A Review of Specification and Verification Methods for Parallel Programs, including the Dataflow Approach

AKSHAY K. DESHPANDE, MEMBER, IEEE, AND KRISHNA M. KAVI, SENIOR MEMBER, IEEE

Parallel programs are usually described informally, and these descriptions are implemented on parallel computer systems. When a program does not run correctly, it is often very difficult to determine whether the program description or the implementation is incorrect. This has led to the search for more formal descriptions of parallel programs, and proof systems for the verification of the implementations. In this paper, we will introduce some such formal methods for the specification and verification of parallel programs. We will also describe a new method that is based on dataflow graphs.

I. Introduction

The growing demand for increased computational speeds while approaching the limits on uniprocessor performance has led to the research into parallel and distributed processing. The research efforts have been directed towards designing architectures, techniques for representing parallelism, new languages, and tools. In addition, considerable amount of work has been devoted to the development of theoretical models and methods of analysis under which inherent properties of parallelism can be precisely defined and studied.

Programs, especially parallel programs, are often described informally. There are advantages to informal descriptions: a problem can be studied without the baggage of a formal notation and proof system. However, such informal descriptions of complex problems can be overwhelming. The errors and inconsistencies contained are usually difficult to discover—"programs that seemed so obviously correct at one time are, in retrospect, so obviously wrong" [1]. This is the reason for formal approaches to the specification of parallel programs, and proof systems to verify the implementation of the specifications.

A complete theory of programming (both sequential and parallel) includes 1) a method for specification of programs permitting a clear statement of all requirements, 2) a method of reasoning about specifications that brings out implementation alternatives, 3) a method of developing programs along with a proof system to verify the correctness of the programs with respect to the specifications, and 4) a method of mapping the programs on to architectures to achieve high efficiency [1]. In this paper we are concerned only with the specification and verification as described below.

A specification is a statement about what the program should do, while a description of how it is achieved is an implementation. A program may be designed hierarchically such that a specification at one level becomes an implementation at other levels. The functional correctness verifies that the implementation realizes the specification.

It may be desirable to distinguish between two views on correctness. When dealing with the proof of a program (we will call it the Computational model), either the program is annotated with predicates—the proof consists of demonstrating that a predicate holds whenever program control is at the corresponding point in the program text; or refining the program by adding more detail, eventually leading to an execution model on a target architecture.

A process can be defined as the realization of a program on a computer system. While dealing with correctness in a process model, only the behavior of a program, insofar as it can be described in terms of a limited set of events, is of interest. Typically, the behavior of a process is described by its interaction with other processes. Apart from functional correctness, verification of processes requires proving safety and liveness properties. Safety properties essentially state that nothing bad will ever happen; in other words the process never enters an unacceptable state. An example of safety property is, if the process receives an input, it will reach a state where an output is produced. Other examples of safety properties are that two processes are never in their critical section simultaneously; that a message is received only if (and after) it has been sent. Liveness properties state that something good will happen in the future; that is, the process will eventually enter a desirable state. An example of liveness is, if a process receives an input, output will be produced in the future.

Manuscript received September 17, 1988; revised June 5, 1989. This work was supported in part by the State of Texas Coordinating Board on Higher Education under the Advanced Research Program, Grant #1770.

The authors are with the Department of Computer Science Engineering, University of Texas at Arlington, Arlington, TX 76019-0015, USA.

IEEE Log Number 8932752.

0018-9219/89/1200-1816$01.00 © 1989 IEEE

Another example of liveness is that the process will progress unless it has reached termination. In this paper we will present methods that are applicable to process models.

In formal models for studying the behavior of processes, it is common to abstract the description of a process. In one such abstraction, a process is described by a set of possible states, a set of inputs that cause a transition in the state of the process, and a set of outputs produced by the process on state transitions. A process is specified using assertions regarding the conditions before (pre) and after (post) a state transition. In a different abstraction, a process is described by the elements through which it affects the environment. A trace is a sequence of recordings of the interaction between a process and its environment. In such an abstraction, it is common to represent a process by its input and output components, and its behavior by the trace. A system may comprise of many processes with interactions among some of the processes. Some models ([1], [2], and [3]) record events of a system based on total ordering of time, assuming that only one event can occur at any given time. Some other models ([4], [5], and [6]) impose partial order on the events recorded (thus defining causality), allowing the occurrence of concurrent events.

Most process models are compositional; that is, a specification of a process is formed from specifications of the component processes. In such systems, the rules of composition of processes are controlled by the algebra on the processes. Sometimes, the concept of trace is extended to a sequence of observations, where an observation is defined as the input and output streams on all ports of all processes (in a network of processes).

The proof of correctness in a process model then becomes equivalent to the demonstration of safeness and liveness of the composite (or network of) processes as specified. Assertions on traces are used to verify that the specified processes do not enter undesirable states, and enter desirable states in the future.

A. An Example

Networks of processes, representing operating systems, often require nondeterministic operations. By nondeterministic, we mean that the execution of the operation is time-independent. It is important that the techniques used to characterize the networks be able to handle nondeterminate operators. In order to illustrate the differences between the various models presented here, we use the example of Brock and Ackerman [4]. This example is of interest because it shows a subtle difference between two very similar compositions. It shows that history relations are insufficient for characterizing a network of processes containing nondeterminate operators.

A process interacts with the environment through well defined interfaces called the *ports* or *links*. The sequence of values that each port receives or transmits during a computation is called a history. Determinate operators have only one possible output history tuple for each input history tuple. The function which maps input history tuple into its output history tuple is called the history function. Nondeterminate operators, which have a set of possible output history tuples for an input history tuple, can be represented by history relations.

Consider the network of processes shown in Fig. 1. The network consists of determinate processes D_1, D_2, and P_i ($i = 1, 2$); and a nondeterminate process *Merge*. The network produces at most two values. Both processes D_1 and D_2 read one value on their input ports and write two copies of it on their output ports. The process *Merge* nondeterministically reads a value on one of its input ports and writes the value on its output port. Two networks are formed by using two different but similar processes P_i's. P_1 is a process which produces a value after reading the first input, while P_2 produces its first value after reading two inputs. The history functions for these processes are

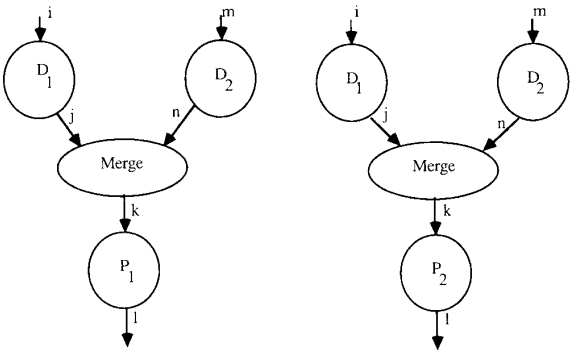

Fig. 1. Brock and Ackerman Example.

$$D_1 \text{ or } D_2 \qquad P_1(\epsilon) = \epsilon \qquad P_2(\epsilon) = \epsilon$$

$$D(\epsilon) = \epsilon \qquad P_1(k) = k \qquad P_2(k) = \epsilon$$

$$D(k.X) = k.k \qquad P_1(k.\lfloor.X) = k.\lfloor \qquad P_2(k.\lfloor.X) = \lfloor$$

where ϵ is empty history, k and $\lfloor$ are values, X represents input history, "." represents concatenation, and $D(X)$ represents the output history.

The history relation for the nondeterministic process *Merge* is

$$Merge(X, \epsilon) = \{X\}; \qquad Merge(\epsilon, Y) = \{Y\}$$

$$Merge(k.X, \lfloor.Y) = \{k.Z \mid Z \in Merge(X, \lfloor.Y)\} \cup \{\lfloor.Z \mid Z \in Merge(k.X, Y)\}$$

Two different networks NET_1 and NET_2 are formed: NET_1 with D_1, D_2, *Merge* and P_1; and NET_2 with D_1, D_2, *Merge* and P_2. The history relations for the two networks NET_1 and NET_2 are the same despite the different behavior of P_1 and P_2. If either network receives an input, then P_i's receive at least two inputs, and the network produces two outputs. The history relation for NET_i's with inputs k and $\lfloor$ will be

$$NET_i(\epsilon, \epsilon) = \{\epsilon\}$$

$$NET_i(k.X, \epsilon) = \{k.k\} \quad NET_i(\epsilon, \lfloor.X) = \{\lfloor.\lfloor\}$$

$$NET_i(k.X, \lfloor.Y) = \{k.k, k.\lfloor, \lfloor.k, \lfloor.\lfloor\}$$

The subtle difference in the behaviors of NET_1 and NET_2 can be detected when placed in a larger network. Let NET_1 and NET_2 be part of two networks $COMP_1$ and $COMP_2$ as shown in Fig. 2. The input to the network is through D_1, while the output of the networks is forked back to D_2 through another process *Add*. *Add* is a process which increments the input received by one, and writes it on the output.

Let $COMP_1$ receive a single input with a value of 2. The value 2 is passed through NET_1 and becomes the first output

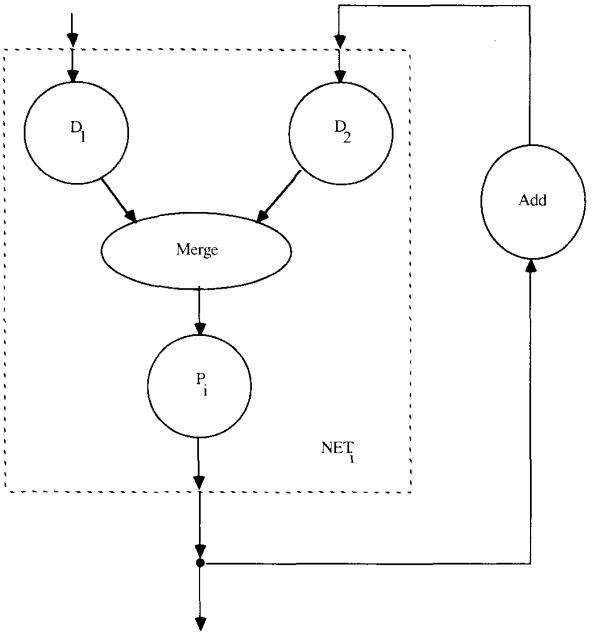

Fig. 2. Network $COMP_i$.

of $COMP_1$. The process *Add* receives the value 2, adds 1 to it and writes a value of 3 on its output. The *Merge* process can now choose either the second output from D_1, with a value of 2, or the first output from D_2, with a value of 3. This is possible because, we cannot assume the execution speed of the individual processes, or the scheduling of the processes is unknown. It is possible that the process *Merge* does not proceed until it has inputs on both its input ports, thus, the second output of $COMP_1$ can be either 2 or 3.

If $COMP_2$ receives an input with a value of 2, the value 2 reaches process P_2 through D_1 and *Merge*. However, P_2 cannot produce a value until it receives two inputs. So, when the second output from D_1 reaches P_2, $COMP_2$ produces the only possible sequence of output $\{2 \cdot 2\}$.

II. Trace-Based Models

In this section, we present three trace-based models. The example described in Section I-A is used to show the difference between the techniques. Trace-based models are attractive because of their simplicity, and their information-hiding property.

A. Communicating Sequential Processes (CSP)

Most early programming languages were designed for sequential programs. Programming constructs required for concurrent processing were not well understood and provisions for handling concurrency were "add-on" features in the languges. Hoare [7] presented a few simple constructs to aid languages in dealing with concurrent processing. Parallel command, and input and output commands were introduced along with Dijkstra's guarded command [8] as means of structuring concurrent programs. A proof technique based on the concepts was presented in [7] and an elaborate theory of processes was presented in [2]. In this paper we will review only the features needed to understand the principles, and to describe the example detailed in Section I-A.

A trace provides an abstraction for a process, hiding irrelevant internal details. A trace is a recording of the sequence of events during some execution of a process. A process is specified by describing the properties of its traces. A crt-terminal (process) which displays a character after a key is pressed, and then halts is specified as

$$TERMINAL = Key \rightarrow Display \rightarrow STOP$$

where *Key* indicates the event of pressing a key on the keyboard, *Display* is the event of displaying the character pressed, *STOP* indicates that the process halts, and *TERMINAL* is the process name. The ordering of events is indicated by "→."

An elaborate set of constructs is defined in order to facilitate specification of the process behavior. A process can be represented as a composition of subprocesses using the parallel command. Non-terminating processes and processes involved in repetitive action are specified by means of recursion. For example, a terminal which when turned on, displays every character pressed on the keyboard until power is turned off can be specified as

$$P_1 = On \rightarrow P_2$$

$$P_2 = Key \rightarrow Display \rightarrow P_2 \mid Off$$

where P_1 and P_2 are process names, *Key* and *Display* are as described above, *On* indicates the event of turning the power on, *Off* indicates the event of turning the power off, and "|" is choice operator (explained below).

A parallel operation (‖) is used to compose processes that execute concurrently. Input (?) and output (!) commands are introduced as the communication primitives through which processes interact. Communication occurs between two parallel processes (processes composed using the par-

allel command) whenever an input command in one process specifies the other process as the source, an output command in the other process specifies the first process as the destination, and the target variable of the input command matches the destination expression in the output command. Input and output commands are said to correspond when the above conditions are satisfied, and they are executed simultaneously. In a variation, input and output commands can name the communication channel instead of the process name. However, both the input and output commands must name the process (or channel) at the other end of the communication.

Choice (|) is an operator which allows an execution to select between two alternatives. *Nondeterminism* is introduced by providing an operation ([]) which arbitrarily selects between its choices. If the first choice cannot be made, then the second choice will be used; or if the second choice cannot be made, the first choice is used. However, if both can be selected, then the choice between them is nondeterministic. While constructing a process from component processes the internal structure is visible. After the construction, the internal details are no longer needed and can be hidden by the *conceal* (\) operation.

1) Example: The process D_1 (see Section I-A) which reads one input (with a value of v_1) on its channel i and outputs the same value twice on its channel j, is specified as

$$D_1 = i?v_1 \rightarrow j!v_1 \rightarrow j!v_1$$

The process D_2, same as D_1 except for the channel names, is specified as

$$D_2 = m?u_1 \rightarrow n!u_1 \rightarrow n!u_1$$

The process *Merge* which nondeterministically reads a value on its input channels j and n, and writes the value (v or u, depending on the choice made) on its output channel k, and iterates, is specified as

$$Merge = (((j?v \rightarrow k!v) \,[]\, (n?u \rightarrow k!u)) \rightarrow Merge)$$

The process P_1 which reads one value on its input channel k, writes the value on its output channel l, and repeats this sequence once more before stopping, is specified as

$$P_1 = (k?w_1 \rightarrow l!w_1 \rightarrow k?w_2 \rightarrow l!w_2 \rightarrow STOP)$$

where *STOP* indicates the process termination.

The process P_2 which reads two values on its input channel k before writing them in the same order to its output channel l and then stops is specified as

$$P_2 = (k?w_1 \rightarrow k?w_2 \rightarrow l!w_1 \rightarrow l!w_2 \rightarrow STOP)$$

The composition of processes D_1, D_2, *Merge* and P_1 using the parallel command and concealing the internal channels (j, k and n) yields the following specification

$$\begin{aligned} NET_1 &= (D_1 \| D_2 \| Merge \| P_1) \setminus \{j, k, n\} \\ &= ((i?v \,[]\, m?u) \rightarrow l!w_1 \rightarrow l!w_2 \rightarrow STOP) \end{aligned}$$

The composition of processes D_1, D_2, *Merge* and P_2 using the parallel command and concealing the internal channels (j, k, and n) yields the following specification

$$\begin{aligned} NET_2 &= (D_1 \| D_2 \| Merge \| P_2) \setminus \{j, k, n\} \\ &= ((i?v \,[]\, m?u) \rightarrow l!w_1 \rightarrow l!w_2 \rightarrow STOP) \end{aligned}$$

In this model the events in a trace are totally ordered. This may not be true in real systems, particularly in distributed systems where each node has its own clock (time). Also, as can be seen from the composition of processes (NET_1 and NET_2) in the example described, it is not possible to differentiate between two similar, but different processes. CSP uses two different basic models in forming concurrent processes. The execution of each process in a composition is sequential, while the interactions among processes can be parallel, nondeterministic, or selective.

B. Scenario Model

History functions which map each input history tuple into an output history tuple are adequate to characterize networks of determinate processes. History relations, extension of history functions, which map each input history tuple into a set of possible output history tuple, have been used to characterize networks of nondeterminate processes. However, Brock and Ackerman [4] have shown, using the dataflow model of computation, that history relations are inadequate to characterize networks of nondeterminate processes. They presented a characterization in which networks are presented by scenario sets. A scenario includes causality information along with input and output history tuples. The causality information relates the elements of input and output history tuples with the elements responsible for their creation. An algebra for dataflow graphs and scenario sets is presented in [9].

Programs in the dataflow model of computation are represented by a graph. The nodes of a dataflow graph are called operators. Each operator of a dataflow graph is identified through a label. A firing (execution) of an operator removes tokens (values) from the input ports and produces tokens on the output ports. Different operators can be defined with different firing rules based on the tokens present. Links of the dataflow graph connect input and output ports of operators (processes). During the execution, tokens flow through these links from one operator to another. The unconnected ports within a graph become the ports of the graph itself. Larger graphs can be built from smaller ones by considering the smaller graphs as operators within the larger graphs.

1) Dataflow Graph Algebra: A dataflow graph algebra, consisting of three operators, enables combining of the dataflow graph operators. The three operators of the dataflow graph algebra are *graph union*, *port relabeling*, and *port connection*. The graph union ($\|_g$) operation associates two disjoint graphs, retaining the port labels. The result of the union operation is to form a new graph whose input and output ports are the union of the input and output ports of the constituent graphs, respectively. It is required that the port labels of the graphs participating in the union be disjoint. The restriction of disjoint port labels in the union operation is eased by providing a port relabeling ($./_g.$) operator. The port relabeling operation $[a/_g b]$ renames the port a to b. The operation cannot introduce duplicate port names, hence it is defined only when b is not a port of the graph G. The result of this operation is to remove the port label a and add the port label b to the set of ports of the graph G.

The port connection ($.\rightarrow_g.$) operator is used to connect an input port to an output port within a graph. The connected ports become internal to the graph and cannot participate in any future graph interconnections. The result of this operation is to remove (hide) the input and output port,

participating in the operation, from the input and output ports of the graph, respectively.

To illustrate the use of these operators, Fig. 3 shows a dataflow graph which computes the dot product of 2 two-

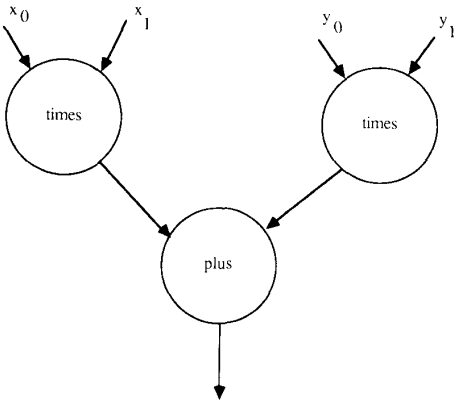

Fig. 3. Dot product computation using Brock's model.

element vectors. The inputs to the graph are the vectors: (x_0, y_0) and (x_1, y_1). In this example, each operator is assumed to have input ports labelled as In_1 and In_2 and the output to be labelled as Out_1. The port through which the graph produces output is to be labelled as *Result*. The description using the algebraic notation is as follows

$$(times[In_1/_g x_0][In_2/_g x_1][Out_1/_g Op_1 Out_1]\|_g$$

$$times[In_1/_g y_0][In_2/_g y_1][Out_1/_g Op_2 Out_1]\|_g$$

$$plus[In_1/_g Op_3 In_1][In_2/_g Op_3 In_2][Out_1/_g Result])$$

$$[Op_1 Out_1 \rightarrow_g Op_3 In_1][Op_2 Out_1 \rightarrow_g Op_3 In_2]$$

2) Scenario Set Algebra: Scenario sets, similar to graphs, have disjoint input and output ports. A scenario is a triple $\langle E, V, C\rangle$, where E is the set of events, V is the valuation function, and C is the causality relation. Each element of the scenario set E is an ordered pair consisting of port label and a nonzero positive integer. If an event $\langle a, n\rangle$ is in E, then it represents the event of producing (or receiving) the nth value at the output (or input) port a. The valuation function V maps the events of E into the set of token values. The value of the nth token produced (or received) on port a is given by $V(\langle a, n\rangle)$. The causality relation provides the dependencies in token production. If $\langle a, n\rangle\, C\, \langle b, m\rangle$ is a causality relation, then the event of producing (or receiving) the nth element at port a must precede the event of producing (or receiving) the mth element at port b.

The scenario set algebra, like the dataflow graph algebra, has three operators: *Union* $(\|_s)$, *Relabeling* $(./_s.)$, and *Connection* $(.\rightarrow_s.)$. Applications of the scenario set operators must satisfy the same constraints on the input and output port labels as the operators of dataflow graph algebra.

The scenario set union is used to produce the composition of scenario sets. The operation is defined only when the two scenario sets have disjoint port labels. The resulting scenario set is also a triple with each of its components being the union of the corresponding components from the constituent scenario sets. For example, let S_1: $\langle E_1, V_1, C_1\rangle$ and S_2: $\langle E_2, V_2, C_2\rangle$ be two scenario sets, then $S_1\|_s S_2 = \langle E_1 \cup E_2, V_1 \cup V_2, C_1 \cup C_2\rangle$. Taking the union of functions and relations works because the event domains are disjoint.

The scenario set port relabeling is similar to the dataflow graph relabeling operator. The relabeling operator renames the ports, that is, it replaces all the occurrences of its first operand with the second operand. For example, let S: $\langle E, V, C\rangle$ be a scenario where

$$E = \{\langle \gamma, m\rangle\};$$

$$V = \{V(\langle \gamma, m\rangle) | \langle \gamma, m\rangle \in E\}; \text{ and}$$

$$C = \{\langle \gamma, m\rangle\, C\, \langle \delta, n\rangle | \langle \gamma, m\rangle, \langle \delta, n\rangle \in E\}$$

Application of the scenario set port relabeling operation $S[\alpha/_s\beta\rangle$ results in a scenario such that

$$E' = \{\langle \gamma, m\rangle | \langle \gamma, m\rangle \in E \text{ and } \gamma \neq \alpha\} \cup \{\langle \beta, m\rangle | \langle \alpha, m\rangle \in E\}$$

$$V'(\gamma, m) = V(\langle \gamma, m\rangle), \quad \text{if } \langle \gamma, m\rangle \in E \text{ and } \gamma \neq \alpha$$

$$V'(\beta, m) = V(\langle \alpha, m\rangle), \quad \text{if } \langle \alpha, m\rangle \in E$$

$$\langle \gamma, m\rangle\, C'\, \langle \delta, n\rangle = \langle \gamma, m\rangle\, C\, \langle \delta, n\rangle \quad \text{if } \gamma \neq \alpha \text{ and } \delta \neq \alpha$$

$$\langle \gamma, m\rangle\, C'\, \langle \beta, n\rangle = \langle \gamma, m\rangle\, C\, \langle \alpha, n\rangle \quad \text{if } \gamma \neq \alpha$$

$$\langle \beta, m\rangle\, C'\, \langle \delta, n\rangle = \langle \alpha, m\rangle\, C\, \langle \delta, n\rangle \quad \text{if } \delta \neq \alpha$$

$$\langle \beta, m\rangle\, C'\, \langle \beta, n\rangle = \langle \alpha, m\rangle\, C\, \langle \alpha, n\rangle$$

The scenario set port connection operation is more complex than the connection operation of dataflow graph algebra. The connect operation may form cycles, resulting in inconsistent causal order. In order to avoid this inconsistency, the connect operation is done in steps. First, all scenario pairs whose data values do not match on ports that are linked to each other are discarded. Each event pair is merged into one scenario. For the matching events on the linked input–output pairs, the valuation function yields the same value, leading to value-consistent scenarios. The second step is to remove any cycle, that would lead to event pairs on linked ports where the event on input port does not precede the event on the output port. This is done by discarding all such cycle-causing event pairs, leading to causality-consistent scenarios. Finally, each scenario for the linked port is removed, resulting in the characterization of the network based on external ports only, while hiding the internal ports. For more details and examples, see [9].

3) Example: We will redo the example of section I-A using the scenario set algebra.

The process D_1 has the following scenario set:

$$E = (\langle In_1, 1\rangle, \langle Out_1, 1\rangle, \langle Out_1, 2\rangle)$$

$$V(\langle In_1, 1\rangle) = V(\langle Out_1, 1\rangle) = V(\langle Out_1, 2\rangle)$$

$$\langle In_1, 1\rangle\, C\, \langle Out_1, 1\rangle;\ \langle In_1, 1\rangle\, C\, \langle Out_1, 2\rangle;$$

$$\langle Out_1, 1\rangle\, C\, \langle Out_1, 2\rangle.$$

The process D_2 has the following scenario set:

$$E = (\langle In_1, 1\rangle, \langle Out_1, 1\rangle, \langle Out_1, 2\rangle)$$

$$V(\langle In_1, 1\rangle) = V(\langle Out_1, 1\rangle) = V(\langle Out_1, 2\rangle)$$

$$\langle In_1, 1\rangle\, C\, \langle Out_1, 1\rangle;\ \langle In_1, 1\rangle\, C\, \langle Out_1, 2\rangle;$$

$$\langle Out_1, 1\rangle\, C\, \langle Out_1, 2\rangle.$$

The process *Merge* has the following input–output history function:

$$Merge(X, \epsilon) = \{X\}; \quad Merge(\epsilon, Y) = \{Y\};$$

$$Merge(a.X, b.Y) = \{a.Z | Z \in Merge(X, b.Y)\} \cup \{j.Z | Z \in Merge(a.X, Y)\}.$$

A sample scenario for an input history tuple of $(p \cdot q, r)$ is

$E = (\langle In_1, 1\rangle, \langle In_1, 2\rangle, \langle In_2, 1\rangle, \langle Out_1, 1\rangle, \langle Out_1, 2\rangle, \langle Out_1, 3\rangle)$

$V(\langle In_1, 1\rangle) = p; V(\langle In_1, 2\rangle) = q; V(\langle In_2, 1\rangle) = r;$

$V(\langle Out_1, 1\rangle) = p; V(\langle Out_1, 2\rangle) = r; V(\langle Out_1, 3\rangle) = q;$

The causality relation consists of the following tuples:

$\langle In_1, 1\rangle C \langle In_1, 2\rangle; \langle In_1, 1\rangle C \langle Out_1, 1\rangle; \langle In_1, 1\rangle C \langle Out_1, 3\rangle; \langle In_2, 1\rangle C \langle Out_2, 2\rangle;$

$\langle Out_1, 1\rangle C \langle Out_1, 2\rangle; \langle Out_1, 1\rangle C \langle Out_1, 3\rangle; \langle Out_1, 2\rangle C \langle Out_1, 3\rangle;$

This is one sample scenario. By varying the sequence of event occurrences, we get other scenarios, and the scenario set is the set of all scenarios.

The process P_1 has the following scenario set:

$E = (\langle In_1, 1\rangle, \langle In_1, 2\rangle, \langle Out_1, 1\rangle, \langle Out_1, 2\rangle)$

$V(\langle In_1, 1\rangle) = V(\langle Out_1, 1\rangle); V(\langle In_1, 2\rangle) = V(\langle Out_1, 2\rangle)$

$\langle In_1, 1\rangle C \langle In_1, 2\rangle; \langle In_1, 1\rangle C \langle Out_1, 1\rangle; \langle In_1, 2\rangle C \langle Out_1, 2\rangle; \langle Out_1, 1\rangle C \langle Out_1, 2\rangle.$

The process P_2 has the following scenario set:

$E = (\langle In_1, 1\rangle, \langle In_1, 2\rangle, \langle Out_1, 1\rangle, \langle Out_1, 2\rangle)$

$V(\langle In_1, 1\rangle) = V(\langle Out_1, 1\rangle); V(\langle In_1, 2\rangle) = V(\langle Out_1, 2\rangle)$

$\langle In_1, 1\rangle C \langle In_1, 2\rangle; \langle In_1, 1\rangle C \langle Out_1, 1\rangle; \langle In_1, 2\rangle C \langle Out_1, 1\rangle; \langle Out_1, 1\rangle C \langle Out_1, 2\rangle.$

The composition (NET_1) of the processes D_1, D_2, *Merge* and P_1 using the scenario set operators yields the following scenario (we use the specific scenarios which bring out the difference between the two compositions):

$E_1 = (\langle In_1, 1\rangle, \langle In_2, 1\rangle, \langle Out_1, 1\rangle, \langle Out_1, 2\rangle)$

$V_1(\langle In_1, 1\rangle) = V_1(\langle Out_1, 1\rangle); V_1(\langle In_2, 2\rangle) = V_1(\langle Out_1, 2\rangle)$

$\langle In_1, 1\rangle C_1 \langle Out_1, 1\rangle; \langle In_2, 1\rangle C1 \langle Out_1, 2\rangle; \langle Out_1, 1\rangle C_1 \langle Out_1, 2\rangle$

$E_2 = (\langle In_1, 1\rangle, \langle In_2, 1\rangle, \langle Out_1, 1\rangle, \langle Out_1, 2\rangle)$

$V_2(\langle In_1, 1\rangle) = V_2(\langle Out_1, 2\rangle); V_2(\langle In_2, 1\rangle) = V_2(\langle Out_1, 1\rangle)$

$\langle In_1, 1\rangle C_2 \langle Out_1, 1\rangle; \langle In_1, 1\rangle C_2 \langle Out_1, 2\rangle; \langle In_2, 1\rangle C_2 \langle Out_1, 1\rangle; \langle Out_1, 1\rangle C_2 \langle Out_1, 2\rangle.$

The composition (NET_2) of the processes D_1, D_2, *Merge* and P_2 using the scenario set operators result in the following scenario (again specific scenarios which bring out the difference between two compositions are used):

$E_1 = (\langle In_1, 1\rangle, \langle In_2, 1\rangle, \langle Out_1, 1\rangle, \langle Out_1, 2\rangle)$

$V_1(\langle In_1, 1\rangle) = V_1(\langle Out_1, 1\rangle); V_1(\langle In_2, 2\rangle) = V_1(\langle Out_1, 2\rangle)$

$\langle In_1, 1\rangle C_1 \langle Out_1, 1\rangle; \langle In_2, 1\rangle C_1 \langle Out_1, 1\rangle; \langle Out_1, 1\rangle C_1 \langle Out_1, 2\rangle$

$E_2 = (\langle In_1, 1\rangle, \langle In_2, 1\rangle, \langle Out_1, 1\rangle, \langle Out_1, 2\rangle)$

$V_2(\langle In_1, 1\rangle) = V_2(\langle Out_1, 2\rangle); V_2(\langle In_2, 1\rangle) = V_2(\langle Out_1, 1\rangle)$

$\langle In_1, 1\rangle C_2 \langle Out_1, 1\rangle; \langle In_2, 1\rangle C_2 \langle Out_1, 1\rangle; \langle Out_1, 1\rangle C_2 \langle Out_1, 2\rangle.$

These are two of the possible scenarios in the resulting scenario sets of NET_1 and NET_2. Note that, In_1 and In_2 refer to the input ports of the network, and Out_1 refers to the output port of the network, and that connection and labeling operations are also applied.

The two compositions of NET_1 and NET_2 are clearly different. Looking at the causality relation, we see that NET_2 does not produce output until two tokens are used. While NET_1 produces token even after reading one token. The scenario model presents a technique which is adequate to characterize a network of processes (including nondeterministic compositions), but it does not provide a proof system. Also, this requires enumeration of all the possible events and their ordering, making it tedious to manage in a large system.

C. Behavior Model

It is difficult to specify liveness properties in trace-based models. Since trace is a finite sequence, it is difficult to specify a process whose computation could possibly be infinite (a process that does not terminate). In order to be able to better specify liveness properties, Nguyen *et al.* [3] presented an extension to traces leading to a behavior model. The model is capable of handling both synchronous and asynchronous communications. They also presented a temporal proof system that is compositional. Time is totally ordered in this model.

An observation is a recording of the events on input and output ports in a network of processes up to some point of execution, along with the current status of the network ports. The status of the network port is viewed through three functions: *In*, *Out*, *Rd*. The function *In* (*Out*) maps the input (output) ports to true or false, indicating the readiness of the port for communication. The function *Rd* gives the number of events that have occurred on a port during an execution. The sequence of observations during some execution of the network is called the communication behavior of the network, and characterizes an execution of the network. A set of communications behavior characterizes the network.

A process is described by its input and output ports: $P(i_1, \cdots, i_m; j_1, \cdots, j_n)$. A network description is obtained by parallel composition of its constituent processes, (with disjoint) port labels. A specification of a network N is of the form: $\langle N \rangle A$, where N is the network description and A is a temporal assertion. For example, a (slow) terminal which displays the key entered, a character at a time, is specified

as

$$\langle TERM \rangle \ \Box\ (display \sqsubseteq key, \wedge\ (In(key) = \neg\ Out(display) = (|display| = Rd(key)))) \wedge \forall m\ (\Diamond\ |key| = m \Rightarrow \Diamond\ Rd(key) = m) \wedge \forall n\ (\Diamond\ Rd(key) = n \Rightarrow \Diamond\ |display| = n)$$

The specification reads as follows: It is always ($\Box$) true that the character displayed is a prefix ($\sqsubseteq$) of the key entered, indicated by "$display \sqsubseteq key$"; and when the key is ready to be pressed, $In(key)$, the display is not ready, $\neg\ Out(display)$, and also that the number of characters displayed is equal to the number of keys accepted, ($|display| = Rd(key)$); and for every nonnegative integer m, if eventually ($\Diamond$) the number of keys entered equals m, then eventually the number of keys accepted is also equal to m; and for every nonnegative integer n, if eventually the number of keys accepted is equal to n, then eventually the number of characters displayed is also equal to n. There is one more temporal operator, *Until* ($\mathcal{U}$), which we will use later. *Until* is a binary operator; $a\ \mathcal{U} b$ means that a is true until b becomes true. Details of temporal logic can be found in [10].

The proof system consists of the following six parts:

1) Axioms and inference rules describing the domain of values that can appear in events.
2) Axioms and rules for temporal logic.
3) Axioms that define the properties of communication behavior.
4) Axioms describing the liveness assumptions.
5) A set of process specifications.
6) Proof rules for deriving network specification.

We present, informally, the axioms describing the communication behavior. Initially a trace is empty and it is extended upon the occurrence of an event. A trace can be extended by only one element at a time; that is, only one event recording can be done at a time. The trace extension is order preserving. An event can occur only when an input port or an output port is ready to communicate. The number of events read on an input port is nondecreasing and cannot exceed the number of events on that input port; that is, an event cannot be read before its occurrence.

1) Proof Rules: Three proof rules are provided to derive a network specification form component processes: Renaming, Network formation, and Consequence. The renaming rule is provided to circumvent the restriction of disjoint port names in parallel composition.

$$\frac{\langle N \rangle P}{\langle N[p_1, \cdots, p_m/q_1, \cdots, q_m] \rangle P[p_1, \cdots, p_m/q_1, \cdots, q_m]}$$

The effect of the rule is to simultaneously substitute port names $p_1, \cdots, p_m$ with $q_1, \cdots, q_m$, with the restriction that no new link is formed as a result of substitution.

The network formation rule is the primary rule for composing a network from smaller networks (or processes).

$$\frac{\langle N_k \rangle P_k,\ k = 1, \cdots, n}{\langle \| (\cdots, N_k, \cdots) \rangle_k P_k}$$

where N_k satisfies the unique port name requirement.

The consequence rule is

$$\frac{\langle N \rangle P,\ P \Rightarrow Q}{\langle N \rangle Q}$$

where $P \Rightarrow Q$ can be proved from the axioms and inference rules for temporal logic, the axioms and inference rules for the data domain, and the axioms that characterize behaviors. This rule is applied to indicate port connections.

2) Example: Once again, we will describe the example of section I-A using the behavior model. The process D_1 has the following specification:

$$\langle D_1 \rangle\ \Box\ j \sqsubseteq [i(1), i(1)] \wedge (\Diamond\ |i| = u \Rightarrow \Diamond\ |j| = 2 * \min(u, 1))$$

The process D_2 has the following specification:

$$\langle D_2 \rangle\ \Box\ n \sqsubseteq [m(1), m(1)] \wedge (\Diamond\ |m| = u \Rightarrow \Diamond\ |n| = 2 * \min(u, 1))$$

The process *Merge* which reads values from j and n and nondeterministically merges them on k is specified as

$$\langle Merge \rangle\ \Box\ \text{preshuffle}(j, n, k) \wedge (\Diamond\ (|j| = u \wedge |n| = v) \Rightarrow |k| = u + v)$$

where preshuffle$(j, n, k) \sqsubseteq$ merge$(j.X, n.Y)$ (see section I-A).

The process P_1 has the following specification:

$$\langle P_1 \rangle\ \Box\ \lfloor [k(1), k(2)] \wedge (\Diamond\ |k| = 1 \Rightarrow \Diamond\ |\lfloor| = 1) \wedge (\Diamond\ |k| \geq 2 \Rightarrow \Diamond\Box\ |\lfloor| = 2)$$

The process P_2 has the following specification:

$$\langle P_1 \rangle\ \Box\ \lfloor [k(1), k(2)] \wedge (\Diamond\ |k| = 1 \Rightarrow \Diamond\ |\lfloor| = 1) \wedge (\Diamond\ |k| \geq 2 \Rightarrow \Diamond\Box\ |\lfloor| = 2)$$

NET_1, the composition of processes D_1, D_2, *Merge*, and P_1 has the following specification:

$$\langle NET_1 \rangle\ \Box\ \text{preshuffle}([i(1), i(1)], [m(1), m(1)], \lfloor) \wedge (\Diamond\ (|i| + |m| \geq 1) \Rightarrow \Diamond\Box\ |\lfloor| = 2) \wedge ((i = \epsilon\ \mathcal{U}\ \lfloor \neq \epsilon) \Rightarrow \Diamond\ \lfloor(1) = m(1)) \wedge ((m = \epsilon\ \mathcal{U}\ \lfloor \neq \epsilon) \Rightarrow \Diamond\ \lfloor(1) = i(1))$$

NET_2, the composition of processes D_1, D_2, *Merge*, and P_2 has the following specification:

$$\langle NET_2 \rangle\ \Box\ \text{preshuffle}([i(1), i(1)], [m(1), m(1)], \lfloor) \wedge (\Diamond(|i| + |m| \geq 1) \Rightarrow \Diamond\Box\ |\lfloor| = 2) \wedge ((i = \epsilon\ \mathcal{U}\ \lfloor \neq \epsilon) \Rightarrow \Diamond\ \lfloor(1) = [m(1), m(1)]) \wedge ((m = \epsilon\ \mathcal{U}\ \lfloor \neq \epsilon) \Rightarrow \Diamond\ \lfloor(1) = [i(1), i(1)])$$

Proofs of the compositions are given in [3].

III. Dataflow Approach

In the previous section we have presented three trace based models. In the Behavior model and CSP, time is totally ordered. Parallelism is not accurately represented as ordering is induced, even on concurrent parallel events. In the Scenario model time is partially ordered. The specification and composition of processes is cumbersome. In this section, we present another trace-based model with partial

ordering of time which is concise. The computation model is based on the dataflow concept and is currently being developed at the University of Texas at Arlington. Dataflow graph algebra along with a proof system is presented in this section. Dataflow suits well for modeling concurrent and distributed processes because of the asynchronous nature of interaction among processes.

A. Review of Dataflow Concepts

A dataflow graph is a bipartite directed graph in which the two types of nodes are actors and links [11]. Nodes are connected through arcs, which denote the communication media used for transferring information in the form of tokens. The actors represent the operations to be performed while the links receive tokens from one actor and transmit them to another actor. The execution of a dataflow graph advances through firing or enabling of actors. Formally, a dataflow graph G is

$$G = (A \cup L, E)$$

where $A = \{a_1, a_2, \cdots, a_n\}$ is the set of actors

$L = \{l_1, l_2, \cdots, l_m\}$ is the set of Links

$E \subseteq (A \times L) \cup (L \times A)$ is the set of edges.

The set of inputs to the graph and the constants needed to initiate the execution of a graph constitute the *starting set*. The set of links which produce the outputs from the graph is called the *terminating set*. These represent the links through which a graph can effect the environment. Formally, the starting and the terminating set are defined as

$$S = \{l \in L \mid (a, l) \notin E, \forall a \in A\}$$

$$T = \{l \in L \mid (l, a) \notin E, \forall a \in A\}.$$

The state of a dataflow graph can be defined using the location of data or the enabled actors. Markings serve this function in our model. A marking M is a mapping from links to an element of the set of natural numbers, (indicating the number of tokens on the link, for each link in the graph G).

$M: L \rightarrow \aleph$, where $\aleph$ is the set of natural numbers.

A link is said to contain a token in a marking M if $M(l) \geq 1$. An initial marking M_s is a marking in which a subset of the starting set of links contains tokens. A terminal marking M_t is a marking in which a subset of the terminating set of links contains tokens.

Input (output) firing semantic set refers to the subset of the input (output) links which must contain tokens in order to fire an actor. These firing semantic sets enable the representation of choice on input and output links. An actor is firable in a marking M if the links belonging to the input firing semantic set contain tokens and the links belonging to the output firing semantic set contain at most $k - 1$ tokens, where k is a bound on the number of tokens that can be held at a link. When an actor is fired, tokens from the input firing semantic set are removed (or consumed) and new tokens are added to each of the links in the output firing semantic set. This firing action results in a new marking M', indicated by $M \xrightarrow{a} M'$.

A firing sequence σ is a sequence of actors in the order in which they are enabled. If actors can be enabled concurrently, the order is arbitrary. An actor a is said to belong to σ if a is fired at least once in the firing sequence σ. If a new marking M' is derived from the marking M, M is said to lead to M' via σ. This is denoted by $M \xrightarrow{\sigma} M'$. The set of markings generated by a firing sequence σ is denoted by M^σ.

$$M^\sigma = \{M' \mid M \xrightarrow{\Sigma} M' \text{ for any subsequence } \Sigma \text{ that is a prefix of } \sigma\}.$$

For more detailed description of the dataflow graph models see [11], [5].

B. Marking Model

A process is represented by the elements through which it affects the environment. A dataflow graph represents the computation performed by the process. In the subsequent discussion the terms graph and process are used interchangeably. The starting set and the terminating set are the elements through which a process can affect the environment. Traces augmented with the markings characterize the network behavior.

A trace is a sequence of tuples representing the values occurring on the links during an execution of the graph. Our augmented traces have the property that they are totally ordered on a link and partially ordered over a set of links. Firing of a graph extends the trace on the output links of the graph.

1) Dataflow Algebra: Three operators, *graph composition, link connection*, and *link relabeling*, are defined to compose a dataflow graph. These operators are similar to those defined by Brock [9], and are sufficient to describe and construct any dataflow graph using the actors defined. The graph composition operator ($\|$) associates two graphs, each with disjoint starting and terminating sets. This operator is used to construct a graph from smaller graphs (or actors).

The link connection ($\rightarrow$) operator is applied to a graph to connect two links. This operation takes two links, one from the terminating set and the other from the starting set, and establishes an edge while hiding the connection. This operation is used to route the results from one graph (process) to another. Since the graph composition requires graphs with disjoint starting and terminating sets, a conflict in labeling may exist. To avoid this problem, we use the link relabeling operation ($\backslash$). This operation requires that the new label be distinct from the other links in the graph.

Figure 4 shows a dataflow graph for computing the Euclidean distance between two points. We construct the graph first by defining each actor itself as a graph by itself. Then we use the dataflow graph algebra to compose the complete graph shown. The following illustrates the steps involved:

$$l_3 \rightarrow l_4 [\text{Sub}_1(\{l_1, l_2\}, \{l_3\}) \| l_1, l_2, l_3 \backslash l_4, l_5, l_6 \times [\text{Dup}_1(\{l_1\}, \{l_2, l_3\})]] - l_3$$

$$\rightarrow l_4[\text{Sub}_1(\{l_1, l_2\}, \{l_3\}) \| \text{Dup}'_1(\{l_4\}, \{l_5, l_6\})] - l_3$$

$$\rightarrow l_4[\text{Sub}'_1(\{l_1, l_2, l_4\}, \{l_3, l_5, l_6\})] - \text{Sub}'_1(\{l_1, l_2\}, \{l_5, l_3\})$$

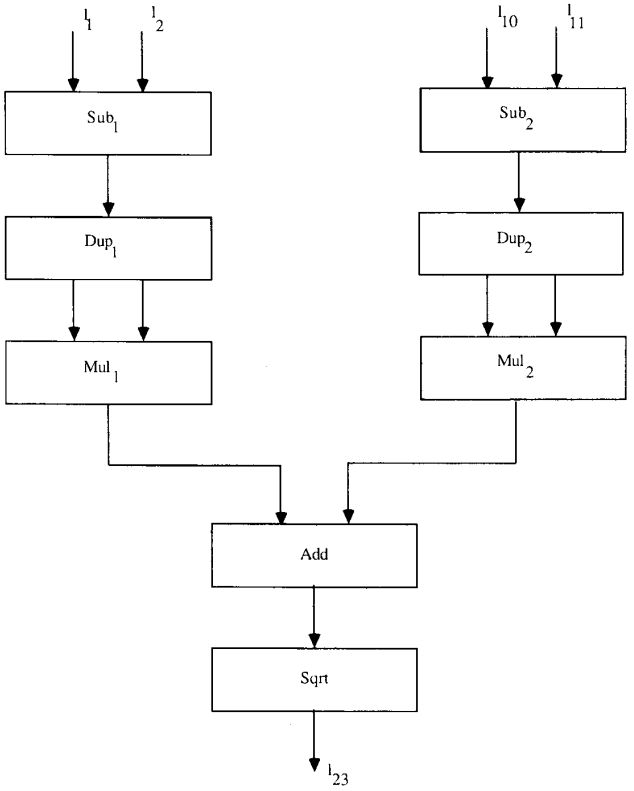

Fig. 4. A graph to compute the distance between two points.

$$l_5, l_6 \rightarrow l_7, l_8[\mathrm{Sub}_1'(\{l_1, l_2\}, \{l_5, l_6\}) \parallel l_1, l_2, l_3 \backslash l_7, l_8, l_9 \times [\mathrm{Mul}_1(\{l_1, l_2\}, \{l_3\})]] - \mathrm{Sub}_1''(\{l_1, l_2\}, \{l_9\})$$

Similarly, $\mathrm{Sub}_2''(\{l_{10}, l_{11}\}, \{l_{18}\})$.

Finally, $\mathrm{Distance}(\{l_1, l_2, l_{10}, l_{11}\}, \{l_{23}\})$.

C. Proof System

The elements of our model are graphs (or processes), links and markings. We use the first-order logic as our assertion language. The only free variables in any assertion are the variables representing the sequence of token values on a link. A marking restricted to a link l is indicated by $M(l)$. We allow functions which represent the computation by the graph. The cardinality symbol $\|$ gives the length of the sequence on a link.

A graph specification is of the form $[G(S, T)]P$, where S is the starting set, T is the terminating set and P is the assertion for the graph G. The links belonging to the starting set and the terminating set are the only links occurring in the assertion. A valid specification is one in which all sequences of values on links satisfy the assertion. A specification is said to be precise if it is valid and if all sequences of values on the links satisfying the assertion are possible sequences.

Examples of simple specifications:
Single Slot Buffer

$$[Buffer_1(\{In\}, \{Out\})]\ \forall M\ \exists M'(M(In) \Rightarrow (M'(In) = M(In) - 1) \wedge (M'(Out) = M(Out) + 1)) \wedge \forall M((M(In) \leq 1) \wedge (M(Out) \leq 1) \wedge Out \sqsubseteq In$$

Multiple Slot Buffer

$$[Buffer_2(\{In\}, \{Out\})]\ \forall M\ \exists M'(M(In) \Rightarrow (M'(In) = M(In) - 1) \wedge (M'(Out) = M(Out) + 1)) \wedge \forall M((M(In) \leq n) \wedge (M(Out) \leq n) \wedge Out \sqsubseteq In$$

The components of the proof system in our model are

1) Axioms and inference rules of first-order logic
2) Axioms that describe properties of traces
3) Axioms about the domain of values contained in the tokens.
4) A set of precise specifications
5) A set of proof rules.

1) Proof Rules: The proof rules correspond to the operators of the dataflow graph algebra and are as follows (assertions are indicated by P's):

Composition Rule ($\|$): This rule allows us to compose a graph from smaller graphs.

$$\frac{\| [G_i(S_i, T_i)]P_i}{[G'(\cup_i S_i, \cup_i T_i)]P}$$

where G' is the new graph obtained by taking the union of starting and terminating sets respectively.

Relabeling Rule ($\setminus$): This rule enables us to rename a link in the graph.

$$\frac{l_i \setminus l_j [G(S, T)]P}{[G'(S - \{l_i\} \cup \{l_j\}, T - \{l_i\} \cup \{l_j\})]P'}$$

where A' is obtained by replacing all occurrences of l_i by l_j.

Connection Rule ($\rightarrow$): This rule is used to establish a connection in the graph while hiding the links involved. By connecting a link from the terminating set to a link in the starting set, we are transferring the assertions on the link from the terminating set to the link in the starting set.

$$\frac{l_i \rightarrow l_j [G(S, T)]P / [p(l_k) \Rightarrow p(l_i)]}{[G'(S - \{l_i\}, T - \{l_i\})]P' / [p(l_k) \Rightarrow p(l_i \setminus l_j)]}$$

where $p(l_i)$, $p(l_j)$ and $p(l_k)$ are terms involving l_i, l_j and l_k respectively.

D. Example

We will now describe the example of Section I-A using our dataflow algebra. The process D_1 has the following specification:

$$[D_1(\{l_1\}, \{l_2\})]\ \forall M\ \exists M'(M(l_1) \geq 1 \Rightarrow (M'(l_1) = M(l_1) - 1) \wedge (M'(l_2) = M(l_2) + 2)) \wedge l_2 \sqsubseteq l_1, l_1$$

The process D_2 has the following specification:

$$[D_2(\{l_3\}, \{l_4\})]\ \forall M\ \exists M'(M(l_3) \geq 1 \Rightarrow (M'(l_3) = M(l_3) - 1) \wedge (M'(l_4) = M(l_4) + 2)) \wedge l_4 \sqsubseteq l_3, l_3$$

The process *Merge* which nondeterministically merges inputs from l_2 and l_4 and writes the value on output l_5:

$$\begin{aligned}[Merge(\{l_2, l_4\}, \{l_5\})]\ &\forall M\ \exists M'((M(l_2) \geq 1 \wedge M(l_4) = 0) \Rightarrow (M'(l_2) = M(l_2) - 1) \wedge (M'(l_5) = M(l_5) + 1) \wedge (l_5 = l_2)) \\ &\forall M\ \exists M'((M(l_4) \geq 1 \wedge M(l_2) = 0) \Rightarrow (M'(l_4) = M(l_4) - 1) \wedge (M'(l_5) = M(l_5) + 1) \wedge (l_5 = l_4)) \\ &\forall M\ \exists M'((M(l_2) \geq 1 \wedge M(l_4) \geq 1) \Rightarrow ((M'(l_2) = M(l_2) - 1) \vee (M'(l_4) = M(l_4) - 1)) \wedge (M'(l_5) = M(l_5) + 1)) \wedge (l_5 \sqsubseteq merge(l_2, l_4))\end{aligned}$$

where l_2 and l_4 are histories, and $merge(l_2, l_4)$ is as defined in Section I-A.

The process P_1 has the following specification:

$$[P_1(\{l_5\}, \{l_6\})]\ \forall M\ \exists M'(M(l_5) \geq 1 \Rightarrow (M'(l_5) = M((l_5) - 1) \wedge (M'(l_6) = M(l_6) + 1)) \wedge l_6 \sqsubseteq l_5 \wedge |l_6| \leq 2$$

The process P_2 has the following specification:

$$[P_2(\{l_5\}, \{l_6\})]\ \forall M, M'\ \exists M''((M(l_5) \geq 1) \wedge (M'(l_5) \geq 1) \Rightarrow (M''(l_5) = M'(l_5) - 2) \wedge (M'(l_6) = M(l_6) + 2)) \wedge l_6 \sqsubseteq l_5, l_5 \wedge |l_6| \leq 2$$

NET_1, composition of D_1, D_2, *Merge*, and P_1, has the following specification:

$$\begin{aligned}[NET_1(\{l_1, l_3\}, \{l_6\})]\ \forall M\ \exists M'((&M(l_1) \geq 1 \vee M(l_3) \geq 1) \\ &\Rightarrow (M'(l_6) = M(l_6) + 1)) \\ &\wedge \forall M'\ \exists M''((M'(l_3, l_6) = 0 \\ &\wedge M''(l_6) = 1) \\ &\Rightarrow l_6 = l_1) \\ &\wedge \forall M'\ \exists M''((M'(l_1, l_6) = 0 \\ &\wedge M''(l_6) = 1) \\ &\Rightarrow l_6 = l_3) \\ &\wedge (l_6 \sqsubseteq merge([l_1, l_1], [l_3, l_3])) \\ &\wedge |l_6| \leq 2\end{aligned}$$

NET_2, composition of D_1, D_2, *Merge*, and P_2, has the following specification:

$$\begin{aligned}[NET_2(\{l_1, l_3\}, \{l_6\})]\ \forall M\ \exists M'((&M(l_1) \geq 1 \vee M(l_3) \geq 1) \\ &\Rightarrow (M'(l_6) = M(l_6) + 2)) \\ &\wedge \forall M'\ \exists M''((M'(l_3, l_6) = 0 \\ &\wedge M''(l_6) = 1) \\ &\Rightarrow l_6, l_6 = l_1, l_1) \\ &\wedge \forall M'\ \exists M''((M'(l_1, l_6) = 0 \\ &\wedge M''(l_6) = 1) \\ &\Rightarrow l_6, l_6 = l_3, l_3) \\ &\wedge (l_6 \sqsubseteq merge([l_1, l_1], [l_3, l_3])) \\ &\wedge |l_6| \leq 2\end{aligned}$$

Like the Scenario model and Behavior model, our Marking model distinguishes NET_1 from NET_2. The proof for the composition is given in Appendix II.

Partially ordered time provides a more natural abstraction for parallel or distributed processing. For example, if the processes D_1 and D_2 are executed on different processors, then the events in D_1 and D_2 can occur independently. When time is totally ordered, as in CSP and Behavior models, sequencing between parallel or independent events is artificially induced. This is evident in the specification of NET_1 and NET_2 where only one input event can

take place between the processes D_1 and D_2. Also total ordering of time implies that the underlying execution model is sequential. In the Scenario and Marking models, the underlying model of execution is dataflow, which allows for parallel execution. The Scenario model requires that the causality relation between the events be specified. In large systems, the number of events could be very large, making it difficult to specify and track each relation. In the Marking model, markings provide a concise way of defining causality relations. Also, synchronization of events can be represented by specifying the conditions under which an actor (or graph) can fire. In the complete work [5], we present a technique for decomposing large dataflow graphs into smaller graphs, where the proof of the large graph can be obtained from the proof of the smaller graphs. This is particularly important while dealing with proof of complex parallel (and distributed) processing software.

IV. Conclusion

Testing of parallel programs is inherently limited by the number of test cases required. Formal specification and verification present an alternative that can be used to reason about programs and, prove properties of the programs even before they are written. In this paper, we have presented trace-based models to characterize the behavior of parallel programs. In CSP, time is totally ordered and the model of computation at the lowest level is sequential. In the Scenario model, time is partially ordered but requires enumeration of all the possible events, making it tedious to handle in large systems. Also, no compositional proof system is available. In the Behavior model, time is totally ordered but contains a compositional proof system. In the Marking model, time is partially ordered and provides a compositional proof system. Currently work is being done to detect deadlocks and handle process termination, using the Marking model.

Although the examples used here are simple, and describe interactions between concurrent processes executing on parallel processors, the formal methods presented here are applicable to functional verification of computational description of the programs constituting the concurrent processes.

Appendix I

Proof for the composition of NET_1 and NET_2 based on CSP. The process *Merge*, when expanded for two inputs, has the following behavior:

$$\begin{aligned}
Merge &= (j?v_1 \rightarrow k!v_1 \rightarrow j?v_2 \rightarrow k!v_2 \,[]\, j?v_1 \rightarrow k!v_1 \rightarrow n?u_1 \rightarrow k!u_1 \,[]\\
&\qquad n?u_1 \rightarrow k!u_1 \rightarrow n?u_2 \rightarrow k!u_2 \,[]\, n?u_1 \rightarrow k!u_1 \rightarrow j?v_1 \rightarrow k!v_1)\\
D_1 \,\|\, D_2 \,\|\, Merge \,\|\, P_1 &\backslash \{j, k, n\}\\
&= (i?v_1 \rightarrow j!v_1 \rightarrow j!v_1) \,\|\, (m?u_1 \rightarrow n?u_1 \rightarrow n!u_1) \,\|\\
&\quad (j?v_1 \rightarrow k!v_1 \rightarrow j?v_2 \rightarrow k!v_2 \,[]\, j?v_1 \rightarrow k!v_1 \rightarrow n?u_1 \rightarrow k!u_1 \,[]\\
&\quad n?u_1 \rightarrow k!u_1 \rightarrow n?u_2 \rightarrow k!u_2 \,[]\, n?u_1 \rightarrow k!u_1 \rightarrow j?v_1 \rightarrow k!v_1) \,\|\\
&\quad (k?w_1 \rightarrow l!w_1 \rightarrow k?w_2 \rightarrow l!w_2 \rightarrow Stop)\backslash\{j, k, n\}
\end{aligned}$$

(Matching the two outputs following events in channels i and m, and concealing j, n)

$$\begin{aligned}
&= (i?v_1 \rightarrow k!v_1 \rightarrow k!v_2 \,[]\, i?v_1 \rightarrow k!v_1 \rightarrow m?u_1 \rightarrow k!u_1 \,[]\\
&\quad m?u_1 \rightarrow k!u_1 \rightarrow k!u_2 \,[]\, m?u_1 \rightarrow k!u_1 \rightarrow i?v_1 \rightarrow k!v_1) \,\|\\
&\quad (k?w_1 \rightarrow l!w_1 \rightarrow k?w_2 \rightarrow l!w_2 \rightarrow Stop)\backslash\{k\}
\end{aligned}$$

(Matching the two outputs on channel k, and hiding the internal events)

$$\begin{aligned}
&= (i?v_1 \rightarrow l!w_1 \rightarrow l!w_2 \rightarrow Stop \,[]\, i?v_1 \rightarrow l!w_1 \rightarrow l!w_2 \rightarrow Stop \,[]\\
&\quad m?u_1 \rightarrow l!w_1 \rightarrow l!w_2 \rightarrow Stop \,[]\, m?u_1 \rightarrow l!w_1 \rightarrow l!w_2 \rightarrow Stop)\\
&= ((i?v_1 \,[]\, m?u_1) \rightarrow l!w_1 \rightarrow l!w_2 \rightarrow Stop)\\
D_1 \,\|\, D_2 \,\|\, Merge \,\|\, P_2 &\backslash \{j, k, n\}\\
&= (i?v_1 \rightarrow j!v_1 \rightarrow j!v_1) \,\|\, (m?u_1 \rightarrow n!u_1 \rightarrow n!u_1) \,\|\\
&\quad (j?v_1 \rightarrow k!v_1 \rightarrow j?v_2 \rightarrow k!v_2 \,[]\, j?v_1 \rightarrow k!v_1 \rightarrow n?u_1 \rightarrow k!u_1 \,[]\\
&\quad n?u_1 \rightarrow k!u_1 \rightarrow n?u_2 \rightarrow k!u_2 \,[]\, n?u_1 \rightarrow k!u_1 \rightarrow j?v_1 \rightarrow k!v_1) \,\|\\
&\quad (k?w_1 \rightarrow k?w_2 \rightarrow l!w_1 \rightarrow l!w_2 \rightarrow Stop)\backslash\{j, k, n\}
\end{aligned}$$

(Matching the two outputs following events in channels i and m, and concealing j, n)

$$\begin{aligned}
&= (i?v_1 \rightarrow k!v_1 \rightarrow k!v_2 \,[]\, i?v_1 \rightarrow k!v_1 \rightarrow m?u_1 \rightarrow k!u_1 \,[]\\
&\quad m?u_1 \rightarrow k!u_1 \rightarrow k!u_2 \,[]\, m?u_1 \rightarrow k!u_1 \rightarrow i?v_1 \rightarrow k!v_1) \,\|\\
&\quad (k?w_1 \rightarrow k?w_2 \rightarrow l!w_1 \rightarrow l!w_2 \rightarrow Stop)\backslash\{k\}
\end{aligned}$$

(Matching the two outputs on channel k, and hiding the internal events)

$$= (i?v_1 \rightarrow [!w_1 \rightarrow [!w_2 \rightarrow Stop \; [] \; i?v_1 \rightarrow [!w_1 \parallel [!w_2 \rightarrow Stop \; []$$

$$m?u_1 \rightarrow [!w_1 \rightarrow [!w_2 \rightarrow Stop [] \; m?u_1 \rightarrow [!w_1 \rightarrow [!w_2 \rightarrow Stop)$$

$$= ((i?v_1 \; [] \; m?u_1) \rightarrow [!w_1 \rightarrow [!w_2 \rightarrow Stop)$$

Appendix II

Proof for composition of NET_1 and NET_2 based on Marking model. Composition of $D1$, $D2$, *Merge* and $P1$.

1st line of specification:
(i) If $M(l_1) \geq 1$ and $M(l_3) = 0$

1) $M'(l_2) = M(l_2) + 2$	From $D1$, line 1
2) $M'(14) = 0$	$M(13) = 0$
3) $M''(l_5) = M'(l_5) + 1$	From *Merge*, line 1 & (1)
4) $M'''(l_6) = M''(l_6) + 1$	From $P1$ & (3)

(ii) If $M(l_3) \geq 1$ and $M(l_1) = 0$
Similar to (i), with 12 and 14 interchanged.
(iii) If $M(l_1) \geq 1$ and $M(l_3) \geq 1$

5) $M'(l_2) = M(l_2) + 2$	From D_1, line 1
6) $M'(l_4) = M(l_4) + 2$	From D_2, line 1
7) $M''(l_5) = M'(l_5) + 1$	From *Merge*, line 3; (5) & (6)
8) $M'''(l_6) = M''(l_6) + 1$	From $P1$ & (7)

2nd line of specification:

9) $M'(l_5) = M(l_5) + 1$	From (1) & (2) or (5) & (6)
10) $l_5 = l_2$	From (1), (2) and *Merge*, line 1
11) $l_5 = l_1$	From (10) and D_1, line 2
12) $l_6 = l_1$	From (11) and $P1$, line 2

3rd line of specification:
Similar to proof for 2nd line.
4th line of specification:

13) $l_2 \sqsubseteq l_1, l_1$	From D_1, line 2
14) $l_4 \sqsubseteq l_3, l_3$	From D_2, line 2
15) $l_5 \sqsubseteq merge(l_2, l_4)$	From *Merge*, line 4
16) $l_5 \sqsubseteq merge([l_1, l_1], [l_3, l_3])$	From (13), (14) & (15)
17) $l_6 \sqsubseteq merge([l_1, l_1], [l_3, l_3])$	From (16) & P_1, line 2
18) $\|l_6\| \leq 2$	From $P1$, line 2.

Composition of D_1, D_2, *Merge* and P_2.
1st line of specification:
(i) If $M(l_1) \geq 1$ and $M(l_3) = 0$

1) $M'(l_2) = M(l_2) + 2$	From D_1, line 1
2) $M'(l_4) = 0$	$M(l_3) = 0$
3) $M''(l_5) = M'(l_5) + 1$	From *Merge*, line 1 & (1)
4) $M'''(l_5) = M''(l_5) + 1$	From *Merge*, line 1 & (1) ($M'(12) = 2$)
5) $M'''(l_6) = M''(l_6) + 2$	From P_1 & (4)

(ii) If $M(l_3) \geq 1$ and $M(l_1) = 0$
Similar to (i), with l_2 and l_4 interchanged.
(iii) If $M(l_1) \geq 1$ and $M(l_3) \geq 1$

6) $M'(l_2) = M(l_2) + 2$	From D_1, line 1
7) $M'(l_4) = M(l_4) + 2$	From D_2, line 1
8) $M''(l_5) = M'(l_5) + 1$	From *Merge*, line 3; (6) & (7)
9) $M'''(l_5) = M''(l_5) + 1$	From *Merge*, line 3; (6), (7)
10) $M'''(l_6) = M''(l_6) + 1$	From P_1, (8) & (9)

2nd line of specification:

11) $M'(l_5) = M(l_5) + 1$	(8)
12) $M''(l_5) = M'(l_5) + 1$	(9)
13) $l_5 = l_2, l_2$	From (1), (2) & *Merge*, line 3
14) $l_5 = l_1, l_2$	From (10) and D_1, line 2
15) $l_6 = l_1, l_1$	From (11) and P_1, line 2

3rd line of specification:
Similar to proof for 2nd line.
4th line of specification:

16) $l_2 \sqsubseteq l_1, l_1$	From D_1, line 2
17) $l_4 \sqsubseteq l_3, l_3$	From D_2, line 2
18) $l_5 \sqsubseteq merge(l_2, l_4)$	From *Merge*, line 4
19) $l_5 \sqsubseteq merge([l_1, l_1], [l_3, l_3])$	From (15), (16) & (17)
20) $l_6 \sqsubseteq merge([l_1, l_1], [l_3, l_3])$	From (18) & P_1, line 2
21) $\|l_6\| \leq 2$	From P_1, line 2.

Acknowledgment

We would like to thank Arun P. Gupta, Francis J. Bush, and Yojak Vasa for their help in preparing this manuscript, and the referees for their help in reviewing this article, and for their suggestions.

References

[1] K. M. Chandy and J. Misra, *Parallel Program Design: A Foundation*. New York, NY: Addison-Wesley Pub. Co., 1988.
[2] C. A. R. Hoare, *Communicating Sequential Processes*. London: Prentice-Hall International, 1985.
[3] V. Nguyen, A. Demers, D. Gries, and S. Owicki, "A model and temporal proof system for networks of processes," *Distrib. Comput.*, vol. 1, no. 1, pp. 7–25, 1986.
[4] J. D. Brock and W. B. Ackerman, "Scenarios: A model of nondeterminate computation," in *Int. Colloq. on Formalization of Programming Concepts*, pp. 252–259, 1981.
[5] A. K. Deshpande, "A Dataflow Approach to Program Correctness," Ph.D. thesis, Dept. of Computer Science and Engineering, UTA, Arlington, Texas, in progress.
[6] V. Pratt, "On the composition of processes," in *Proc. 9th Symp. Principles of Programming Languages*, pp. 213–223, Jan. 1982.
[7] C. A. R. Hoare, "Communicating sequential processes," *CACM*, vol. 21, no. 8, pp. 666–667, Aug. 1978.
[8] E. W. Dijkstra, "Guarded commands, nondeterminacy, and formal derivation of programs," *CACM*, vol. 18, no. 8, pp. 453–457, Aug. 1975.
[9] J. D. Brock, "A Formal Model of Non-Determinate Dataflow Computation," Ph.D. thesis, Dept. of Electrical Engineering and Computer Science, MIT, Cambridge, Mass., Aug. 1983.
[10] Z. Manna and A. Pnueli, "Verification of concurrent programs: Temporal proof principles," in *Logics of Programs, Lecture Notes in Computer Science*, vol. 131. New York, NY: Springer-Verlag, 1984.
[11] K. M. Kavi, B. P. Buckles, and U. N. Bhat, "A formal definition of data flow graph models," *IEEE Trans. Comput.*, vol. C-35, no. 11, pp. 940–948, Nov. 1986.

Further Reading

The following are not referenced in the paper but provide valuable sources for many of the concepts presented in the paper.

— S. D. Brookes, "A semantics and proof system for communicating processes," in *Lecture Notes in Computer Science*, vol. 164. New York, NY: Springer-Verlag, 1984.
— C. Hewitt and H. G. Baker, "Laws for communicating parallel processes," in *1977 IFIP Congress Proc.*, pp. 987–992, Aug. 1977.
— G. Kahn, "The semantics of a simple language for parallel programming," in *1974 IFIP Congress Proc.*, pp. 471–475, 1974.
— G. Kahn and D. B. MacQueen, "Coroutines and networks of parallel processes," in *1977 IFIP Congress Proc.*, pp. 993–998, Aug. 1977.

— G. M. Levin and D. Gries, "A proof technique for communicating sequential processes," *Acta Informatica*, pp. 281–302, 1981.
— Z. Manna and A. Pnueli, "Verification of concurrent programs, Part I: The temporal framework," in *The Correctness Problem in Computer Science*, R. S. Boyer and J. S. Moore, Eds. *Int. Lecture Series in Computer Science*. London: Academic Press, 1981.
— J. Misra and K. M. Chandy, "Proofs of networks of processes," *IEEE Trans. Software Engineering*, vol. SE-7, no. 4, pp. 417–426, July 1981.

Akshay K. Deshpande (Member, IEEE) is currently a doctoral student at the University of Texas at Arlington. He was assigned to the IBM Thomas J. Watson Research Center as a member of the co-operative student program during the summers of 1985 and 1986. His research interests are in semantics of programs, programming languages, distributed systems, and computer architecture.

Mr. Deshpande is a student member of the IEEE Computer Society and of ACM.

Krishna M. Kavi (Senior Member, IEEE) received the B.E. degree in electrical engineering from the Indian Institute of Science, Bangalore, India, and the M.S. and Ph.D. degrees in computer science from Southern Methodist University, Dallas, TX. He is currently an Associate Professor in Computer Science Engineering at the University of Texas at Arlington. Previously, he was at the University of Southwestern Louisiana, Lafayette. His areas of research interest include computer systems architecture (dataflow, object-based, capability-based, and high-level language systems), performance and reliability modeling of computer systems (using stochastic dataflow and Petri nets), and distributed processing systems (object-based operating systems, languages for distributed processing, fault-tolerance).

Dr. Kavi serves as a member of the IEEE Computer Society Press editorial board, and in the Society's distinguished visitor and Chapter tutorials programs. He is a member of ACM, Sigma Xi, and Upsilon Phi Epsilon.