *non-local components* of g . The guard g can have only a local or nonlocal component and must have at least one such component.

Syntactically, a PPSA, say PP , consists of a declaration followed by a list of shared actions and a list of processes. The declaration defines a set of variables of PP . Each shared action in the shared action list has a name, a list of names of processes that share it, and a *body*. The body is a GCL command or a Boolean expression. Similarly each process in the process list has a name, a list of its variables, and process code. The set of variables listed in a process is disjoint from the set of variables listed in any other process in the list and union of all these sets is the set of variables declared at the head of PP . The process code of a process is an action list consisting of names of simple shared/alternate/iterative actions. The names of processes and shared actions should be disjoint.

The following is an example of PP :

```
PP :: /*list of variables of PP*/
var  $x_1, x'_1, x_2, x'_2, x_3, x'_3, x_4, x'_4$ :integer
/* list of shared actions of PP*/
init1: shared by  $P_1$ :
     $x_1 := k_1$ 
end;
:
init4: shared by  $P_4$ :
     $x_4 := k_4$ 
end;
exchange1: shared by  $P_1, P_2$ :
     $x'_1 := x_2$ ;
    if  $x_1 > x'_1 \rightarrow x_1, x'_1 := x'_1, x_1$ 
    □  $x_1 \not> x'_1 \rightarrow$  skip
    fi;  $x_2 := x'_1$ 
end;
:
exchange3: shared by  $P_3, P_4$ :
     $x_{3'} := x_4$ ;
    if  $x_3 > x'_{3'} \rightarrow x_3, x'_{3'} := x'_{3'}, x_3$ 
    □  $x_3 \not> x'_{3'} \rightarrow$  skip
    fi;  $x_4 := x'_{3'}$ 
end;
unsorted: shared by  $P_1, P_2, P_3, P_4$ :
     $(x_1 > x_2) \vee (x_2 > x_3) \vee (x_3 > x_4)$ 
end;
/* list of processes of PP */
[ $P_1$ ::var  $x_1, x'_1$ ; /* variables of  $P_1$  */
```

```

init1;
/* process code of P1 */
do unsorted → exchange1 od
|| P2::var x2,x'2; /* variables of P2 */
init2;
/* process code of P2 */
do unsorted → exchange1; exchange2 od
|| P3::var x3,x'3;
init3;
do unsorted → exchange3; exchange2 od
|| P4::var x4,x'4;
init4;
do unsorted → exchange3 od
]

```

It may be noted that $init_1, \dots, init_4, exchange_1, exchange_2$, and $exchange_3$ are simple shared actions while “unsorted” is a Boolean shared action of PP ; $k_1, \dots, k_4$ are fixed integers. The shared actions $init_2$ and $init_3$ are not shown above, as they can be obtained from $init_1$ by replacing the subscript 1 of k, x and P , by 2 and 3, respectively; similarly $exchange_2$ can be obtained from $exchange_1$.

Model of Execution

The execution of a PPSA involves the concurrent execution of its processes. Each process is executed by an agent (processor). The execution of a process consists of executing the actions in its process code (in the order specified). The execution of an action *succeeds* or *fails* (for the ease of presentation, we ignore trivial modes of failures like arithmetic overflow, division by zero, etc.). The execution of a shared action in a process fails if one of the processes sharing that action has terminated. Otherwise, the execution of the action is delayed until all processes sharing that action reach the stage of executing the action in their respective process codes. The action is then *collectively* executed by all these processes.

In the case of simple shared action, such a *collective* execution may result in a change of values of the variables involved in the action. This change is the same as that would be produced when the *body* of the shared action is executed in accordance with GCL semantics. On the other hand, such a *collective* execution of a Boolean shared action makes the processes know the value of the *body* which is nothing but a Boolean expression. It may be noted that the effect of collective execution of a shared action shared by only one process can be produced by simply executing/evaluating the body of the shared action.

The collective execution of a simple shared action succeeds iff the execution of its *body*, in accordance with GCL semantics, succeeds whereas that of a Boolean shared action succeeds iff the *body* evaluates to true.

The execution of an alternate action consists of selecting (the selection is nondeterministic) a guarded action, with a successfully executable guard, and executing it; if no guarded action has a successfully executable guard, the execution of the alternate action fails. The execution

of a guarded action consists of executing its guard first and then the action list following the guard. A guard is executed by executing its constituent components from left to right. The guard succeeds iff the execution of its constituent component(s) succeed(s). The execution of an iterative action similarly consists of repeatedly performing the following action until all of its guards fail: select nondeterministically a guarded action, with a successfully executable guard, and execute it.

The execution of a process terminates successfully when all the actions executed during the execution have been successfully executed while that of a PPSA terminates successfully when all of its processes have been successfully executed. The execution of a PPSA fails whenever the execution of at least one of its processes fails or a deadlock occurs. A deadlock occurs when there is a set of processes each of which is ready to execute a shared action but is waiting for one or more processes (in the set) to reach the stage of executing the shared action.

As an illustration of this model of executing, consider the execution of PP (the PPSA given earlier). Concurrent execution of P_1, P_2, P_3 , and P_4 constitutes the execution of PP . The execution of each process $P_i, i = 1, \dots, 4$, involves first executing $init_i$ and then repeatedly executing the only guarded action in the iterative action, that follows $init_i$, until “unsorted” fails. Each $init_i$ is shared by only one process and hence does not require any synchronization; it initializes x_i to k_i . The action “unsorted” is shared by all the processes and hence involves synchronization of all the four processes. Each $exchange_i, i = 1, 2, 3$, exchanges the value of x_i and x_{i+1} , if $x_i > x_{i+1}$. Consequently repeated execution of the guarded actions of the iterative actions, eventually makes “unsorted” fail thereby sorting $x_1, \dots, x_4$.

Remark 2.1: It is evident from the execution model of PPSA’s that shared actions are (like CSP primitives [8]) the means of process interactions. However, they differ from CSP primitives in one important aspect: shared actions are a multiprocess generalization of two-process CSP primitives, as they can allow an arbitrary number of processes to synchronize and communicate with each other.

Remark 2.2: The model of execution of PPSA’s is abstract as it specifies *only* the effect of collective execution of shared actions without explaining how this execution is carried out. The precise way in which a shared action is collectively executed is a matter of implementation and for the purpose of easy understanding, analysis, and design, such an abstract model is sufficient and convenient.

Remark 2.3: The most important property of PPSA’s is that they can specify arbitrary degree of synchronization, i.e., they can allow any number of processes to synchronize and communicate with each other, through shared actions. This gives rise to tight synchronization and poor performance whenever the number of processes sharing an action is very large. Hence, whenever possible, PPSA’s involving shared actions shared by small number of processes should be developed. The methodology presented in Section III has been proposed precisely

to develop such programs. This methodology exploits the above mentioned property of PPSA's to achieve the required goal. As we shall see in Section III, highly synchronous and centralized programs can be developed with ease directly from their specifications while highly asynchronous and decentralized programs can be constructed from these programs by successfully transforming them, in a systematic fashion. Multiprocess synchronization has been employed precisely to have such a systematic methodology.

Based on the above model of execution, correctness measures similar to those given for CSP [8] can be defined.

Definition: A PPSA PP is (totally) correct with respect to a pair of predicates $\langle Q, R \rangle$, iff *each* execution that starts from an initial state satisfying Q successfully terminates in a final state satisfying R .

Because of the presence of nondeterminism, PPSA's can exhibit time-dependent behaviors: the execution of a PPSA with one particular set of relative speeds of execution of its processes terminates successfully, while execution with a different set of relative speeds may not terminate. The above definition of correctness, as it requires all executions of a correct PPSA to terminate successfully, rules out time-dependent behaviors in correct PPSA's.

For the ease of presentation, we shall denote in the sequel any PPSA by an ordered pair $\langle [P_1 \parallel \dots \parallel P_n], \mathcal{A} \rangle$ where $P_1, \dots, P_n$ are its processes and $\mathcal{A}$ is the set of its shared actions. Furthermore, shared actions shared by only one process will appear explicitly as GCL commands/Boolean expression in process codes. Shared actions will be denoted by a, a' (with possible subscripts), GCL commands by c, c' (with possible subscripts), and expressions by e, e' (with possible subscripts).

Implementation of PPSA's

Let us recall that the execution model of PPSA's is abstract. In order to use PPSA's for describing distributed systems, it is necessary to furnish a concrete distributed model of computation that can implement PPSA's. A brief discussion of such an implementation is given in this section; full details, however, can be found in [15].

The implementation of a PPSA, say PP , requires a network of nodes, each of which houses one or more processors. Each node has a unique index, which is a natural number, identifying the node. Furthermore, each node is connected via communication channels to a few other nodes called its neighbors. The processors of different nodes do not share any memory space but can communicate by passing messages via the channels. The processors at each node know the indexes of the node and its neighboring nodes. The communication channels are bi-directional and they neither corrupt, lose, nor change the order in which messages are to be communicated.

Furthermore, the implementation involves associating a process in PP with each node in the network. One of the processors in the node has a memory location for every

variable of the associated process. This processor executes a piece of code corresponding to each constituent action in the process code of the associated process. These codes involve only local computations and message passing activities.

The piece of code corresponding to a simple shared action, say a , whose body is $x := e$, is as follows, where P is the process (actually, processor) executing the code:

```

P :: if P owns x1
    then receive values of nonlocal variables (if any)
           occurring in  $e$ , from the processes that own
           these variables;
           evaluate  $e$  using these values;
           assign the result of evaluation to  $x$ 
    else send values of local variables (if any) occur-
           ring in  $e$  to the process that owns  $x$ 

```

When the body of a is an iterative (alternate) command c , the piece of code executed by P is:

```

P :: repeat
    if P is the process having the maximum index
       among all processes sharing  $a$ 
    then begin
        receive from each other process that shares
         $a$ , value(s) of nonlocal variable(s) occurring
        in the guards of  $c$ ;
        evaluate the guards of  $c$  using these values;
        if there is atleast one successful guard
        then begin
            select one of the guarded commands
            with successful guards;
            Inform all the other processes (sharing
             $a$ ) the identity of the command se-
            lected;
            execute the command following the
            guard in the command selected.2
        end
    else /* no guarded command succeeds */
        direct all the processes to terminate
        (abort) the execution of  $c$  before itself
        terminating (aborting) the execution of  $c$ .
    else /* P is not the maximum index-process */
        begin
            send to the maximum-index-process values
            of local variables (if any) occurring in the
            guards of  $c$ ;
            receive message from the maximum-index-
            process;
            if the message directs to terminate (abort)
            then terminate (abort) the execution of  $c$ 
            else select and execute the command iden-
            tified in the message
        until execution is terminated (execution is aborted
        or once one of the guarded commands has been ex-
        ecuted)

```

¹More precisely, if the process in PP , associated with P , owns x .

²More precisely, execute the code corresponding to the command.

The piece of code executed by P is empty when the body of a is **skip** while it is the code obtained by concatenating the codes corresponding to c_1 and c_2 when the body of a is $c_1; c_2$.

P executes the following code for an iterative (alternate) action, say, a :

```

P :: repeat
  evaluate the local components of  $a$ ;
  if local components of all the guards evaluate
  to false
  then terminate (abort) the execution of  $a$ 
  else begin
    attempt to select a guarded action whose local
    and nonlocal components evaluate to true;
    if the attempt fails
    then terminate (abort) the execution of  $a$ 
    else execute the action list following the
    guarded action selected
  end
until execution is terminated (execution is aborted
or once one of the guarded actions has been executed)

```

Certain noteworthy points about the above implementation are: 1) The codes, corresponding to a simple shared action a , executed by the processes sharing a , are similar to the codes of the processes of the CSP program obtained by translating the body of a , using the method proposed in [12]. 2) The messages sent (during the execution) should have proper identities so that the processes receiving them can identify them. 3) The selection of a guarded action is a nontrivial step in the execution of the code corresponding to an iterative/alternate action. This requires agreement among all the other processes that share the nonlocal component of the action selected in the sense that these processes also should select the actions with the same nonlocal component. This selection problem is essentially the multiprocess generalization of the guard selection problem in the implementation of CSP with output guards [2]. The guard selection protocol employed in [16] for the implementation of CSP, can be generalized and used for the selection of guarded actions. 4) In the implementation, during the execution, any pair of processes sharing an action may require to communicate with each other. It is shown in [15] that it is sufficient if the nodes corresponding to processes sharing any action form a connected subnetwork in the whole network; a set of nodes in a network forms a connected subnetwork if any node in this set is connected to any other node in the set, either directly or indirectly through a chain of nodes all of which are in the set.

In conclusion, the entire implementation of PPSA's can be realized in a distributed environment that involves message passing as the only means of interaction among processors.

III. METHODOLOGY FOR DEVELOPING PPSA's

In this section, we shall see how to construct PPSA's from specifications. The specification of a PPSA is assumed to consist of two parts: 1) process structure specification specifying the number of processes, their variables, and the topology of the underlying network and 2) pre- and postconditions in the form of first-order logical assertions that specify the initial and final states of the PPSA.

A transformational approach is adopted for the construction of PPSA's. This involves the following steps:

- 1) Furnish a correct initial design of a PPSA from the given specification.
- 2) Successively improve the design by applying certain transformation rules until a PPSA with desirable properties is obtained.

A Method for Deriving the Initial Design

This method is an extension of the top-down approach for developing sequential programs. Given a specification of the PPSA to be developed, it proceeds to develop the PPSA using the three steps discussed below.

Let $\langle Q, R \rangle$ be the pre- and postcondition of the PPSA PP to be developed. Further, let PP be specified to consist of n -processes, $P_1, \dots, P_n$, each of which has a set of variables and a set of neighbors, i.e., processes with which it can communicate (during its execution).

Step 1: Divide the given pair $\langle Q, R \rangle$ into a number of pre-postconditions, $\langle Q_1, Q_2 \rangle, \dots, \langle Q_{m-1}, Q_m \rangle$ such that $Q \Rightarrow Q_1$ and $Q_m \Rightarrow R$ for some $m > 1$ and $Q_1, \dots, Q_m$.

Remark 3.1: The predicates $Q_1, \dots, Q_m$ may involve new variables, not specified in the specification, which are appropriately associated with the processes without affecting the property of disjointness of the sets of variables of different processes.

Step 2: For each pair $\langle Q_j, Q_{j+1} \rangle, j \in (m-1) = \{1, \dots, m-1\}$ find a collection of predicates $Q_{j,k}$ and $Q'_{j,k}$ such that

$$Q_j \Rightarrow \bigwedge_{k \in h} Q_{j,k}$$

$$\bigwedge_{k \in h} Q'_{j,k} \Rightarrow Q_{j+1}$$

for some $h \in n$; h is dependent upon j as it may be different for different j 's. Then construct a set of GCL commands $c_{j,k}, k \in h$ such that for $k' \in h$ and $k' \neq k$, the following five conditions are satisfied:

- 1) $Q_{j,k} \Rightarrow WP(c_{j,k}, Q'_{j,k})$.
- 2) $Q_{j,k} \wedge Q_{j,k'} \Rightarrow WP(c_{j,k}, Q_{j,k'})$.
- 3) $Q_{j,k} \wedge Q'_{j,k'} \Rightarrow WP(c_{j,k}, Q'_{j,k'})$.
- 4) $\text{Ind}(c_{j,k}) \cap \text{Ind}(c_{j,k'}) = \emptyset$.
- 5) The process in $\{P_r \mid r \in \text{Ind}(c_{j,k})\}$ form a connected subnetwork in the underlying network specified.

$\text{Ind}(c_{j,k})$ is the set of indexes of processes whose variables are referred in $c_{j,k}$ and WP is the weakest precondition function [4].

Remark 3.2: Each $c_{j,k}$ may also involve new variables which are to be partitioned and associated with those processes whose variables are referred in $c_{j,k}$. It can be easily seen that condition 1) implies that executing $c_{j,k}$ in a state satisfying $Q_{j,k}$ terminates in a state satisfying $Q'_{j,k}$, while conditions 2) and 3) imply that such an execution of $c_{j,k}$ does not effect $Q_{j,k'}$ and $Q'_{j,k'}$ (if they were true before the execution) for $k' \neq k$.

Step 3: Construct from these GCL commands, the following PPSA,

$$PP = \langle [P_1 \parallel \dots \parallel P_n], \mathcal{Q} \rangle$$

where

$$\mathcal{Q} = \{a_{j,k}/k \in h, j \in (m-1)\}$$

$a_{j,k}$: **shared by** P_i , $i \in \text{Ind}(c_{j,k})$:

$c_{j,k}$
end

$P_i :: \text{var } x_{i,1}, \dots, x_{i,s};$
 $pc(i)$

$x_{i,1}, \dots, x_{i,s}$ are variables associated with P_i for some $s \geq 1$, s being dependent upon i , $pc(i)$ is the process code of P_i , defined recursively as follows:

$$pc(i) = pc'(i, 1)$$

$$pc'(i, j) = \text{if } j = m \text{ then nil (empty list)} \\ \text{else if there exists } k \in h \text{ such that} \\ i \in \text{Ind}(c_{j,k}) \\ \text{then }^3 a_{j,k}; pc'(i, j+1) \\ \text{else } pc'(i, j+1)$$

Theorem: PP is correct with respect to $\langle Q, R \rangle$.

Proof: See Appendix.

Remark 3.3: $c_{j,k}$, $k \in h$, $j \in (m-1)$ can be derived using WP -calculus [4], [7] for appropriate pre-post condition pairs $\langle Q_{j,k}, Q'_{j,k} \rangle$. The topology specification influences both the decomposition of specification and the choice of $c_{j,k}$'s. This method of developing PPSA's is similar to constructing distributed programs using the layer construct suggested in [5].

Improving the Initial Design: Unless extra care is taken the initial design developed using the above method will be highly sequential, synchronous and centralized. As remarked in Section II, these features are undesirable and we recommend an incremental approach to develop a highly asynchronous and decentralized program from the initial design. According to this approach, the initial design is successively transformed, using well-defined transformation rules, to derive programs with increased concurrency, asynchrony, and decentralization.

The following classes of transformation rules, that alter PPSA's, have been found to be useful (in what follows $PP \Rightarrow PP'$ denotes a rule that transforms the PPSA, PP into PP').

Decomposition: A decomposition rule has one of the following forms:

$$1) \langle [P_1 \parallel \dots \parallel P_n], \mathcal{Q}U\{a\} \rangle \\ \Rightarrow \langle [P'_1 \parallel \dots \parallel P'_n], \mathcal{Q}U\{a_1, a_2\} \rangle$$

where⁴ a : c_1 ; c_2 , a_1 : c_1 and a_2 : c_2 for some GCL commands c_1 and c_2 and P'_i is the process obtained from P_i by replacing the occurrence of a by a_1 ; a_2 for each P_i sharing a , while other P'_i 's are the same as the corresponding P_i 's.

$$2) \langle [P_1 \parallel \dots \parallel P_n], \mathcal{Q}U\{a\} \rangle \\ \Rightarrow \langle [P'_1 \parallel \dots \parallel P'_n], \mathcal{Q}U\{a_1, \dots, a_{2k}\} \rangle$$

where a : **if** $e_1 \rightarrow c_1 \square \dots \square e_k \rightarrow c_k$ **fi** or a : **do** $e_1 \rightarrow c_1 \square \dots \square e_k \rightarrow c_k$ **od** for some Boolean expressions $e_1, \dots, e_k$ and GCL commands $c_1, \dots, c_k$, a_1 : e_1 , $\dots$, a_k : e_k , a_{k+1} : c_1 , $\dots$, a_{2k} : c_k and for each P_i sharing a , P'_i is obtained from P_i by replacing a by **if** $a_1 \rightarrow a_{k+1} \square \dots \square a_k \rightarrow a_{2k}$ **fi** (or **do** $a_1 \rightarrow a_{k+1} \square \dots \square a_k \rightarrow a_{2k}$ **od**, respectively) while other P'_i 's are the same as the corresponding P_i 's.

Rearrangement: Any rule in this class transforms a PPSA by rearranging the occurrences of a few of its shared actions in some of its processes.

Removal: A typical rule in this class is of the form

$$\langle [P_1 \parallel \dots \parallel P_i \parallel \dots \parallel P_n], \mathcal{Q} \rangle \\ \Rightarrow \langle [P_1 \parallel \dots \parallel P'_i \parallel \dots \parallel P_n], \mathcal{Q} \rangle$$

where P'_i is the process obtained from P_i by removing an occurrence of a shared action.

Replacement: A replacement rule has one of the following forms:

$$1) \langle [P_1 \parallel \dots \parallel P_n], \mathcal{Q}U\{a\} \rangle \\ \Rightarrow \langle [P'_1 \parallel \dots \parallel P'_n], \mathcal{Q}U\{a'\} \rangle$$

where a and a' are simple shared action and P'_i is obtained from P_i by replacing the occurrence of a , if any by a' , for each $i \in n$.

$$2) \langle [P_1 \parallel \dots \parallel P_n], \mathcal{Q}U\{a, a_1, \dots, a_k\} \rangle \\ \Rightarrow \langle [P'_1 \parallel \dots \parallel P'_n], \mathcal{Q}U\{a', a'_1, \dots, a'_k\} \rangle$$

where a is the nonlocal component occurring in those guards, in processes $P_1, \dots, P_n$ whose local components are $a_1, \dots, a_k$ and P'_i , $i \in n$ is the same as P_i , except for the difference that a' , a'_j , $j \in k$ take the place of a , a_j , respectively.

PPSA's need to satisfy some syntactic and semantic conditions in order that their correctness is preserved under these transformations.

We shall now briefly illustrate how these transformation rules can be used to improve PPSA's.

Consider the PPSA, PP given by

³Because of condition 4), there will be at most one k satisfying $i \in \text{Ind}(c_{j,k})$.

⁴In the sequel sometimes, shared actions will be denoted by $a:c$ or $a:e$ where a is the name and c/e is the body of the shared action.

$$PP = \langle [P_1 \parallel P_2 \parallel P_3 \parallel P_4], \{a_1, a_2, a_3\} \rangle$$

$$P_1 :: a_1, P_2 :: a_1; a_2, P_3 :: a_2; a_3, P_4 :: a_3$$

This program is highly sequential; a_1 , a_2 , and a_3 are executed in strict sequential order. If a_2 and a_3 are “commutative” then a PPSA, PP' that is more concurrent can be obtained where

$$PP' = \langle [P'_1 \parallel P'_2 \parallel P'_3 \parallel P'_4], \{a_1, a_2, a_3\} \rangle$$

$$P'_1 :: a_1, P'_2 :: a_1; a_2, P'_3 :: a_3; a_2, P'_4 :: a_3$$

by rearranging (reversing) the shared actions a_2 and a_3 in P_3 .

Similarly, a more asynchronous PPSA

$$PP' = \langle [P'_1 \parallel P'_2 \parallel P'_3], \{a_1, a_2, a_3, a_4, a'_2, a'_3\} \rangle$$

$$P'_1 :: a_1; \text{do } a'_2 \rightarrow a_2 \text{ od}$$

$$P'_2 :: \text{done}_1, \text{done}_2 := \text{false}, \text{false};$$

$$\quad \text{do } a'_2 \rightarrow a_2; \text{done}_1 := \text{true}$$

$$\quad \square a'_3 \rightarrow a_3; \text{done}_2 := \text{true}$$

$$\text{od}$$

$$P'_3 :: a_4; \text{do } a'_3 \rightarrow a_3 \text{ od}$$

$$a'_2 \rightarrow \neg \text{done}_1$$

$$a'_3 \rightarrow \neg \text{done}_2$$

can be obtained from

$$PP = \langle [P_1 \parallel P_2 \parallel P_3], \{a_1, a_2, a_3, a_4\} \rangle$$

$$P_1 :: a_1; a_2$$

$$P_2 :: a_2; a_3$$

$$P_3 :: a_4; a_3$$

by

1) making the following *replacements*

$$a_2 \Rightarrow \text{done}_1 := \text{false}; \text{do } a'_2 \rightarrow a_2; \text{done}_1 := \text{true} \text{ od}$$

$$a_3 \Rightarrow \text{done}_2 := \text{false}; \text{do } a'_3 \rightarrow a_3; \text{done}_2 := \text{true} \text{ od}$$

($\text{done}_1, \text{done}_2$ are two new variables associated with P_2),

2) *decomposing* the two new shared actions into

$$\text{done}_1 := \text{false}, a'_2, a_2, \text{done}_1 := \text{true} \text{ and } \text{done}_2$$

$$:= \text{false}, a'_3, a_3, \text{done}_2 := \text{true}, \text{ respectively,}$$

3) *removing* the shared actions $\text{done}_1 := \text{false}, \text{done}_1 := \text{true}$ and $\text{done}_2 := \text{false}, \text{done}_2 := \text{true}$ from P_1 and P_3 , respectively, and

4) *rearranging* the shared actions in P_2 .

For this transformation to be valid a_2 and a_3 should be “commutative.”

It must be noted that in the sense of [5], PP possesses *insistent* communication while PP' has *indulgent* communication.

A most important transformation is the *decentralization* transformation that replaces a shared action by a new shared action which is shared by a fewer number of processes than the former; a desirable instance is the replacement of a shared action shared by more than one process, by a shared action shared by a single process. This transformation involves applying the rules *replacement* and *removal* in succession. The possibility of applying *decentralization* depends upon the redundancy present in the system. Whenever possible this transformation should be

attempted in the development of PPSA's as it reduces the communication complexity of the corresponding distributed programs.

IV. ILLUSTRATION OF THE METHODOLOGY

We shall now illustrate our methodology by developing a distributed program using the methodology. The specification of the program to be developed is as follows.

Process Structure Specification: The topology of the underlying network is a *rooted, complete* binary tree T of n -processes, $P_1, \dots, P_n$. Each process P_i has a variable x_i and can communicate with $P_{s1(i)}$, $P_{s2(i)}$ and $P_{f(i)}$ where $s1(i)$, $s2(i)$, and $f(i)$ are the indexes of the left, right and father processes of P_i ($i \in n$); when i is the index of a leaf process (root process) then $s1(i) = s2(i) = 0$ ($f(i) = 0$).

Precondition:

$$Q \equiv \bigwedge_{i=1}^n (x_i = k_i).$$

Postcondition:

$$R \equiv x_1 = \text{sum}(1)$$

where $\text{sum}(i)$ is the sum of the keys recorded in the nodes of the subtree rooted at i ($\text{sum}(0) = 0$).

First an initial design is constructed for this specification. Following the method given in Section III, a decomposition of $\langle Q, R \rangle$ is attempted. One possible decomposition is $\langle Q_1, Q_2 \rangle, \dots, \langle Q_{m-1}, Q_m \rangle$, where

$$Q_j \equiv \bigwedge_{r=1}^j \bigwedge_{i \in I_r} (\text{FIN}(i) \wedge \text{INV}(i))$$

$$\bigwedge_{r=j+1}^m \bigwedge_{i \in I_r} (\text{INIT}(i) \wedge \text{INV}(i)).$$

m —depth of T .

I_r —set of indexes of processes at level r (levels are measured from the leaves up the root.)

$$\text{INV}(i) \equiv \text{sum}(i) (\text{sum}(s1(i)) + \text{sum}(s2(i)) + k_i).$$

$$\text{INIT}(i) \equiv x_i = \text{sum}(i).$$

$$\bigwedge_{r=m+1}^m \dots \equiv \text{true}.$$

It is obvious that $Q \Rightarrow Q_1, Q_m \Rightarrow R$. It may be noted that Q_j corresponds to that state of the network in which all the processes at levels up to j have updated their x_i 's to the required value while the other processes have not.

Sequential Segments for $\langle Q_j, Q_{j+1} \rangle$: As per step 2 of the methodology, for each $j = 1, \dots, m-1$, sequential segments satisfying conditions 1)–5) of Section III are developed.

Consider the following two predicates:

$$Q_{j,i} \equiv \bigwedge_{r=1}^{j-1} \bigwedge_{t \in I_r} (\text{FIN}(t) \wedge \text{INV}(t))$$

$$\begin{aligned}
& \bigwedge_{r=j+2}^m \bigwedge_{t \in I_r} (\text{INIT}(t) \wedge \text{INV}(t)) \\
& \wedge \text{FIN}(s1(i)) \wedge \text{FIN}(s2(i)) \\
& \wedge \text{INIT}(i) \wedge \text{INV}(i) \\
Q'_{j,i} & \equiv \bigwedge_{r=1}^j \bigwedge_{t \in I_r} (\text{FIN}(t) \wedge \text{INV}(t)) \\
& \wedge \bigwedge_{r=j+2}^m \bigwedge_{t \in I_r} (\text{INIT}(t) \wedge \text{INV}(t)) \\
& \wedge \text{FIN}(i) \wedge \text{INV}(i).
\end{aligned}$$

For each $i \in I_{j+1}$ we have

$$\begin{aligned}
Q_j & \Rightarrow \bigwedge_{i \in I_{j+1}} Q_{j,i} \\
Q_{j+1} & \equiv \bigwedge_{i \in I_{j+1}} Q'_{j,i}.
\end{aligned}$$

Also, for $c_i = x_i := x_{s1(i)} + x_{s2(i)} + x_i$

$$\begin{aligned}
Q_{j,i} & \Rightarrow \text{WP}(c_i, Q'_{j,i}) \\
(Q_{j,i} \wedge Q_{j,i'}) & \Rightarrow \text{WP}(c_i, Q_{j,i'}) \\
(Q_{j,i} \wedge Q'_{j,i'}) & \Rightarrow \text{WP}(c_i, Q'_{j,i'}) \\
\text{Ind}(c_i) \cap \text{Ind}(c_{i'}) & = \emptyset
\end{aligned}$$

for every $i, i' \in I_{j+1}$ and $i \neq i'$, where $\text{Ind}(c_i) = \{i, s1(i), s2(i)\}$.

Furthermore, the processes whose indexes are in $\text{Ind}(c_i)$ form a connected subnetwork. Thus all the five conditions of Section III are satisfied for the c_i 's.

Initial Design: Accordingly the initial design is given by

$$PP = \langle [P_1 \parallel \dots \parallel P_n], \alpha \rangle$$

where

$\alpha = \{a_i \mid i \in I_{j+1}, j = 1, \dots, m-1\}$
 a_i : shared by $P_i, P_{s1(i)}, P_{s2(i)}$:
 c_i
end
 $P_i :: \text{var } x_i : \text{integer} \text{ /* initial value is } k_i \text{ */}$
 $a_i ; a_{f(i)}$
for $i \in I_j$ and $1 < j < m$,
 $P_i :: \text{var } x_i : \text{integer} \text{ /* initial value is } k_i \text{ */}$
 $a_i ; a_{f(i)}$
for $i \in I_1$ and
 $P_i :: \text{var } x_i : \text{integer} \text{ /* initial value is } k_1 \text{ */}$
 a_i
for $i \in I_m$.

PP is highly synchronous as the execution of a_i requires simultaneous participation of $P_i, P_{s1(i)}$ and $P_{s2(i)}$. We shall improve PP by applying certain transformation rules.

Transformation 1: For each $i \in I_{j+1}$, $1 \leq j < m$, replace a_i by a'_i given by

$$a'_i : \text{set}(i); \text{do cond}(i, s1(i)) \rightarrow \text{upd}(i, s1(i))$$

$$\square \text{ cond}(i, s2(i)) \rightarrow \text{upd}(i, s2(i))$$

od

where

$$\begin{aligned}
\text{set}(i) & = \text{done}_{i,1}, \text{done}_{i,2} := \text{false}, \text{false}; \\
\text{done}_{s1(i),3}, \text{done}_{s2(i),3} & := \text{false}, \text{false} \\
\text{cond}(i, k) & \equiv \neg \text{done}_{i,1} \wedge \neg \text{done}_{k,3} \\
\text{upd}(i, k) & = x_i := x_i + x_k; \text{done}_{i,1} \\
& := \text{true}; \text{done}_{k,3} := \text{true}
\end{aligned}$$

for $k = sr(i)$ and $r = 1, 2$; $\text{done}_{i,r}$ for $r = 1, 2$ are new variables of type Boolean that are associated with P_i , $i \in I_{j+1}$ and $1 \leq j < m$ while $\text{done}_{i,3}$ is a Boolean variable associated with P_i for $i \in I_j$ and $1 \leq j < m$.

Transformation 2: Decompose a'_i into $\text{set}(i)$, $\text{cond}(i, s1(i))$, $\text{cond}(i, s2(i))$, $\text{upd}(i, s1(i))$ and $\text{upd}(i, s2(i))$ for $i \in I_{j+1}$, $1 \leq j < m$.

Transformations 1 and 2, as such, do not improve the program. But they enable subsequent application of the rules that improve the program.

Transformation 3: Remove $\text{cond}(i, s1(i))$, $\text{upd}(i, s1(i))$, and $\text{cond}(i, s2(i))$, $\text{upd}(i, s2(i))$, respectively, from processes $P_{s2(i)}$ and $P_{s1(i)}$ for $i \in I_j$, $1 < j \leq m$.

The amount of synchronization is reduced by this transformation, as it reduces the number of processes sharing some actions.

Final Transformation: Replace $\text{cond}(i, k)$ in P_i by $\neg \text{done}_{i,1}$; $\text{synch}(i, k)$ and in P_k by $\neg \text{done}_{k,3}$; $\text{synch}(i, k)$ where $\text{synch}(i, k)$: **true** is a Boolean shared action shared by P_i and P_k for $i \in I_j$, $k = sr(i)$, $r = 1, 2$. The processes of the resulting PPSA are given by

$P_i :: \text{var } x_i : \text{integer};$
 $\text{set}(i); \text{do } \neg \text{done}_{i,1}; \text{synch}(i, s1(i))$
 $\rightarrow \text{upd}(i, s1(i))$
 $\square \neg \text{done}_{i,2}; \text{synch}(i, s2(i))$
 $\rightarrow \text{upd}(i, s2(i))$
od;
 $\text{set}(f(i)); \text{do } \neg \text{done}_{i,3}; \text{synch}(f(i), i)$
 $\rightarrow \text{upd}(f(i), i) \text{ od}$

for $i \in I_j$ and $1 < j < m$.

$P_i :: \text{var } x_i : \text{integer};$
 $\text{set}(f(i)); \text{do } \neg \text{done}_{i,3}; \text{synch}(f(i), i)$
 $\rightarrow \text{upd}(f(i), i) \text{ od}$

for $i \in I_1$,

$P_i :: \text{var } x_i : \text{integer};$
 $\text{set}(i); \text{do } \neg \text{done}_{i,1}; \text{synch}(i, s1(i))$
 $\rightarrow \text{upd}(i, s1(i))$
 $\square \neg \text{done}_{i,2}; \text{synch}(i, s2(i))$
 $\rightarrow \text{upd}(i, s2(i))$
do;

for $i \in I_m$.

No more transformation is required as no further improvement is possible. As indicated in Section III, this can be directly implemented in terms of communicating processes. For the sake of those familiar with CSP, we give a CSP version of this program.

```

 $P_i :: \text{done}_{i,1}, \text{done}_{i,2} : \text{false}, \text{false}$ 
 $\text{do } \neg \text{done}_{i,1}; P_{s1(i)}?x'_i \rightarrow x_i := x_i + x'_i;$ 
 $\text{done}_{i,1} := \text{true}$ 
 $\square \neg \text{done}_{i,2}; P_{s2(i)}?x'_i \rightarrow x_i := x_i + x'_i;$ 
 $\text{done}_{i,2} := \text{true}$ 
 $\text{od}; \text{done}_{i,3} := \text{false};$ 
 $\text{do } \neg \text{done}_{i,3}; P_{f(i)}!x_i \rightarrow \text{done}_{i,3} := \text{true} \text{ od}$ 

```

for $i \in I_j$ and $1 < j < m$. P_i for $i \in I_1$ and $i \in I_m$ are similar.

Remark 4.1: A program similar to the final PPSA has been developed in [5] using their methodology. The similarities and dissimilarities of the development of this program and that of the final PPSA may be noted.

Remark 4.2: The correctness of the final PPSA derived follows from that of the initial design and the validity of each of the transformations. The initial design is correct by construction because of Theorem 1. Intuitively, it is obvious that each of the transformations leads to correct program. This fact can, however, be formally verified but this requires formal treatment of correctness, transformations, and their validity. These are discussed in [15].

Remark 4.3: The derivation of the final program seems to be more complex and lengthier compared to the complexity of the final program arrived at. One may say that the final program can be arrived at intuitively in a straightforward fashion. But the difference between such an intuitively developed program and the one derived above is that the former requires an *a posteriori* correctness proof that establishes its correctness whereas the latter does not; as noted above the latter is automatically correct by virtue of being the end product of a sequence of well-defined and correct steps. Accordingly, the complexity of the above derivation is to be compared to that of the *a posteriori* proof of correctness. The complexity of the *a posteriori* proof of correctness is well-known and has forced us to adopt the principle of hand-in-hand development of programs and their proofs [7] in the case of sequential programs. The proof of correctness of distributed programs is much more complex than that of sequential program. Hence such a principle is much more valid and advantageous in the case of distributed programs.

V. CONCLUSION

A methodology, based on the well-known idea of developing distributed programs via synchronous and centralized programs, has been reported. Its distinguishing features are as follows. 1) Specifications include topology information and distributed programs are developed taking this information into account. 2) A new class of programs (PPSA's) is used in the development process. 3) A transformational approach is suggested to solve the problems inherent in the method of developing distributed programs through synchronous and centralized programs. Obvious advantages of this transformation approach are 1) highly concurrent, decentralized, and asynchronous programs can be developed with relative ease and 2) the correctness of the programs so developed follows from

the correctness of the initial PPSA's designed and from the correct application of the transformation rules. PPSA's have been found to be suitable for developing distributed programs with prespecified topologies. In contrast, sequential programs are not useful as they do not exhibit process structures. PPSA's are also superior to layered programs as their construction and manipulation is easier and more systematic. They are different from joint action systems as the former are based on a model different from that of the latter. Thus the methodology proposed, although it makes use of some of the known ideas from [1], [5], [12], [13], is different and more powerful than these approaches.

The proposed methodology has been applied to many standard problems, like the one discussed in this paper, and found to be quite useful. Currently work is in progress to demonstrate more effectively the practical utility of the methodology proposed. As the first step towards this, a more formal treatment of PPSA's and the methodology has been attempted [14], [15]. With the transformation rules discussed in this paper, the number of steps required for the development of "big" programs may be large. Considering the complexity of intuitive development followed by a *a posteriori* proof of correctness, the large number of systematic and correct steps may not matter. However, use of higher level transformation rules that are combinations of the rules described in this paper, will considerably reduce the number of steps and the amount of reasoning. Hence attempts are being made to discover such high level transformation rules.

APPENDIX

PROOF OF THEOREM 1

Consider an execution of PP with the following additional synchronization: execution of any shared action $a_{j,k}$, $k \in h$ is allowed if and only if all the other shared actions $a_{j,k'}$, $k' \in h$ are about to be executed for $j \in (m-1)$. Then we have,

1) executing PP , with an initial state satisfying Q and with the above mentioned additional synchronization, results in a final state satisfying R .

2) any ordinary execution of PP with an initial state is equivalent to the execution PP with the same initial state and with the additional synchronization.

1) follows from

$$Q \Rightarrow Q_1,$$

$$Q_j \Rightarrow WP(c_{j,1} \parallel \cdots \parallel c_{j,h}, Q_{j+1})$$

for $j \in (m-1)$, where $c_{j,1} \parallel \cdots \parallel c_{j,h}$ is a concurrent segment whose execution consists of executing the constituent GCL-commands in any arbitrary order and $Q_m \Rightarrow R$.

These three conditions are implied by the conditions 1)–4) of Section III. Any additional synchronization discipline can, at most, introduce deadlocking. Hence 2) follows from 1). 1) and 2) together prove the theorem.

ACKNOWLEDGMENT

We thank the referees for their constructive comments on earlier versions of this paper. We also wish to thank N. B. Joshi and N. S. Chaudhari for their help in the preparation of this paper.

REFERENCES

- [1] R. J. R. Back and R. Kurki-Suonio, "Decentralization of process nets with centralized control," in *Proc. 2nd Conf. Principles of Distributed Computing*, Montreal, P.Q., Canada, Aug. 1983.
- [2] G. N. Buckley and Silberschatz, "An effective implementation for the generalized input-output construct of CSP," *ACM Trans. Program. Lang. Syst.*, vol. 5, no. 2, 1983.
- [3] K. M. Chandy, "Paradigms for distributed computing," in *Rec. 3rd Conf. Foundations of Software Technology and Theoretical Computer Science*, Bangalore, India, Dec. 1983.
- [4] E. W. Dijkstra, *A Discipline of Programming*. Englewood Cliffs, NJ: Prentice-Hall, 1976.
- [5] T. Elrad and N. Francez, "Decomposition of distributed programs into communication closed layers," *Sci. Comput. Program.*, vol. 2, 1982.
- [6] N. Francez and B. Hailpern, "SCRIPTS—A communication abstraction mechanism," in *Proc. 2nd Conf. Principles of Distributed Computing*, Montreal, P.Q., Canada, Aug. 1983.
- [7] D. Gries, *The Science of Computer Programming*. New York: Springer-Verlag, 1981.
- [8] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, 1978.
- [9] G. Levin and D. Gries, "Proof techniques for communicating sequential processes," *Acta Inform.*, vol. 15, 1981.
- [10] Z. Manna and A. Pnueli, "How to cook a temporal proof system for your pet language," in *Proc. 10th Annu. Conf. Principles of Programming Languages*, Austin, TX, Jan. 1983.
- [11] R. Mccurley and F. B. Schneider, "Derivation of a distributed algorithm of finding paths in directed networks," Cornell Univ., Ithaca, NY, Tech. Rep. 83-86, Dec. 1983.
- [12] A. Moitra, "Synthesis of communicating processes," in *Proc. 2nd Conf. Principles of Distributed Computing*, Montreal, P.Q., Canada, Aug. 1983.
- [13] V. R. Prasad, "Derivation of distributed programs from sequential programs," TIFR, Bombay, India, Tech. Rep. TR 87, Aug. 1983.
- [14] S. Ramesh and S. L. Mehndiratta, "A new class of high level programs for distributed computing systems," in *Proc. Fifth Conf. Foundations of Software Technology and Theoretical Computer Science (Lecture Notes in Computer Science, Vol. 206)*. New York: Springer-Verlag, 1985.
- [15] S. Ramesh, "Programming with shared actions: A methodology for developing distributed programs," Ph.D. dissertation, Indian Inst. Technol., Bombay, India, to be submitted.
- [16] J. S. Schwartz, "Distributed synchronization of communicating sequential processes," *Artificial Intell.*, Univ. Edinburgh, Scotland, Tech. Rep., July 1978.
- [17] L. Shriram, N. Francez, and M. Rodeh, "Distributed K-selection: From a sequential to a distributed algorithm," in *Proc. 2nd Conf. Principles of Distributed Computing*, Montreal, P.Q., Canada, Aug. 1983.

S. Ramesh was born in Kumbakonam, India, on November 29, 1958. He received the B.Sc. degree in applied sciences from Thiagarajar College of Engineering in 1978 and the B.E. degree in electronics and communication engineering from the Indian Institute of Science, Bangalore, 1981.

Shortly, he will be submitting his Ph.D. dissertation in the Department of Computer Science and Engineering at Indian Institute of Technology, Bombay. He is currently working on distributed programming. His other research interests are

specification, verification, and formal semantics of concurrent and distributed programs and logic programming.

S. L. Mehndiratta has worked at Tata Institute of Fundamental Research, Bombay, India, Dalhousie University, Canada, and the Indian Institute of Technology, Madras. Currently he is a Professor of Computer Science with the Department of Computer Science and Engineering, Indian Institute of Technology, Bombay. He has published papers in the areas of computer science, operations research, and environmental engineering. His current areas of interest are distributed computing, methodologies for developing software, fault-tolerant protocols for distributed systems, message systems, and optimization of industrial systems.