Data Flow Analysis of Communicating Finite State Machines

WUXU PENG
Southwest Texas State University

and

S. PUROSHOTHAMAN
The Pennsylvania State University

Let $\langle P_1, P_2, \ldots, P_n \rangle$ be a network of n finite state machines, communicating with each other asynchronously using typed messages over unbounded FIFO channels. In this paper we present a data flow approach to analyzing these communicating machines for nonprogress properties (deadlock and unspecified reception). We set up flow equations to compute the set of pending messages in the queues at any given state of such a network. The central technical contribution of this paper is an algorithm to compute approximations to solutions for the ensuing equations. We then show how these approximate solutions can be used to check that interactions between the processes are free of nonprogress errors. We conclude with a number of example protocols that our algorithm certifies to be free of nonprogress errors. Included in the examples is a specification of X25 call establishment/clear protocol.

Categories and Subject Descriptors: C.2.2 [**Computer-Communication Networks**]: Network Protocols—*protocol verification*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*distributed programming*; D.2.4 [**Software Engineering**]: Program Verification; F 3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms: Algorithms, Theory, Verification

Additional Key Words and Phrases: Communicating finite state machines, static analysis

1. INTRODUCTION

A communicating finite state machine is a very useful abstract model for specifying, verifying, and synthesizing communication protocols [4, 8]. In this model processes are expressed as finite state machines that communicate

A preliminary version appeared in ACM PODC 89.
Authors' addresses: S. Purushothaman, Department of Computer Science, The Pennsylvania State University, University Park, PA 16802; W. Peng, Department of Computer Science, Southwest Texas State University, San Marcos, TX 78666.

over (potentially) unbounded buffers by sending and receiving typed messages asynchronously. The buffer models the communication medium and is used to store messages that have been sent but have not yet been received.

A central issue in this model is whether a network of communicating finite state machines is free of progress errors. Three widely addressed progress properties are absence of unspecified receptions, absence of deadlocks, and absence of unbounded communication. The problem of checking for nonprogress in communicating finite state machines is known to be undecidable [4, 9]. In this paper we take a new approach to coping with this undecidable problem; we show how data flow analysis of communicating finite state machines can be carried out to obtain sufficient conditions under which they are free of unspecified receptions and deadlocks. A novel feature of this analysis is its ability to handle networks whose communication demands unbounded buffers.

Let $N = \langle P_1, P_2, \ldots, P_n \rangle$ be a network of n communicating processes. In general, the reachability set of a network can be unbounded; in fact, the class of all reachability sets is recursively enumerable. Given a network N, we compute a superset of the reachability set of N, which is regular. By constructing a finite representation of this superset, we are able to handle both bounded and unbounded buffers with equal ease. This is in contrast to the traditional state exploration techniques in which a reachability graph, similar to the coverability graph of Petri nets, is constructed; for the kinds of networks under consideration, such a reachability graph can be constructed only if the buffers are bounded. In Section 3 we set up the necessary formalism, in Section 4 we show how to set up the necessary data flow equations, and in Section 5 we present an algorithm to solve these equations for a class of networks called strict FIFO networks. We extend these algorithms to the entire class of all communicating finite state machines in Section 6.

Though we concentrate on unspecified reception and deadlock analysis in this paper, the algorithms that we develop can equally well be applied to the reachability and boundedness problems. In Section 7 we present a number of example protocols from the literature, which our algorithm certifies to be free of nonprogress errors. Included in the examples is a specification of an X.25 call establishment/clear protocol as found in the literature [3, 8].

The techniques proposed here suffer from the state explosion problem found in analyzing concurrent systems. But, apart from its ability to handle an unbounded number of states, the worst-case time and space requirements are less than or comparable to those found in the analysis of networks of communicating finite state machines with bounded buffers.

The model of communicating finite state machines has also been used elsewhere. For instance, given a program containing two communicating processes, if we wish to analyze the communication part of the processes then we can safely abstract out the conditions and assignment statements from the program. The result of the abstraction captures the control structure and the communication part of the program, yielding a set of

communicating finite state machines [14]. Such an abstraction introduces internal nondeterminism that can be easily handled. We discuss these issues in Section 8. We conclude in Section 9 with a discussion of future work.

2. PREVIOUS WORK AND MOTIVATION

Previous research on checking for freedom from deadlock and other related path problems has been presented in [9], [13], [18], [20], and [21]. In a series of papers, Gouda et al. [9, 24] investigated the deadlock and unboundedness problem for networks of communicating systems containing two processes. In [18] we investigated the deadlock and unspecified reception problem for two-party protocols under the assumption that the receive commands do not respect their message types.

Reif and Smolka consider the problem of reachability in a network of processes that communicate over unbounded buffers using messages of a single type. With static communication, wherein queue names are statically known, they show that reachability in Petri nets is reducible to reachability in their model. With dynamic communication, they show that reachability is undecidable [20]. They also offer a tractable approximation algorithm that is complete for communicating systems wherein messages cannot be deleted from the queues holding pending messages [21].

The analysis problem (reachability, deadlock, etc.) for synchronous communication has been considered in [13] and [23]. Taylor [23] shows that the analysis problem, though decidable, is intractable. Kanellakis and Smolka [13] consider the nonprogress problem under various assumptions, such as (1) the interconnection of processes is a tree and (2) the processes are acyclic.

Note that communicating systems that have asynchronous communication over bounded buffers can be expressed as a system of processes communicating synchronously. As the deadlock and reachability problems for such systems are decidable, most of the work done on state exploration of bounded buffer asynchronous communication have dealt with reducing the complexity of the state space search. These include, for instance, the empty medium abstraction of Bochmann [3] and the combination of hashing, to manage states that have been visited, and probabilistic search by Holzmann [10].

To put things in perspective, we wish to deal with communicating finite state systems with both bounded buffers and potentially unbounded buffers in a uniform way. Given a specification of a network we do not a priori know if the buffers are being used in a finite way. By being able to deal with unbounded buffers, any errors we may find as a result of our analysis would only have to be due to mismatches in the way messages are sent by one process and received by another process. Put differently, we are able to distinguish between errors caused by mismatches and errors caused by a need for unbounded buffer sizes.

3. DEFINITIONS AND NOTATIONS

In this section we present definitions and notations that are used in the rest of this paper.

3.1 Communicating Finite State Machines

Informally, a communicating finite state machine (CFSM) is a labeled directed graph with a distinguished initial state, where each edge is labeled by an event. The events of a CFSM are **send** and **receive** commands over a finite set of message types $\sum$. The communication between CFSMs is assumed to be asynchronous (i.e., nonblocking sends and blocking receives). Consequently, an infinite FIFO buffer between each pair of machines, to store pending messages, is assumed.

Let $I = \{1, \ldots, n\}$, where $n \geq 2$ is some constant (number of machines in a network). Formally, we have the following definition:

Definition 3.1. (CFSM). A CFSM P_i is a four-tuple $(S_i, \langle \sum_{i,j} \rangle_{j \in I} \cup \langle \sum_{j,i} \rangle_{j \in I}, \delta_i, p_{0i})$, where

—S_i is the set of local states.

—$\sum_{i,j}$ is the set of message types that P_i can send to machine P_j, and $\sum_{j,i}$ is the set of message types that P_i can receive from machine P_j. It is assumed that $\sum_{i,i} = \emptyset$, since P_i cannot directly send messages to or receive messages from itself.

—Let $-\sum_{i,j} = \{-m \mid m \in \sum_{i,j}\}$ and $+\sum_{j,i} = \{+m \mid m \in \sum_{j,i}\}$. δ_i is a partial mapping, δ_i: $S_i \times (\langle \sum_{i,j} \rangle_{j \in I} \cup \langle + \sum_{j,i} \rangle_{j \in I}) \times I \rightarrow 2^{S_i}$. $\delta_i(p, -m, j)$ is the set of new states that machine P_i can possibly enter after sending message of type m to machine P_j, and $\delta_i(p, +m, j)$ is the set of new states that machine P_i can possibly enter after receiving message of type m from machine P_j.

—p_{0i} is the initial local state.

A transition $p' \overset{-m}{\rightarrow} p$ ($p' \overset{+m}{\rightarrow} p$, resp.) in P is called a *send edge* (*receive edge*, resp.). A state p in P_i is said to be a *send* (*receive*, resp.) *state* if and only if (iff) all of its outgoing edges are send (receive, resp.) edges. p is said to be a *mixed state* iff it has both outgoing send and receive edges. Define $RMsg(p)$ and $SMsg(p)$ to be the set of message types that can be received and sent in state p, respectively; that is,

$$RMsg(p) = \{m \mid \exists p' \exists j,\ p' \in \delta_i(p, +m, j)\},$$
$$SMsg(p) = \{m \mid \exists p' \exists j,\ p' \in \delta_i(p, -m, j)\}.$$

By definition, $RMsg(p) = \emptyset$ if p is a send state, and $SMsg(p) = \emptyset$ if p is a receive state. The set of receive states in P_i will be denoted as S_i^r. The terms *node* and *state* are used synonymously. Since the channels are assumed to be FIFO, receive states are the only states that may cause communication errors.

Define the following:

$$\sum_i = \bigcup_{j \in I} (\sum_{i,j} \cup \sum_{j,i}) \qquad \text{messages Process } P_i \text{ has to deal with,}$$

$$\sum = \bigcup_{i \in I} \sum_i \qquad \text{message types in a network,}$$

$$-\Sigma_i = \bigcup_{j \in I} -\Sigma_{i,j} \qquad \text{send events in process } P_i,$$

$$+\Sigma_i = \bigcup_{j \in I} +\Sigma_{j,i} \qquad \text{receive events in process } P_i,$$

$$\pm\Sigma_i = -\Sigma_i \bigcup +\Sigma_i \qquad \text{alphabet of events in process } P_i,$$

$$-\Sigma = \bigcup_{i \in I} -\Sigma_i \qquad \text{send events in a network,}$$

$$+\Sigma = \bigcup_{i \in I} +\Sigma_i \qquad \text{receive events in a network,}$$

$$\pm\Sigma = +\Sigma \bigcup -\Sigma \qquad \text{set of all events in a network.}$$

Without loss of generality, we assume that $\Sigma_{i,j} \cap \Sigma_{k,l} = \emptyset$ if $(i, j) \neq (k, l)$. In practice, we can always ensure this property by, for instance, appending the identity of the sender and the receiver as part of every message. Due to this assumption, for any $a \in \pm\Sigma_i$ we can simplify the notation $\delta_i(p, a, j)$ to $\delta_i(p, a)$. If $\delta_i(p, a) = \emptyset$, then we say that event a is undefined at state p.

Let $P_1, \ldots, P_n$ be n machines that communicate with each other. Let V_N be the Cartesian product of the sets $S_1, \ldots, S_n$, that is, $V_N = S_1 \times \cdots \times S_n$, and let C_N be the Cartesian product of the sets $\Sigma^*_{2,1}, \ldots, \Sigma^*_{n,1}, \Sigma^*_{1,2}, \Sigma^*_{3,2}, \ldots, \Sigma^*_{n,2}, \ldots, \Sigma^*_{1,n}, \ldots, \Sigma^*_{n-1,n}$; that is, $C_N = \Sigma^*_{2,1} \times \cdots \times \Sigma^*_{n,1} \times \Sigma^*_{1,2} \times \cdots \times \Sigma^*_{n,2} \times \cdots \times \Sigma^*_{1,n} \times \cdots \times \Sigma^*_{n-1,n}$.

We are now ready to define the semantics of a network of communicating finite state machines.

Definition 3.2. (Network of Communicating Finite State Machines). A network of communicating finite state machines (NCFSM) is a tuple $N = \langle P_1, \ldots, P_n \rangle$, where each P_i $(i \in I)$ is a CFSM.

A global state of N is a tuple $[\langle p_i \rangle_{i \in I}, \langle c_{i,j} \rangle_{i,j \in I}]$, where p_i is a local state of machine P_i, and $c_{i,j}$ is the sequence of messages in the channel from machine P_j to P_i.

Initially, N is in its initial state $[\langle p_{0i} \rangle_{i \in I}, \langle c_{i,j} \rangle_{i,j \in I}]$, where $c_{i,j} = \varepsilon$ $(i \neq j)$. Let $[\langle P_i \rangle_{i \in I}, \langle c_{i,j} \rangle_{i,j \in I}]$ be a global state. The global state transition function δ_N: $(V_N \times C_N) \times \pm\Sigma \to 2^{V_N \times C_N}$ is a partial function defined as

(1) If $p'_i \in \delta_i(p_i, -m, j)$, then $[\langle p'_k \rangle_{k \in I}, \langle c'_{k,l} \rangle_{k,l \in I}] \in \delta_N([\langle p_k \rangle_{k \in I}, \langle c_{k,l} \rangle_{k,l \in I}], -m)$, where

$$p'_k = \left\{ \begin{array}{ll} p'_i & \text{if } k = i \\ p_k & \text{otherwise} \end{array} \right\} \quad \text{and} \quad c'_{k,l} = \left\{ \begin{array}{ll} c_{k,l} \cdot m & \text{if } (k,l) = (j,i) \\ c_{k,l} & \text{otherwise} \end{array} \right\};$$

(2) If $p'_I \in \delta_i(p_i, +m, j)$, then $[\langle p'_k \rangle_{k \in I}, \langle c'_{k,l} \rangle_{k,l \in I}] \in \delta_N([\langle p_k \rangle_{k \in I}, \langle c_{k,l} \rangle_{k,l \in I}], +m)$, where

$$p'_k = \left\{ \begin{array}{ll} p'_i & \text{if } k = i \\ p_k & \text{otherwise} \end{array} \right\} \quad \text{and} \quad c_{k,l} = \left\{ \begin{array}{ll} m \cdot c'_{k,l} & \text{if } (k,l) = (i,j) \\ c'_{k,l} & \text{otherwise} \end{array} \right\}.$$

We use $P_i \rightarrow P_j$ to denote the channel $c_{j,i}$ from P_i to P_j. In essence, the first case in Definition 3.2 denotes the event that P_i sends a message m to P_j, which causes the message m to be appended to the end of channel $P_i \rightarrow P_j$. The second case represents the event that P_i receives a message of type m sent by P_j, which has the effect of removing the first message (which must be of type m, or an unspecified reception error would occur) in the channel $P_j \rightarrow P_i$. In both cases, after the successful completion of the event, P_i enters local state p_i', while all other machines remain in the same local states and the contents of all other channels are unchanged.

The definition given above is in its full generality since it assumes that there exists a unidirectional channel from each machine P_i to every other machine P_j. We call this network topology *fully connected*. In practice, however, fully connected topology is usually unlikely because of various limitations. If machine P_i cannot directly send messages to some other machine, say, P_j, we can simply set $\sum_{i,j} = \emptyset$. We define the network *topology graph* $TG(N)$ of a network N as a directed graph with the set of sites (finite state machines) as its node set. A directed edge from P_i to P_j is in $TG(N)$ if there exists a unidirectional channel from P_i to P_j.

To simplify the presentation, we use the notation $[\bar{v}, \bar{c}]$ to denote a global state, where, by convention, $\bar{v} = \langle p_i \rangle_{i \in I}$ denotes an n tuple and $\bar{c} = \langle c_{i,j} \rangle_{i,j \in I}$ denotes an $n(n-1)$ tuple. $[\overline{v_0}, \overline{c_0}]$ will be used to denote the initial state. We use the notation $[p_1, \ldots, p_n, c_{1,2}, c_{1,3}, \ldots, c_{n,n-1}]$ whenever we need to refer to the individual state components p_i or to the individual buffers $c_{i,j}$. If $\bar{v}$ is a vector, the notation $\bar{v}|_i$ will be used to denote the i^{th} component of $\bar{v}$. The terms *channel*, *buffer*, and *queue* are used interchangeably throughout this paper.

The global state transition function δ_N can be easily extended to the following reachability function δ_N^*: $(V_N \times C_N) \times \pm \sum^* \rightarrow 2^{V_N \times C_N}$:

(1) $\delta_N^*([\bar{v}, \bar{c}], \varepsilon) = \{[\bar{v}, \bar{c}]\}$,

(2) $\delta_N^*([\bar{v}, \bar{c}], a.e) = \{[\overline{v'}, \overline{c'}] \mid \exists [\overline{v''}, \overline{c''}] \in \delta_N([\bar{v}, \bar{c}], a), [\overline{v'}, \overline{c'}] \in \delta_N^*([\overline{v''}, \overline{c''}], e)\}$.

We often write $\delta_N^*([\overline{v_0}, \overline{c_0}], e)$ as $\delta_N^*(e)$.

We drop the subscripts in δ_N and δ_N^* if no confusion arises. Furthermore, as $\delta^*([\bar{v}, \bar{c}], a) = \delta([\bar{v}, \bar{c}], a)$ for a single event $a \in \pm \sum$, we use δ instead of δ^* from now on. A word e in $\pm \sum^*$ is called an *event sequence*. We say that an event sequence e can *lead* the network from a global state $[\overline{v'}, \overline{c'}]$ to another global state $[\bar{v}, \bar{c}]$ if $[\bar{v}, \bar{c}] \in \delta([\overline{v'}, \overline{c'}], e)$. Sometimes it is more convenient to be able to refer to the individual transitions that contribute events in an event sequence. We use the notation $ts = [\overline{v'}, \overline{c'}] \overset{+}{\rightarrow} [\bar{v}, \bar{c}]$ to denote the transition sequence that leads the network from $[\overline{v'}, \overline{c'}]$ to $[\bar{v}, \bar{c}]$. In this case we use $label(ts)$ to denote the sequence of events that occur on ts. We use the notation $[\overline{v'}, \overline{c'}] \overset{*}{\rightarrow} [\bar{v}, \bar{c}]$ to denote the fact that there exists some event sequence e that can lead the network from $[\overline{v'}, \overline{c'}]$ to $[\bar{v}, \bar{c}]$.

For an event sequence e, let $pref(e)$ be the set of prefixes of e; that is, $pref(e) = \{e_1 \mid e_1, e_2 \in \pm \sum^* \& e_1 e_2 = e\}$. For $a \in \pm \sum$, let $|e|_a$ be the number of occurrences of the event a in e. Define $f_{i,j}$ as a homomorphism from $\pm \sum^*$

to $\pm\Sigma_{i,j}^*$,

$$f_{i,j}(a) = \begin{cases} a & \text{if } a \in \pm\sum_{i,j}, \\ \varepsilon & \text{otherwise.} \end{cases}$$

For an event sequence e, $f_{i,j}(e)$ is the subsequence of the events from e that only affect the channel $P_i \to P_j$.

An event sequence e is *executable* if $\delta(e)$ is defined; that is, if $\delta(e) \neq \emptyset$.

A tuple $\bar{v} \in V_N$ is a *receive* node if every component state p_i of $\bar{v}$ is a local receive state. $RV(N)$ denotes the set of all receive nodes in V_N.

Two event sequences e_1 and e_2 are *equivalent* ($e_1 \simeq e_2$) iff

(1) e_1 and e_2 are different orderings of the same collection of events, and
(2) $\delta(e_1) = \delta(e_2)$.

For a NCFSM $N = \langle P_1, \ldots, P_n \rangle$, the reachability set $RS(N) \subseteq V_N \times C_N$ is the set of all reachable global states, namely,

$$RS(N) = \{[\bar{v}, \bar{c}] \mid [\bar{v}, \bar{c}] \in \delta(e),\ e \in \pm\Sigma^*\}.$$

The reachability graph $RG(N) = (RS(N), E)$ for N is a directed labeled graph with $RS(N)$ as its node set and $E = \{[\bar{v}, \bar{c}] \xrightarrow{a} [\bar{v'}, \bar{c'}] \mid [\bar{v_0}, \bar{c_0}] \xrightarrow{*} [\bar{v}, \bar{c}]$ and $[\bar{v'}, \bar{c'}] \in \delta([\bar{v}, \bar{c}], a)\}$.

Example 1. Figure 1 shows a NCFSM and its reachability graph. Note that $RG(N_1)$ is an infinite graph.

3.2 Shuffle-Products

The concept of shuffle-product is helpful in analyzing NCFSMs since it gives a finite description of all of the possible interleavings of events in a network. More importantly, possible contents of the buffer will be represented as paths in the product graph. We show in Proposition 3.1 that no information is either gained or lost in setting up these new definitions.

We need the following mappings to encode the contents of all of the buffers as a single buffer:

$h_{i,j}$ is a homomorphism from Σ^* to $\Sigma_{j,i}^*$:

$$h_{i,j}(g) = \begin{cases} g & \text{if } g \in \Sigma_{j,i}, \\ \varepsilon & \text{otherwise.} \end{cases}$$

proj is a function from Σ^* to

$$\underbrace{\Sigma_{2,1}^* \times \Sigma_{3,1}^* \times \cdots \times \Sigma_{n,1}^* \times \Sigma_{1,2}^* \times \cdots \times \Sigma_{n,2}^* \times \cdots \times \Sigma_{1,n}^* \times \cdots \times \Sigma_{n-1,n}^*}_{n(n-1)}.$$

For a word $z \in \Sigma^*$,

$$proj(z) = (h_{1,2}(z), h_{1,3}(z), \ldots, h_{1,n}(z), \ldots, h_{n,1}(z), \ldots, h_{n,n-1}(z)).$$

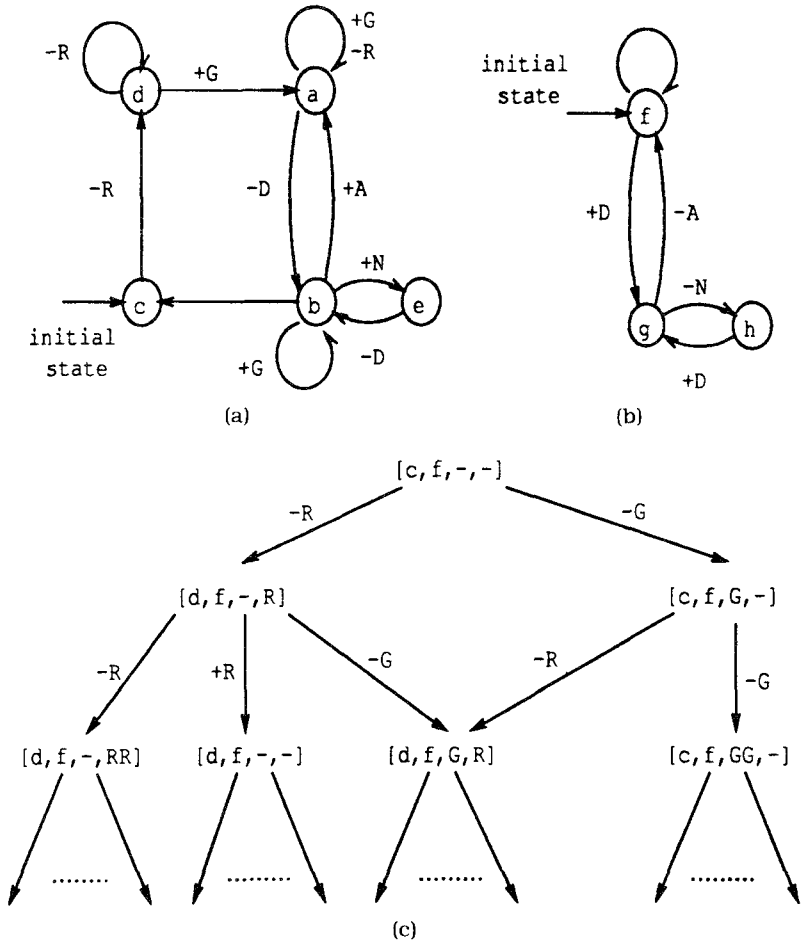


Fig 1. Network N_1 and its reachability graph $RG(N_1)$. (a) Machine P_1; (b) Machine Q_1; (c) reachability graph of $N_1 = \langle P_1, Q_1 \rangle$.

Definition 3.3. (shuffle-product of NCFSMs). Let $N = \langle P_1, \ldots, P_n \rangle$ be a NCFSM. The shuffle-product of N, written as $SP(N)$, is a five-tuple $(V_N, \Sigma, T, \Delta, v_0)$, where

(1) $\overline{v_0} = [p_{01}, p_{02}, \ldots, p_{0n}] \in V_N$.

(2) The finite control part of $SP(N)$ is the transition function T: $V_N \times \pm\Sigma \to 2^{V_N}$ defined as

$$\overline{v'} \in T(\bar{v}, a),$$

where

$$a \in \pm \sum_i, \qquad \text{iff} \quad \overline{v'}|_j = \bar{v}|_j \ (j \in I \& j \neq i) \quad \text{and} \quad \overline{v'}|_i \in \delta_i(\bar{v}|_i, a).$$

(3) A global state in $SP(N)$ is of the form $[\bar{v}, z]$, where $\bar{v} \in V_N$, and $z \in \sum^*$ is the content of the single buffer of the shuffle-product. In particular, $[\overline{v_0}, \varepsilon]$ is the initial global state of $SP(N)$.

(4) The semantics of the shuffle-product is captured by the *semantic* transition function Δ: $(V_N \times \sum^*) \times \pm\sum \to 2^{V_N \times \sum^*}$. Let $[\bar{v}, z]$ be a global state.
   (a) If $\overline{v'} \in T(\bar{v}, -g)$, then $[\overline{v'}, z'] \in \Delta([\bar{v}, z], -g)$, where $-g \in -\sum$ and $z' = z \cdot g$; and
   (b) if $\overline{v'} \in T(\bar{v}, +g)$, then $[\overline{v'}, z'] \in \Delta([\bar{v}, z], +g)$, where $g \in \sum_{j,i}$, $z_1.g.z_2 = z$, $z' = z_1.z_2$, and $h_{i,j}(z) = g.h_{i,j}(z_2)$ (i.e., z_1 does not contain any messages from $\sum_{j,i}$).

In Definition 3.3 send events are still nonblocking. However, to receive a message $g \in \sum_{j,i}$ in the single channel of the shuffle-product, the first message of z that belongs to $\sum_{j,i}$ must be g. In Section 4 networks where the single buffer of the shuffle-product can be treated as a strict FIFO queue are presented; this is in contrast to how the buffer z is being used in part (4b) of Definition 3.3.

The finite control part of the shuffle-product $SP(N)$ can be viewed as a (nondeterministic, in general) *finite state automaton* (*FSA*) by identifying some subset $F \subseteq V$ as the final state set. Such a FSA is called the *shuffle-product automaton* (*SPA*) and will be denoted as $SPA(N, F)$. Sometimes it is convenient to view the finite control part of the shuffle-product simply as a directed labeled graph, which identifies all the transitions (engendered by T) in $SP(N)$ as the set of edges E. We call such a graph the *shuffle-product graph*(*SPG*) and write it as $SPG(N) = (V_N, E)$.

Example 2. Figure 2 shows a network $N_2 = \langle P_2, Q_2 \rangle$ and its shuffle-product.

As in the network case, the semantic transition function Δ can be extended to a function from $(V_N \times \sum^*) \times \pm \sum^*$ to $2^{V_N} \times \sum^*$. Hence, $\Delta(e)$ denotes the set of all global states that a shuffle-product can be led to by the event sequence e. The concepts of reachability and reachability set of the networks carry over to the case of shuffle-products. In particular, $RS(SP(N))$, the reachability set for $SP(N)$, is the set of all global states reachable from $[\overline{v_0}, \varepsilon]$,

$$RS(SP(N)) = \left\{ [\bar{v}, z] \,|\, [\overline{v_0}, \varepsilon] \xrightarrow{*} [\bar{v}, z] \right\}.$$

$RS(SP(N))$ is closely related to $RS(N)$, the reachability set of the network. The relationship between the two is expressed by the following proposition:

PROPOSITION 3.1.

(1) *For a word* $z \in \sum^*$, *if* $[p_1, \ldots, p_n, z] \in RS(SP(N))$ *then* $[p_1, \ldots, p_n, h_{1,2}(z), h_{1,3}(z), \ldots, h_{n,n-1}(z)] \in RS(N)$.
(2) *If* $[p_1, \ldots, p_n, x_{2,1}, x_{3,1}, \ldots, x_{n-1,n}] \in RS(N)$, *then there exists* $[p_1, \ldots, p_n, z] \in RS(SP(N))$ *such that* $proj(z) = (x_{2,1}, x_{3,1}, \ldots, x_{n-1,n})$.

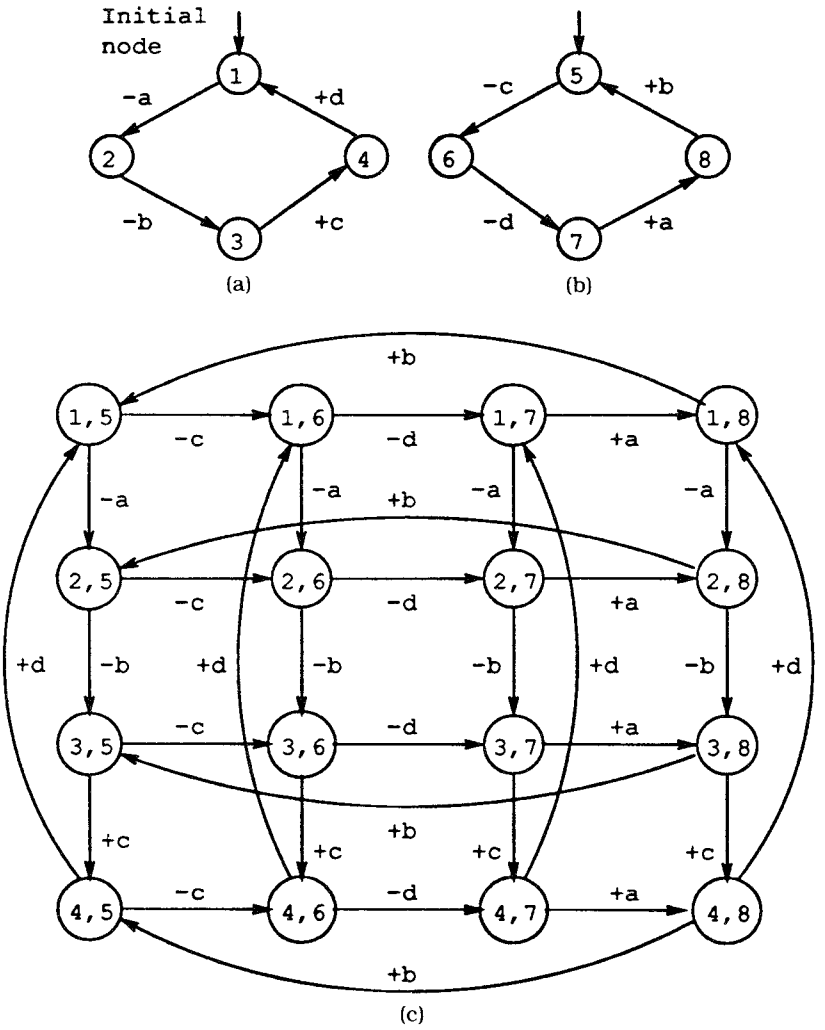


Fig. 2. Network N_2 and its shuffle-product $SP(N_2)$

PROOF. Strictly from the definitions. □

Proposition 3.1 really says that the two reachability sets, $RS(N)$ and $RS(SP(N))$, are *equivalent*, since we can effectively translate one from the other.

3.3 Safety Properties of CFSMs

Given a network of CFSMs, we are interested in finding out whether the network satisfies certain *nice* safety properties. A number of safety properties of CFSMs have received broad attention. We list three of them here, which are of special interest in this paper.

Let Z be the set of nonnegative integers. For a language L, let $INIT(L)$ denote the set of letters that appear as the first letter of some word in L, that is,

$$INIT(L) = \{a \mid a.w \in L\}.$$

Let N be a network of CFSMs and $[\bar{v}, \bar{c}] \in RS(N)$ be a reachable global state. Formally, we say that

(1) $[\bar{v}, \bar{c}]$ is a *deadlock* state if the following predicate holds:

$$(\bar{v} \in RV(N)) \& (\bar{c} = \overline{c_0}).$$

Namely, every machine is in a local receive state, and all channels are empty.

(2) $[\bar{v}, \bar{c}]$ is an *unspecified reception* state if the following predicate is true:

$$\exists i \in I((p_i \in S_i^r) \& \forall j \in I(INIT(c_{i,j}) \notin RMsg(p_i))).$$

(3) The communication of N is *bounded* if the following predicate holds:

$$\exists K \in Z \forall [\bar{v}, \bar{c}] \in RS(N) \forall i, j \in I(|c_{i,j}| \leq K).$$

Otherwise, we say that the communication of N is *unbounded*.

For each of the safety properties, there is a corresponding detection problem:

(1) The *deadlock detection problem* (*DDP*) is, given a network N, is N free of deadlocks?
(2) The *unspecified reception detection problem* (*URDP*) is, given a network N, is N free of unspecified receptions?
(3) The *unbounded detection problem* (*UBDP*) is, given a network N, is the communication of N bounded?

It is well known that, in general, it is undecidable whether a NCFSM is free of deadlocks or unspecified receptions, or has bounded communication [4, 9]. Intuitively, a system of two CFSMs has the same computational power as Turing machines since at least two unbounded FIFO queues are employed in the system. We state this fact in the following theorem:

THEOREM 3.1. *DDP, URDP, and UBDP are undecidable.*

4. DATA FLOW EQUATIONS

Consider the shuffle-product $SP(N) = (V_N, \Sigma, T, \Delta, v_0)$ of a NCFSM $N = \langle P_1, \ldots, P_n \rangle$. As in traditional data flow analysis (say, as found in [1, Chap. 10]), we will set up a system of data flow equations capturing the set of all possible contents of the buffer when the machine is in a particular state $\bar{v} \in V_N$. More formally, for a given state $\bar{v}$ we want to characterize the set $S_{\bar{v}} = \{z \mid [\overline{v_0}, \varepsilon] \xrightarrow{*} [\bar{v}, z]\}$. It is easy to see that all the safety properties mentioned above can be reformulated using the sets $S_{\bar{v}}$. For example, we can

say that a network N is free of deadlocks iff

$$\forall \bar{v} \in RV(N) \varepsilon \notin S_{\bar{v}}.$$

Obviously, $S_{\bar{v}}$ is not computable. We obtain a conservative approximation that is a superset of $S_{\bar{v}}$. We do so in two stages, first showing how to capture a superset of $S_{\bar{v}}$ for strict FIFO networks and then extending it to *quasi*-FIFO networks.

4.1 Quasi- and Strict FIFO Shuffle-Products

Recall that in Section 3 we indicated that, in general, the single buffer of the shuffle-product need not be a strict FIFO queue. However, there are many networks whose shuffle-product's buffers can still be treated as strict FIFO queues. Let us define the concepts *strict* and *quasi*-FIFO shuffle-products:

Definition 4.1. (Strict and quasi-FIFO shuffle-products). The shuffle-product $SP(N)$ of a NCFSM N is *strict* FIFO if, for every event sequence e that leads the shuffle-product from the initial state to a global state $[\bar{v}, z] \in RS(SP(N))$ and for every event $-g \in -\sum_{i,j}$, the following holds:

$$\left(z = z_1 g z_2 \& h_{j,i}(z_1) = \varepsilon\right) \Rightarrow \left(\exists e' \in \pm \sum{}^{*} \left((e' \simeq e) \& \left([\bar{v}, g z_1 z_2] \in \Delta(e')\right)\right)\right).$$

Otherwise, $SP(N)$ is said to be *quasi*-FIFO.

Put differently, this definition essentially says that the shuffle-product $SP(N)$ of a NCFSM N is strict FIFO if, whenever a global state $[\bar{v}, z_1 g z_2]$ is reachable by an event sequence, $e, g \in \sum_{i,j}$, and z_1 do not contain any messages from the channel $P_i \rightarrow P_j$, then the global state $[\bar{v}, g z_1 z_2]$ is reachable by some event sequence e' such that $e' \simeq e$. The advantage of this characterization is that the single queue used in the shuffle-product can be treated as a FIFO queue.

We extend the terminology by saying that a NCFSM N is a strict (*quasi*-, resp.) FIFO network if its shuffle-product is strict (*quasi*-, resp.) FIFO. We now define an important class of *strict* FIFO networks. A network $N = \langle P_1, \ldots, P_n \rangle$ is *cyclic* if the topology graph $TG(N)$ is a simple cycle. Two-machine networks considered in an earlier version of this paper [19] are cyclic.

THEOREM 4.1. *If $N = \langle P_1, \ldots, P_n \rangle$ is cyclic, then N is a strict FIFO network.*

PROOF. Assume that $N = \langle P_1, \ldots, P_n \rangle$ is cyclic. We show that for any event sequence $e \in \pm \sum^{*}$ and for any $i, j \in I$, if $[\bar{v}, z] \in \Delta(e)$, where $z = z_1 g z_2$, $g \in \sum_{i,j}$, and $h_{j,i}(z_1) = \varepsilon$, there exists another event sequence $e' \simeq e$ such that $[\bar{v}, g z_1 z_2] \in \Delta(e')$.

As $h_{j,i}(z_1) = \varepsilon$ and N is cyclic, we can write z_1 as $g_1 g_2 \cdots g_k$, where $g_l \in \sum_{i_l, j_l}$ and $i_l \neq i$ $(1 \leq l \leq k)$. The event sequence e itself can be written as

$$w_0(-g_1) w_1(-g_2) w_2 \cdots (-g_k) w_k(-g) w_{k+1},$$

where $-g_l (1 \leq l \leq k)$ is the send command that placed the message g_l $(1 \leq l \leq k)$ in the buffer and $w_l \in (\pm \sum)^{*}$, $0 \leq l \leq k + 1$.

We want to show that all the send and receive events from $\pm\sum_{i}$ in w_l $(1 \le l \le k)$ can be pushed to a position before $-g_1$. After such a reordering of e, the event $-g$ can then be pushed to a position before $-g_1$, thus proving our theorem.

Let us first look at w_1. Let $w_1 = w_1' r_1 w_1''$, where w_1' contains only send events and r_1 is the first receive event on w_1. As the message g_1 is still in the buffer at state $[\bar{v}, z]$, we are ensured that w_1 does not contain any send events from $-\sum_{i_1, j_1}$. Therefore, all the send events on w_1' can be moved to a position before the event $-g_1$ without affecting the final global state $[\bar{v}, z]$. If $r_1 \notin +\sum_{i_1}$, it can also be pushed to a position before $-g_1$. Otherwise, let r_1 be left in its current position. The same procedure can be recursively applied to w_1'' until an event sequence $u_1 = u_1'(-g_1)u_1''$ is obtained, where $w_0 u_1 \simeq w_0(-g_1)w_1$, $u_1' \in ((\pm\sum) - (\pm\sum_{i_1}))^*$, and $u_1'' \in (+\sum_{i_1})^*$.

Inductively, the above procedure can be applied to $w_2, \ldots, w_k$. When applied to w_l, the key observation is that w_l does not contain any send events from $-\sum_{i_1} \cup \cdots \cup -\sum_{i_1}$. An event sequence $u_l = u_l'(-g_1)u_1''(-g_2)u_2'' \cdots (-g_l)u_l''$ can be obtained, where $w_0 u_1' u_2' \cdots u_{l-1}' u_l \simeq w_0 (-g_1)w_1 \cdots (-g_l)w_l$, and each of the $u_{k'}'' \in (+\sum_{i1} \cup \cdots \cup +\sum_{i_{k'}})^*$, $1 \le k' \le l$.

Since N is cyclic, $i_l \neq i$ for all $1 \le l \le k$. After u_k is obtained, we can safely push the event $-g$ before g_1. As every event sequence resulting from each of the above steps is equivalent to the sequence before the procedure is applied, we are assured that $e' \simeq e$. □

4.2 Data Flow Equations for Quasi-FIFO Networks

For a language $L \subseteq \sum^*$, the *quasi-derivative* of L with respect to a letter (i.e., a message in this context) $a \in \sum_{i,j} \subset \sum$, notated as $\hat{a}L$, is the language $\{z \mid \exists z' = z_1 a z_2 \in L,\ z = z_1 z_2 \& h_{j,i}(z_1) = \varepsilon\}$. Consider a typical state $\bar{v}$ in the shuffle-product $SP(N)$, as shown in Figure 3. Observe the following:

(1) If $\overline{v_1} \xrightarrow{-m} \bar{v}$, then $S_{\overline{v_1}} m \subseteq S_{\bar{v}}$.

(2) If $\overline{v_1} \xrightarrow{+g} \bar{v}$, then $\hat{g} S_{\overline{v_1}} \subseteq S_{\bar{v}}$.

Therefore, for each node $\bar{v} \in SP(N)$, we create an equation $EQ_{\bar{v}}$, defined as

$$S_{\bar{v}} = S_{\overline{v_1}} m_1 + \cdots + S_{\overline{v_j}} m_j + \hat{g}_1 S_{\overline{v_1'}} + \cdots + \hat{g}_k S_{\overline{v_k'}}.$$

The only exception is that for the state $\overline{v_0}$ the term ε is also added to the right-hand side of the equation. This indicates that the channel is initially empty. For a shuffle-product $SP(N)$, let $EQ(SP(N))$ be the system of data flow equations set up according to above-mentioned rules.

Example 3. For the network N_2 shown in Figure 2, the system of data flow equations $EQ(SP(N_2))$ is

$$S_{[1,5]} = \hat{b} S_{[1,8]} + \hat{d} S_{[4,5]} + \varepsilon; \qquad S_{[1,6]} = S_{[1,5]} c + \hat{d} S_{[4,6]};$$

$$S_{[1,7]} = S_{[1,6]} d + \hat{d} S_{[4,7]} \qquad S_{[1,8]} = \hat{a} S_{[1,7]} + \hat{d} S_{[4,8]};$$

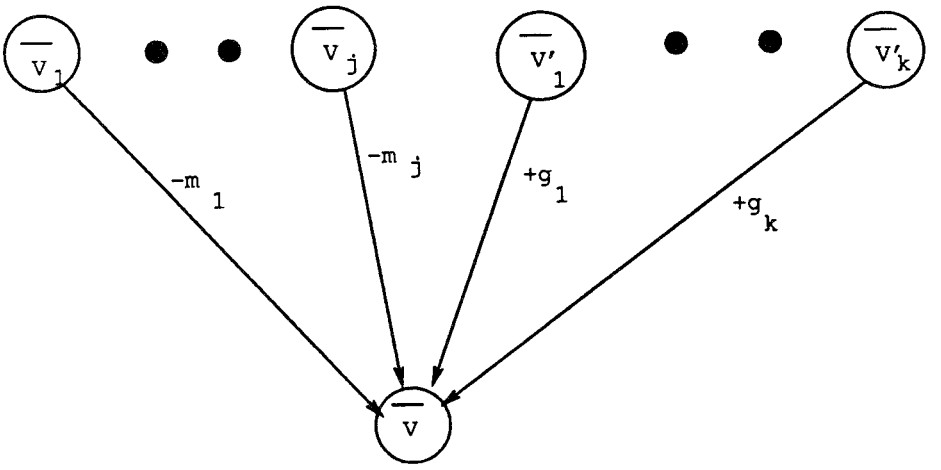


Fig. 3. A typical node in $SP(N)$.

$$S_{[2,5]} = \hat{b}S_{[2,8]} + S_{[1,5]}a; \qquad S_{[2,6]} = S_{[2,5]}c + S_{[1,6]}a;$$
$$S_{[2,7]} = S_{[2,6]}d + S_{[1,7]}a; \qquad S_{[2,8]} = \hat{a}S_{[2,7]} + S_{[1,8]}a;$$
$$S_{[3,5]} = \hat{b}S_{[3,8]} + S_{[2,5]}b; \qquad S_{[3,6]} = S_{[3,5]}c + S_{[2,6]}b;$$
$$S_{[3,7]} = S_{[3,6]}d + S_{[2,7]}b; \qquad S_{[3,8]} = \hat{a}S_{[3,7]} + S_{[2,8]}b;$$
$$S_{[4,5]} = \hat{b}S_{[4,8]} + \hat{c}S_{[3,5]}; \qquad S_{[4,6]} = S_{[4,6]} = S_{[4,5]}c + \hat{c}S_{[3,6]};$$
$$S_{[4,7]} = S_{[4,6]}d + \hat{c}S_{[3,7]}; \qquad S_{[4,8]} = \hat{a}S_{[4,7]} + \hat{c}S_{[3,8]}.$$

We can interpret the data flow equations as follows: Each variable $S_{\bar{v}}$ represents a language. A term of the form $S_{\bar{v}} \cdot m$ represents the concatenation of language $S_{\bar{v}}$ and the letter m. A term of the form $\hat{g}S_{\bar{v}}$ represents the quasi-derivative of language $S_{\bar{v}}$ with respect to the letter g. The symbol $+$ represents the union of languages. Intuitively, the equations capture the fact that the contents of the buffer at a state depend on the contents of the buffer at its predecessor states. Such relations have been collected into the set of equations $EQ(SP(N))$.

It is easy to see that $\cdot$, $+$, and quasi-derivative are all continuous functions over complete lattice $(2^{\Sigma^*}, \subseteq)$. Hence, a unique least fix-point to the data flow equation $EQ(SP(N))$ exists.

4.3 Data Flow Equations for Strict FIFO Networks

The quasi-derivative operation is defined to characterize the receive operation on the single buffer of the shuffle-product. Strict FIFO networks are of special interest, since we can replace the quasi-derivative operation in the data flow equations by the standard derivative operation. For a language $L \subseteq \Sigma^*$, the *derivative* of L with respect to a letter $a \in \Sigma$, notated as $\bar{a}L$, is the language $\{z \mid \exists z' = az \in L\}$ [5].

The question is, can we decide whether a given word belongs to the solution of the data flow equations? Unfortunately, the answer is negative. It is shown in [17] that the class of possible solutions to these equations is equivalent to the class of recursively enumerable languages.

4.4 Usefulness of Approximate Solutions

A superset of the least solution gives sufficient conditions to ensure that something bad (e.g., deadlocks, unspecified receptions) will never happen.

For instance, let $\{L_{fix}(\bar{v}) \mid \bar{v} \in V_N\}$ be the least fix-point to the data flow equations $EQ(SP(N))$. Let $\{L(\bar{v}) \mid \bar{v} \in V_N\}$ be a system of superset solutions to the data flow equations, namely,

$$L_{fix}(\bar{v}) \subseteq L(\bar{v}).$$

If $\varepsilon \notin L(\bar{v})$, then we can reason as follows:

$$\varepsilon \notin L(\bar{v}) \Rightarrow \varepsilon \notin L_{fix}(\bar{v}) \Rightarrow [\bar{v}, \varepsilon] \notin RS(SP(N)) \Rightarrow [\bar{v}, \overline{c_0}] \notin RS(N).$$

If $\bar{v}$ is a receive state tuple and $\varepsilon \notin L(\bar{v})$, then we can conclude that $[\bar{v}, \overline{c_0}]$ is not a reachable state.

A natural question is: can we find a system of supersets that is useful in determining the safety properties of NCFSMs? The term *useful* implies several things. First, the membership problem of the superset solutions should be decidable, as can be seen from the example. Second, it should be reasonably tight, that is, close to the actual solutions. Any approximation technique will fail to work under certain circumstances. However, to have practical value, it should be able to work well for a relatively large number of practical problems. And, last, but certainly not least, it should be computable with reasonable costs (time and space).

In the next section, we propose approximation methods to compute superset solutions to the least fix-point.

5. APPROXIMATE SOLUTIONS FOR STRICT FIFO NETWORKS

Based on the data flow equations set up in the last section, we present our main results in this section. In Section 5.1, we present a general discussion of the heuristics used. In Sections 5.2–5.5, we provide an intuitive explanation of our algorithm. The algorithm itself is presented in Section 5.6. In Sections 5.7 and 5.8, we formally show that our algorithm provides semantically sound information.

5.1 General Heuristics

A commonly used technique for solving equations like $EQ(SP(N))$ is to iterate through the equations using empty sets as the first approximation. As we are dealing with lattices of unbounded height, we are not assured of termination.

Certainly, one needs a bounded lattice to have a terminating flow analysis. From the definition of deadlock and unspecified reception in Section 3, it can be seen that given a state $[\bar{v}, \bar{c}] = [p_1, \ldots, p_n, c_{1,2}, \ldots, c_{n,n-1}]$ it is enough to

know

—whether $c_{i,j}$ is empty; and
—if $c_{i,j}$ is not empty, the first message in that queue, that is, $INIT(c_{i,j})$.

This suggests bounding the length of strings in the languages $L_{\bar{v}}$. But, by doing so, one would be obtaining an analysis based on bounded buffers; a solution that is not acceptable. In Section 3 we saw a correspondence between transition sequences and event sequences. In much the same way, one could talk about a correspondence between transition sequences and sequences of messages in a buffer. The equations $EQ(SP(N))$ could then be framed such that the identity of the edge sending a message is captured instead of the message itself. Clearly, the message type can be easily recovered once we know the identity of an edge. For example, the equation for $S_{[1,7]}$ could now be read as

$$S_{[1,7]} = S_{[1,6]}\left([1,6] \xrightarrow{-d} [1,7]\right) + \hat{d}S_{[4,7]}.$$

Of course, we would have to redefine what the quasi-derivative operator is. But, once we have set up such equations, a solution would consist of a set of sequences of send edges/transitions in the shuffle-product graph/automaton. The advantage of rephrasing these equations is that we can now bound the length of the sequence of transitions and, yet, obtain a solution to the original problem of dealing with unbounded buffers. For instance, by bounding the length of the sequence of edges to one, we would essentially be computing the set of all edges that contribute the first message to some buffer in some string of $L_{\bar{v}}$. What follows is an attempt to come up with a definition of quasi-derivative on sequences of edges in a way that preserves the semantics. Corresponding to the two pieces of information we need for deadlock and unspecified reception analysis, we compute the following:

—a Boolean $empty[\bar{v}]$ satisfying the property

$$[\bar{v}, \varepsilon] \in RS(SP(N)) \Rightarrow empty[\bar{v}] = 1;$$

—a set $first[\bar{v}] \subseteq E_s$ (the set of send edges in the shuffle-product graph) such that

$$\{m \mid -m \in label(first[\bar{v}])\} \supseteq \{INIT(z) \mid [\bar{v}, z] \in RS(SP(N))\}.$$

Furthermore, computation of $first[\bar{v}]$ would allow us to encode the possible contents of the buffer at $\bar{v}$ (i.e., members of $S_{\bar{v}}$) as certain paths between $e \in first[\bar{v}]$ and $\bar{v}$ in the shuffle-product graph. We develop a solution to the data flow analysis problem in two stages: First, we consider strict FIFO networks and make use of the fact that quasi-derivatives can be replaced by derivatives in the flow equations. We then extend the algorithms for the strict FIFO networks to the more general quasi-FIFO networks.

The work presented here is also similar to Jones and Muchnick's work [12] on analysis of LISP-like programs. Much like their work, we would like to do away with the differential terms and arrive at equations with just the

concatenation and union operator. Again, we cannot be completely successful due to the undecidable nature of the problem. What follows is an attempt to do away with the receive edges in a shuffle-product graph, replacing them with appropriate *first* sets and ε-edges (to be defined and explained shortly). In carrying out this replacement, we want to make sure that the original reachable global states are preserved.

5.2 Heuristics for Strict FIFO Networks

Strict FIFO networks possess a very nice property, as expressed in the following lemma:

LEMMA 5.1. *Let* $[\bar{v}, z_1 g z_2] \in SP(N)$ *be a reachable global state, where, for some* $i, j \in I$, $g \in \sum_{i,j}$, *and* z_1 *do not contain any message from* $\sum_{i,j}$. *If* $[\bar{v}, z_1 g z_2]$ *is reachable by a transition sequence ts, then there exists another transition sequence ts′ leading to a global state* $[\bar{v}, g z_1 z_2]$. *Furthermore, label* $(ts) \simeq label(ts')$.

PROOF. Directly from Definition 4.1 of strict FIFO networks and the correspondence between transition sequences and event sequences. □

From Lemma 5.1, whenever $\overline{v'} \xrightarrow{-g} \overline{v''}$ can occur in a transition sequence *ts* that leads to a global state $[\bar{v}, z_1 g z_2]$, where $g \in \sum_{i,j}$, and whenever z_1 does not contain any message from $\sum_{i,j}$, there always exists another transition sequence *ts′* that leads to $[\bar{v}, g z_1 z_2]$. As messages are added to the end of the queue of a shuffle-product, the edge *se*: $\overline{v'} \xrightarrow{-g} \overline{v''}$ that contributes the message g in *ts′* occurs before the edges that contribute messages in $z_1 z_2$. If we analyze the states reached in prefixes of the transition sequence *ts′*, we find that, in all prefixes that include the event *se*, the single buffer will never be empty (the message g will always be in the queue). Furthermore, we would find that the edge *se*: $\overline{v'} \xrightarrow{-g} \overline{v''}$ is the first send edge whose contribution to the buffer (g in this case) has not been removed in the sequence *ts′*. Now, the transition sequence *ts′* leads the system to the node $\bar{v}$ with the effect that the transition *se* contributes the first message in the queue.

Based on this observation, we associate a set *first*$[\bar{v}]$ with each node $\bar{v}$, which is intended to include send edges $\overline{v'} \xrightarrow{-g} \overline{v''}$ with the property that there exists an execution sequence

$$[\overline{v_0}, \varepsilon] \xrightarrow{*} [\overline{v'}, \alpha_1 \alpha_2 \cdots \alpha_n] \xrightarrow{-g} [\overline{v''}, \alpha_1 \alpha_2 \cdots \alpha_n g] \xrightarrow{*} [\bar{v}, g\beta_1 \beta_2 \cdots \beta_m],$$

such that labels of the transitions in the execution sequence *ts*: $[\overline{v''}, \alpha_1 \alpha_2 \cdots \alpha_n g] \xrightarrow{*} [\bar{v}, g\beta_1 \beta_2 \cdots \beta_m]$ are a shuffle of the sequence of receive events $(+\alpha_1)(+\alpha_2), \ldots, (+\alpha_n)$ and the sequence of send events $(-\beta_1)(-\beta_2), \ldots, (-\beta_m)$.

5.3 Information Propagation

We call a path r from $\bar{v}$ to $\overline{v'}$ in $SP(N)$ a *send path* if all the edges in r are send edges, written as $\bar{v} \xrightarrow{s*} \overline{v'}$; and *ε-path* if all the edges are ε edges, written

as $\bar{v} \xrightarrow{\varepsilon^*} \overline{v'}$.

If r contains a combination of send, receive, and ε-edges, then it is written as $\bar{v} \xrightarrow{*} \overline{v'}$.

Consider a path

$$r: \overline{v_0} \xrightarrow{-g_1} \overline{v_1} \xrightarrow{-g_2} \cdots \xrightarrow{-g_k} \overline{v_k}$$

in the shuffle-product graph of Figure 4a, and its corresponding execution sequence

$$r: [\overline{v_0}, \varepsilon] \xrightarrow{-g_1} [\overline{v_1}, g_1] \xrightarrow{-g_2} \cdots \xrightarrow{-g_k} [\overline{v_k}, g_1 g_2 \cdots g_k].$$

At each of the states $\overline{v_1}, \overline{v_2}, \ldots, \overline{v_k}$, the first message in the queue is of type g_1. Since the edge se_1: $\overline{v_0} \xrightarrow{-g_1} \overline{v_1}$ is the cause for this first message, we can infer that se_1 should be in the *first* sets of the nodes $\overline{v_1}, \overline{v_2}, \ldots, \overline{v_k}$. This observation provides us with the initialization step of the algorithm:

$$first[\bar{v}] = \left\{ \overline{v_0} \xrightarrow{-g} \overline{v'} \mid \overline{v'} \xrightarrow{s^*} \bar{v} \right\} \quad \text{for all } \bar{v}.$$

Again consider the execution sequence r of Figure 4a. The execution sequence r can be extended to include the receive edge re_1: $\overline{v_k} \xrightarrow{+g_1} \overline{v_1'}$ to obtain an execution sequence r',

$$r': [\overline{v_0}, \varepsilon] \xrightarrow{-g_1} [\overline{v_1}, g_1] \xrightarrow{-g_2} \cdots \xrightarrow{-g_k} [\overline{v_k}, g_1 g_2 \cdots g_k] \xrightarrow{+g_1} [\overline{v_1'}, g_2 \cdots g_k].$$

By inspecting r' we can observe that (1) the effect of the receive edge re_1 has been to remove the message enqueued by the send edge se_1, and (2) the message g_2 sent as a result of the edge se_2: $\overline{v_1} \xrightarrow{-g_2} \overline{v_2}$ is the first message in the queue at state $[\overline{v_1'}, g_2 \cdots g_k]$; thus, se_2 should be in $first[\overline{v_1'}]$. But how can we infer this without explicit construction of the possible execution sequences? The former can be inferred based on the fact that a receive edge re: $\bar{v} \xrightarrow{+g} \overline{v'}$ *matches* a send edge se: $\overline{v'} \xrightarrow{-g} \overline{v_1'}$ provided se is in $first[\bar{v}]$. The latter was inferred based on the fact that there is an execution sequence e containing send edges se_1 and se_2 and a receive edge re_1, such that (1) se_1 appears before se_2, which in turn appears before re_1 in e; (2) se_1 and se_2 can be matched; and (3) there are no other send events between se_1 and se_2 in e.

Clearly, the problem of finding a send edge se_2 whose effect is exposed when we match a send edge se_1 against a receive edge re_1 satisfying all the conditions given above is not decidable. We, therefore, have to settle for conservative approximations to the set of edges that *could possibly* be exposed when a receive edge is matched against a send edge. Zooming in on a receive edge re_1 and a send edge se_1 that are *matched*, we can see that a send edge se_2 that is exposed as a result of the match would lie on a path, of the shuffle-product graph, starting from the target of the send edge and ending at the source of the receive edge. Furthermore, se_2 should be the first send edge

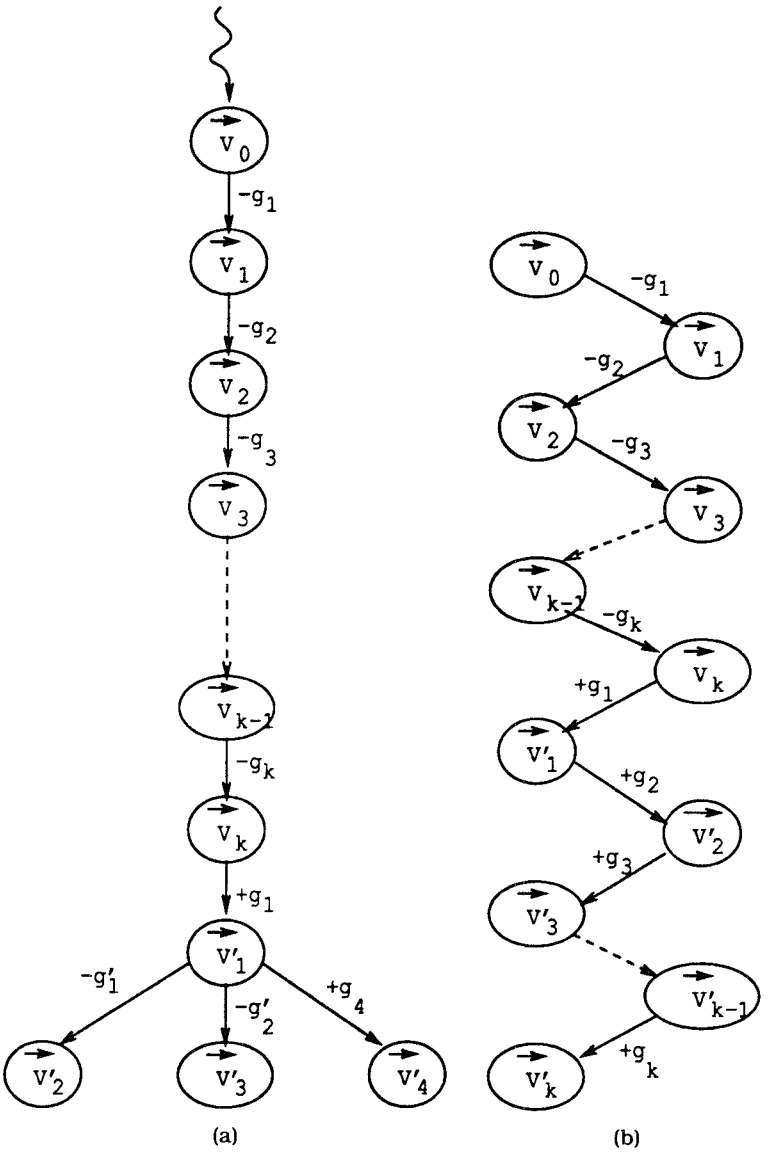


Fig. 4. Intuitive explanation. $\overline{v_0} \xrightarrow{-g_1} \overline{v_1}$ is *matched* with $\overline{v_k} \xrightarrow{+g_1} \overline{v'_1}$.

on this path. Since we do not know the identity of this path, we would have to consider all possible paths between the target node of se_1 and source node of re_1, in the shuffle-product graph, for edges that could be exposed as a result of matching se_1 and re_1. Clearly, any arbitrary path should not be used. A path, of the shuffle-product graph, should be considered if there is an execution sequence that includes it. To characterize such paths of a shuffle-product graph, we use two relations, *reach* and *epsc* (read as ε-closure), among nodes

such that

$$\overline{v'} \in reach[\bar{v}] \text{ if } ts_1: [\overline{v_0}, \varepsilon] \xrightarrow{*} [\bar{v}, x] \text{ and } ts_2: [\bar{v}, x] \xrightarrow{*} [\overline{v'}, y];$$

$$\overline{v'} \in epsc[\bar{v}] \text{ if } ts_1: [\overline{v_0}, \varepsilon] \xrightarrow{*} [\bar{v}, x], \quad ts_2: [\bar{v}, x] \xrightarrow{*} [\overline{v'}, y], \text{ and } ts_2 \text{ contains no send edges.} \tag{1}$$

Thus, the iterative step of our algorithm is,

Let $se_1: \overline{v_1} \xrightarrow{-g} \overline{v_2}$ and $re_1: \overline{v_1'} \xrightarrow{+g} \overline{v_2'}$ match, that is, $se_1 \in first[\overline{v_1'}]$.
a send edge $se: \bar{v} \xrightarrow{-m} \overline{v'}$ can be added to $first[\overline{v_2'}]$ provided $\bar{v} \in epsc[\overline{v_2}]$ and $\overline{v_1'} \in reach[\overline{v'}]$.

Note that there are only a finite number of edges that can be added to *first* sets. Hence, this iterative step is guaranteed to terminate.

5.4 Empty Buffer, Reach Set and *epsc* Set

We associate a Boolean variable $empty[\bar{v}]$ with each node $\bar{v}$, which is intended to record whether the node $\bar{v}$ will ever be reached with the buffer being empty. $empty[\bar{v}]$ is initialized to 0 and will be set to 1 when we can infer that node $\bar{v}$ *could possibly* be reached from $\overline{v_0}$ with an empty buffer.

Consider Figure 4b as an example. Initially, the buffer is empty. At node $\overline{v_i}$, $0 \le i \le k$, the buffer contains $g_1 \cdots g_i$; while at node $\overline{v_i'}$, $1 \le i \le k$, the buffer contains $g_{i+1} \cdots g_k$. After the send edge $\overline{v_{k-1}} \xrightarrow{-g_k} \overline{v_k}$ is matched against the receive edge $\overline{v_{k-1}'} \xrightarrow{+g_k} \overline{v_k'}$, the buffer is empty at node $\overline{v_k'}$. The problem is that we have no information to infer that $\overline{v_k'}$ is reachable with an empty buffer. Furthermore, the send edge $\overline{v_{k-1}} \xrightarrow{-g_k} \overline{v_k}$ and the receive edge $\overline{v_{k-1}'} \xrightarrow{+g_k} \overline{v_k'}$ could be very far apart in the shuffle-product. To solve this problem, we add an ε-edge $\overline{v_k} \xrightarrow{\varepsilon} \overline{v_1'}$ to the shuffle-product when $\overline{v_0} \xrightarrow{-g_1} \overline{v_1}$ is matched against $\overline{v_k} \xrightarrow{+g} \overline{v_1'}$. Similarly, an ε-edge $\overline{v_i'} \xrightarrow{\epsilon} \overline{v_{i+1}'}$ is added when $\overline{v_i} \xrightarrow{-g_{i+1}} \overline{v_{i+1}}$ is matched against $\overline{v_i'} \xrightarrow{+g_{i+1}} \overline{v_{i+1}'}$, $1 \le i \le k-1$. Whenever a send edge $\overline{v_i} \xrightarrow{-g_{i+1}} \overline{v_{i+1}}$ is matched with a receive edge $\overline{v_i'} \xrightarrow{+g_{i+1}} \overline{v_{i+1}'}$, we test if $\overline{v_i'}$ can be reached from $\overline{v_{i+1}}$ purely through ε-edges. If so, we can infer that the buffer could possibly be empty at node $\overline{v_{i+1}'}$, and we can set $empty[\overline{v_{i+1}'}]$ to 1. Note that this is a conservative inference. Even though we might set $empty[\overline{v_{i+1}'}]$ to 1, there might be no actual execution sequence that leads to the global state $[\overline{v_{i+1}'}, \varepsilon]$. On the other hand, if there exists an execution sequence such that $[\bar{v}, \varepsilon]$ is reachable, our algorithm would definitely set $empty[\bar{v}]$ to 1.

The ε-edges added in this process can be used for fixing *reach* and *epsc* sets. It is easy to see that the following definition of *reach* and *epsc* satisfy

the requirements given in condition (1):

$$epsc[\bar{v}] = \{\overline{v'} \mid \text{there exists an } \varepsilon\text{-path from } \bar{v} \text{ to } \overline{v'}\},$$

$$reach[\bar{v}] = \{\overline{v'} \mid \text{there exists a path containing only } \varepsilon\text{- and send edges from } \bar{v} \text{ to } \overline{v'}\}.$$

5.5 Jumping over Nodes

Adding ε-edges after a match will cause some false reports; that is, we may report that some node $\bar{v}$ can be reached with the buffer being empty, although the state $[\bar{v}, \varepsilon]$ might never be reached. This is one of the reasons why the method proposed in this paper is *approximate*. As indicated earlier, ε-edges are needed to propagate information. However, indiscriminately adding ε-edges would cause some obvious false reports, which can be prevented using some simple heuristics.

Consider the case shown in Figure 5a, where $\overline{v_1} \xrightarrow{-g} \overline{v_2} \in first[\overline{v_2}]$. If we add an ε-edge $\overline{v_2} \xrightarrow{\varepsilon} \overline{v_3}$ when the receive edge $\overline{v_2} \xrightarrow{+g} \overline{v_3}$ is checked against the send edge $\overline{v_1} \xrightarrow{-g} \overline{v_2}$, the path $\overline{v_3} \xrightarrow{-g_1} \overline{v_2} \xrightarrow{\varepsilon} \overline{v_3}$ could be used to claim that execution can reach state $\overline{v_3}$ with message g_1 in front of the buffer. Obviously, this is a false report. To avoid such undesirable reports, *jumping over nodes* is used. Whenever a send edge $\overline{v_1} \xrightarrow{-g} \overline{v_2}$ is matched with a receive edge $\overline{v_3} \xrightarrow{+g} \overline{v_4}$, we test if $\overline{v_3}$ can be reached from $\overline{v_2}$ purely through ε-edges, that is, if $\overline{v_3} \in epsc[\overline{v_2}]$. If so, we do not add an ε-edge $\overline{v_3} \xrightarrow{\varepsilon} \overline{v_4}$. Instead, we set $empty[\overline{v_4}]$ to 1 and place the send edge $\overline{v_4} \xrightarrow{-g'} \overline{v_5}$ into $first[\bar{v}]$, where $\bar{v}$ is reachable from $\overline{v_5}$ purely through send edges. This process is depicted in Figure 5b. For instance, when the edge $\overline{v_1} \xrightarrow{-g} \overline{v_2}$ is matched with edge $\overline{v_2} \xrightarrow{+g} \overline{v_3}$ in Figure 5a, $empty[\overline{v_3}]$ is set to 1, and the send edge $\overline{v_3} \xrightarrow{-g_1} \overline{v_2}$ is added to $first[\overline{v_2}]$ and $first[\overline{v_4}]$. Jumping over nodes more accurately reflects the execution semantics of NCFSMs.

5.6 The Algorithm

We can apply the ideas given above recursively to match each send edge se: $\overline{v_1} \xrightarrow{-g} \overline{v_2}$ with a receive edge $\overline{v_3} \xrightarrow{+g} \overline{v_4}$, where $se \in first[\overline{v_3}]$, until none of the *first* sets can be augmented. During the algorithm, the *first* sets of some nodes in $SPG(N)$ will be augmented, and new ε-edges might be added to the shuffle-product graph.

We are now ready to describe our approximation algorithm formally. Algorithm 5.1 takes the shuffle-product graph $SPG(N) = (V_N, E)$ as input. It then initializes a new graph $G' = (V_N, E')$, where E' contains only the send edges in E.

Besides the arrays *first*, *empty*, *reach*, and *epsc*, two additional arrays, *send-reach* and *iepsc*, are employed. The send-reachable set (*send-reach*) for a

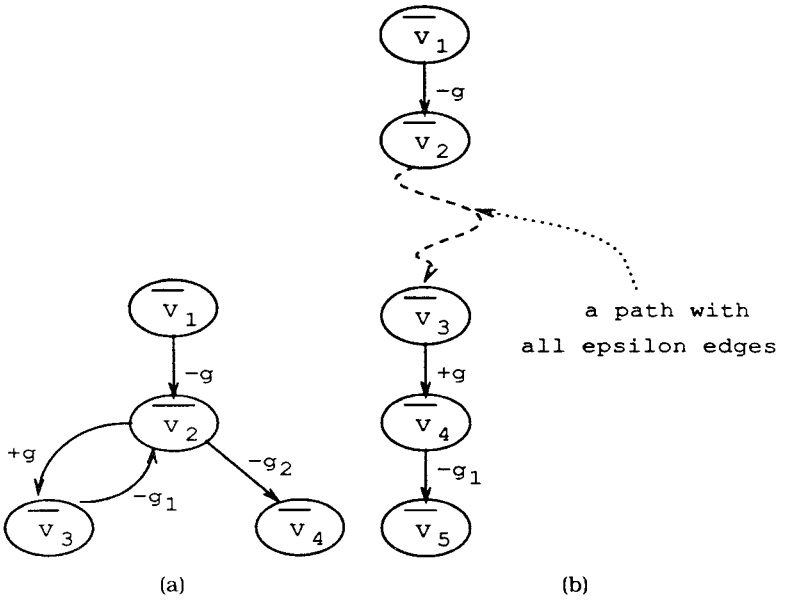


Fig. 5. Jumping over nodes

node $\bar{v} \in V_N$ is defined as

$$send\text{-}reach[\bar{v}] = \left\{\overline{v'} \mid \bar{v} \xrightarrow{s^*} \overline{v'} \in G'\right\}.$$

Notice that *send-reach* is initialized once by Algorithm 5.1 and is not changed afterward. The ε-closure (*epsc*) and inverse ε-closure (*iepsc*) are initialized as

$$epsc[\bar{v}] = \{\bar{v}\},$$
$$iepsc[\bar{v}] = \{\bar{v}\}$$

and are changed whenever necessary to maintain the invariant:

$$epsc[\bar{v}] = \left\{\overline{v'} \mid \bar{v} \xrightarrow{\varepsilon^*} \overline{v'} \in G'\right\},$$

$$iepsc[\bar{v}] = \left\{\overline{v'} \mid \overline{v'} \xrightarrow{\varepsilon^*} \bar{v} \in G'\right\}.$$

The arrays *first* and *reach* are initialized as

$$first[\bar{v}] = \left\{\overline{v_0} \xrightarrow{-g} \overline{v_1} \mid \overline{v_1} \xrightarrow{s^*} \bar{v} \in SP(N)\right\},$$

$$reach[\bar{v}] = \left\{\overline{v'} \mid \bar{v} \xrightarrow{*} \overline{v'} \in G'\right\}.$$

In the algorithm the adjacency lists for $SPG(N)$ and G' and the variables *epsc*, *iepsc*, *reach*, and *send-reach* are globally referenced. The algorithm

consists of a main program and five procedures: *propagation*, *node-jump*, *adjust-eps*, *adjust-reach*, and *insert-Q*.

Step 1 initializes the relevant variables used in the algorithm in terms of their definitions. The output graph G' initially contains only send edges from the shuffle-product graph. Q is intended to hold all of nodes $\bar{v}$, the *first* set of which has been changed. Initially, a node $\bar{v}$ will be put into Q if $first[\bar{v}]$ is not empty.

Step 2 consists of a **while** loop from lines 2.1 to 2.5. At the beginning of each iteration, Line 2.2 picks up a node $\overline{v_1}$ from Q. The **for** loop at line 2.3 picks up a receive edge e: $\overline{v_1} \overset{+g}{\rightarrow} \overline{v_2}$. The **for** loop at line 2.5 then scans each send edge e': $\overline{v_3} \overset{-g}{\rightarrow} \overline{v_4}$ from the set $first[\overline{v_1}]$ and invokes the procedure *propagation*, with the (four) end state tuples of these two edges as parameters.

Procedure *propagation* does three things: (1) It calls procedure *node-jump* if it is necessary to jump over nodes (lines 2–3); (2) it modifies the *first* sets of relevant nodes if necessary (lines 1, and 4–9); and (3) it calls relevant procedures to modify the *epsc*, *iepsc*, and *reach* sets accordingly. It takes four parameters: *rec-start*, *rec-end*, *send-start*, and *send-end*, which are, respectively, the starting and ending nodes of the receive edge *re* and the starting and ending nodes of the send edge *se*. At line 1 it first collects all send edges of the form $\bar{v} \overset{-g'}{\rightarrow} \overline{v'}$, such that there exists an ε-path from *send-end* to $\bar{v}$ and there also exists a path from $\overline{v'}$ to *rec-start*, as illustrated in Figure 6, putting all of these send edges into the variable *send-set*. Line 2 checks if it is necessary to jump over nodes; lines 5–9 propagate the send edges.

Algorithm 5.1 Compute the *first* sets and Boolean array *empty*.
Input: The shuffle-product graph $SPG(N) = (V_N, E)$, where N is strict FIFO.
Output: A graph $G' = (V_N, E')$. Each node $v \in V_N$ has a first edge set $first[\bar{v}]$ and a Boolean variable $empty[\bar{v}]$ that satisfy the following:
(1) If there is a global state of the form $[\bar{v}, z]$ where $h_{J,\iota}(z) = gx$, then a send edge of the form $\overline{v_1} \overset{-g}{\rightarrow} \overline{v_2}$, where $\overline{v_1}|_\iota = p_\iota$ and $\overline{v_2}|_\iota = p'_\iota$, will be in $first[\bar{v}]$.
(2) If the global state $[\bar{v}, \varepsilon]$ is reachable, then $empty[\bar{v}] = 1$.

Step 1. /* initialize relevant variables */
1.1 $E' := \emptyset$;
1.2 **for** each send edge e: $\overline{v_1} \overset{-g}{\rightarrow} \overline{v_2} \in E$ **do**
$E' := E' \cup \{e\}$;
1.3 $Q := \emptyset$;
1.4 **for** each node $\bar{v} \in V_N$ **do**
begin
$empty[\bar{v}] := 0$;
$epsc[\bar{v}] := \{\bar{v}\}$;
$iepsc[\bar{v}] := \{\bar{v}\}$;
$reach[\bar{v}] := \{\overline{v'} \mid \bar{v} \overset{*}{\rightarrow} \overline{v'} \in G'\}$;
$send\text{-}reach[\bar{v}] := \{\overline{v'} \mid \bar{v} \overset{*}{\rightarrow} \overline{v'} \in G'\}$;
$first[\bar{v}] := \{\overline{v_0} \overset{-g}{\rightarrow} \overline{v_1} \mid \bar{v} \in send\text{-}reach[\overline{v_1}]\}$;
if $first[\bar{v}] \neq \emptyset$ **then**
$Q := q \cup \{\bar{v}\}$;
end;

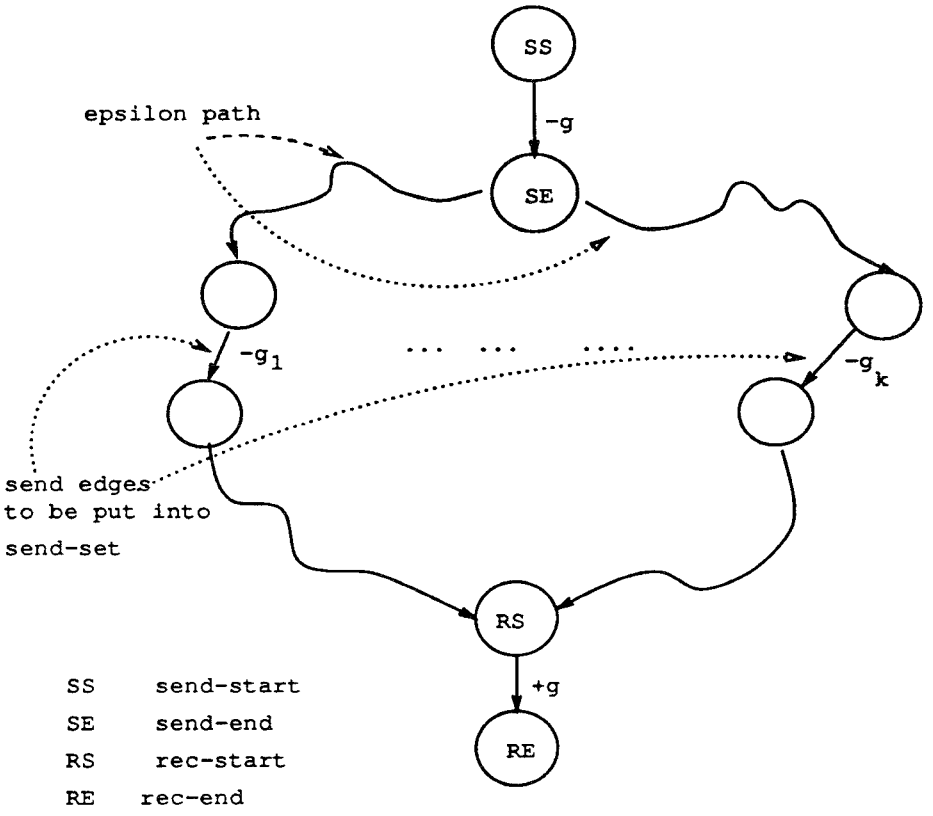


Fig. 6 Send edges to be collected at line 1

```
Step 2.
  2.1  while Q ≠ ∅ do
       begin
  2.2      remove the first node v̄1 from Q;
  2.3      for each receive edge v̄1 -+g-> v̄2 ∈ E do
  2.4        for each send edge v̄3 -−g-> v̄4 ∈ first[v̄1] do
  2.5          propagation(v̄1, v̄2, v̄3, v̄4);
       end
procedure adjust eps(rec-start, rec-end);
  begin
    for each node v̄ ∈ iepsc[rec-start] do
      espc[v̄]:= epsc[v̄] ∪ epsc[rec-end];
    for each node v̄ ∈ epsc[rec-end] do
      iepsc[v̄]:= iepsc[v̄] ∪ iepsc[rec-start];
    E':= E' ∪ {rec-start -ε-> rec-end};
  end /* end of adjust-eps */
procedure propagation(rec-start, rec-end, send-start, send-end);
  begin
    1. Let send-set = {v̄ -−g'-> v̄' | v̄ ∈ epsc[send-end] and rec-start ∈ reach[v̄']};
```

```
2.  if rec-start ∈ epsc[send-end] then /* jump over nodes */
3.     node-jump(rec-end);
4.  if send-set ≠ ∅ then
5.     for each node v1 ∈ send-reach(rec-end)do
6.        for each send edge e' = v̄ -g'-> v̄' ∈ send-set do
7.           if e' ∉ first[v̄1] do
             begin
8.              first[v̄1]:= first[v̄1] ∪ {e'};
9.              insert-Q(v̄1);
             end
10. if (rec-start ∉ eps[send-end]) and (rec-start -ε-> rec-end ∉ E') then
11.    adjust-epsc(rec-start, rec-end);
12.  adjust-reach(send-end, rec-start, rec-end);
end /* end of propagation */

procedure node-jump(rec-end)
  begin
    empty[rec-end]:= 1;
    for each send edge e' = v̄ -g'-> v̄' such that v̄ ∈ epsc[rec-end] do
      for each node v̄1 such that v̄1 ∈ send-reach[v̄'] do
        if v̄ -g'-> v̄' ∉ first[v̄1] then
          begin
            insert-Q(v̄1);
            first[v̄1]:= first[v̄1] ∪ {e'};
          end;
  end; /* end of node-jump */

procedure insert-Q(v̄);
  begin
    if v̄ ∉ Q then Q:= Q ∪ {v̄};
  end /* end of insert-Q */

procedure adjust-reach(send-end, rec-start, rec-end);
  begin
    for each node v̄ ∈ reach(send-end) do
      if rec-start ∈ reach[v̄] do
        reach[v̄]:= reach[v̄] ∪ reach(rec-end);
  end /* end of adjust-reach */
```

Line 5 considers each node $\overline{v_1}$ that is in the *send-reach* set of the node *rec-end*. Lines 6–9 add each send edge e in the *send-set* to *first*$[\overline{v_1}]$ if e is not already in it. Line 9 calls procedure *insert-Q* to insert $\overline{v_1}$ into Q if its *first* set is changed. Line 10 tests if an ε-edge should be added. If so, line 11 calls procedure *adjust-eps* to modify the *epsc* and *iepsc* sets. (An explanation for procedure *adjust-eps* follows.) Finally, it calls procedure *adjust-reach* to modify the *reach* sets. Procedure *node-jump* takes one parameter *rec-end*, which is the end node of the receive edge *re*. It first sets *empty*[*rec-end*] to 1 and then performs the necessary actions for jumping over nodes, as explained in Section 5.5.

Procedure *adjust-eps* takes two parameters, *rec-start* and *rec-end*, which are, respectively, the start and end nodes of the receive edge *re*. It is called only when a new ε-edge from *rec-start* to *rec-end* can be added into G' (lines

10–11 in procedure *propagation*.) Since and ε-edge *rec-start* $\xrightarrow{\varepsilon}$ *rec-end* is added, we need to adjust the *epsc* set for all nodes in *iepsc*[*rec-start*] and the *iepsc* set of all nodes in *epsc*[*rec-end*].

Procedure *adjust-reach* takes three parameters, *send-end*, *rec-start*, and *rec-end*, which are, respectively, the end node of the send edge, and start and end nodes of the receive edge *e*. It is called only whenever a send edge *se* is matched with a receive edge *re*. For each node $\bar{v}$, if

$$[\textit{send-end}, x] \xrightarrow{*} [\bar{v}, y] \xrightarrow{*} [\textit{rec-start}, z]$$

then all nodes in *first*[*reach-end*] are added to *reach*[$\bar{v}$].

5.7 Correctness of Algorithm 5.1

The following theorem shows that Algorithm 5.1 always terminates, and bounds the time and space requirements of the algorithm:

THEOREM 5.1. *Algorithm* 5.1 *terminates. Let* $N = (P_1, \ldots, P_n)$ *be a NCFSM where machine* P_i *has* m_i *states and* $|\sum| = k$. *Let* $t_1 = m_1 m_2 \cdots m_n$ *and* $t_2 = m_1 m_2 + \cdots + m_n$. *The algorithm needs time* $O(t_1^6 t_2^4 k^4)$ *and space* $O(t_1^2 t_2 k)$.

PROOF. Since $|\sum| = k$, a state p_i in P_i can have at most $m_i k$ receive edges. So a state tuple $\bar{v}$ in $SP(N)$ can have at most $t_2 k$ receive transitions. Similarly, a state tuple $\bar{v}$ can have at most $t_2 k$ send transitions. Therefore, the number of possible edges in $SP(N)$ is bounded by $O(t_1 t_2 k)$. From these, it is clear that Step 1 takes $O(t_1^2 t_2 k)$ time.

The size of a *first* set for a node $\bar{v}$ in $SP(N)$ is bounded by $O(t_1 t_2 k)$. Line 1 and lines 5–9 in procedure *propagation* each take time $O(t_1^2 t_2 k)$. An invocation of procedure *node-jump* in procedure *propagation* also takes time $O(t_1^2 t_2 k)$. An invocation of procedure *adjust-eps* or procedure *adjust-reach* takes time $O(t_1^2)$. So a call of procedure propagation at line 2.5 of Step 2 takes time $O(t_1^2 t_2 k)$.

Each iteration of the **while** loop in Step 2 adds at least one edge into the *first* set of some nodes and takes time $O(t_1^2 t_2^2 k^2)$. So the **while** loop in Step 2 iterates at most $t_1^2 t_2 k$ times. Therefore, the total time is bounded by $O(t_2^6 t_2^4 k^4)$.

The adjacency lists for the shuffle-product need space $O(t_1^2 t_2 k)$. The *first* sets also take $O(t_1^2 t_2 k)$ space. The variables *epsc*, *iepsc*, *reach*, and *send-reach* occupy $O(t_1^2)$ space. Altogether, the space requirement is bounded by $O(t_1^2 t_2 k)$. □

We now show the soundness of the algorithm by proving that every reachable state with a nonempty buffer is characterized by appropriate *first* and *reach* sets.

LEMMA 5.2. *Let* $\bar{v}$ *be a node in* V_N. *If there is a global state of the form* $[v, g_1 \ldots g_j]$, *then first*[$\bar{v}$] *contains a send edge of the form* $\overline{v_1'} \xrightarrow{-g_1} \overline{v_1''}$.

Furthermore, there exists a path r: $\overline{v_1'} \xrightarrow{-g_1} \overline{v_1''} \xrightarrow{\varepsilon^*} \overline{v_2'} \xrightarrow{-g_2} \overline{v_2''} \cdots \overline{v_j'} \xrightarrow{-g_j} \overline{v_j''} \xrightarrow{\varepsilon^*} \bar{v}$ *in* $G' = (V, E')$.

PROOF. Assume that $[\bar{v}, g_1 \cdots g_j]$ is a reachable global state in $SP(N)$, that is, that there exists a transition sequence $ts = [\overline{v_0}, \varepsilon] \xrightarrow{a_1} [\overline{v_1}, z_1] \xrightarrow{a_2} [\overline{v_2}, z_2] \xrightarrow{*} [\overline{v_l}, z_l] \xrightarrow{a_l} [\bar{v}, z]$. Let $e = label(ts) = a_1 a_2 \cdots a_l$. We show by induction on the number k of receive events in e that the theorem holds.

Basis: $k = 0$; that is, e does not contain any receive events. Then all events in e are send events. Line 1.3 would put the edge $\overline{v_0} \xrightarrow{a_1} \overline{v_1}$ into $first[\bar{v}]$. Obviously, the send path $\overline{v_1} \xrightarrow{a_2} \overline{v_2} \xrightarrow{*} \overline{v_l} \xrightarrow{a_l} \bar{v}$ is always in G'. By identifying a_i with $-g_i$, we can see that the conclusion trivially holds.

Induction: Assume that for some $k - 1 > 0$, when the global state $[\bar{v}, g_1 \cdots g_j]$ is reached by no more than $k - 1$ receive events, the conclusion holds. Consider a global state $[\bar{v}, z]$ reachable by k receive events. Let $ts = [\overline{v_0}, \varepsilon] \xrightarrow{*} [\overline{v_1}, z_1] \xrightarrow{b_1} [\overline{v_2}, z_2] \xrightarrow{*} \cdots \xrightarrow{*} [\overline{v_{2k-1}}, z_{2k-1}] \xrightarrow{b_k} [\overline{v_{2k}}, z_{2k}] \xrightarrow{*} [\bar{v}, z]$, where b_i $(1 \le i \le k)$ is the ith receive event on ts. The state $[\overline{v_{2k-1}}, z_{2k-1}]$ is reached by $k - 1$ receive events. By the induction hypothesis, there exists a path r: $\overline{v_1'} \xrightarrow{-g_1} \overline{v_1''} \xrightarrow{\varepsilon^*} \overline{v_2'} \xrightarrow{-g_2} \overline{v_2''} \xrightarrow{\varepsilon^*} \cdots \overline{v_j'} \xrightarrow{-g_j} \overline{v_j''} \xrightarrow{\varepsilon^*} \overline{v_{2k-1}}$ in G', such that $\overline{v_1'} \xrightarrow{-g_1} \overline{v_1''} \in first[\overline{V_{2k-1}}]$ and $g_1 \cdots g_j = z_{2k-1}$. Without loss of generality, assume that $b_k = +g \in +\sum_{j',i'}$ for some $i', j' \in I$; namely $P_{i'}$ is trying to receive a message g sent by $P_{j'}$. Note that all of the transitions after the state $[\overline{v_{2k}}, z_{2k}]$ on ts are send transitions. Therefore, the send path r': $\overline{v_{2k}} \xrightarrow{s^*} \bar{v}$, which corresponds to the portion of the sequence ts from $[\overline{v_{2k}}, z_{2k}]$ to $[\bar{v}, z]$, is in G'. We distinguish two cases here:

Case 1. $z_{2k} = \varepsilon$. Then r must be of the form $\overline{v_1'} \xrightarrow{-g} \overline{v_1''} \xrightarrow{\varepsilon^*} \overline{v_{2k-1}}$. Let $[\overline{v_{2k}}, z_{2k}] \xrightarrow{-g'} [\overline{v_{2k+1}}, z_{2k+1}]$ be the send transition after the state $[\overline{v_{2k}}, z_{2k}]$. Then jumping over nodes (line 3 in procedure *propagation*) would put the send edge $\overline{v_{2k}} \xrightarrow{-g'} \overline{v_{2k+1}}$ into $first[\bar{v}]$. Let w be the concatenation of the messages sent on path r'. It is easy to see that $w = z$ and that the theorem is true.

Case 2. $z_{2k} \neq \varepsilon$. Let w be the concatenation of the send messages on path r'. Again, we consider two cases.

If $-g_1 = -g$ (i.e., $g_1 = g$), then when the receive edge $\overline{v_{2k-1}} \xrightarrow{o_k} \overline{v_{2k}}$ is matched with the send edge $\overline{v_1'} \xrightarrow{-g_1} \overline{v_1''}$ at lines 2.3–2.4 the invocation of procedure *propagation* at line 2.5 would put the edge $\overline{v_2'} \xrightarrow{-g_2} \overline{v_2''}$ into $first[\bar{v}]$. Obviously, $g_2 \cdots g_j w = z$.

If $-g_1 \neq -g$, then $-g_1 \notin -\sum_{j',i'}$ (otherwise, either an unspecified reception would occur or e is not executable). Let ts_1 be the portion of ts from

$[\overline{v_0}, \varepsilon]$ to $[\overline{v_{2k-1}}, z_{2k-1}]$. Let $[\overline{v_t'}, z_t'] \xrightarrow{d_t} [\overline{v_t''}, \overline{z_t''}]$ be the send transitions on ts_1, where $dt = -g$ and $d_t = -g_t \notin -\sum_{j',t'}$ $(1 \le i < t)$. As N is a strict FIFO network, by Lemma 5.1, there exists another transition sequence $ts_2 = [\overline{v_0}, \varepsilon] \xrightarrow{*} [\overline{v_{2k-1}}, z_{2k-1}']$, where $z_{2k-1}' = gg_1 \cdots g_{t-1}g_{t+1} \cdots g_j$ (note that $z_{2k-1} = g_1 \cdots g_{t-1}gg_{t+1} \cdots g_j$) and $label(ts_2) \simeq label(ts_1)$. Since ts_2 has $k-1$ receive events, by the induction hypothesis there is a path

$$r'': \overline{v_1^1} \xrightarrow{-g} \overline{v_1^2} \xrightarrow{\varepsilon^*} \overline{v_2^1} \xrightarrow{-g_1} \overline{v_2^2} \cdots \overline{v_t^1} \xrightarrow{-g_{t-1}} \overline{v_t^2} \xrightarrow{\varepsilon^*} \overline{v_{t+2}^1} \xrightarrow{-g_{t+1}} \overline{v_{t+2}^2} \cdots \overline{v_{j+1}^1} \xrightarrow{-g_j} \overline{v_{j+1}^2} \xrightarrow{\varepsilon^*} \overline{v_{2k-1}}$$

in G', where the edge $\overline{v_1^1} \xrightarrow{-g} \overline{v_1^2} \in first[\overline{v_{2k-1}}]$ and $gg_1 \cdots g_{t-1}g_{t+1} \cdots g_j = z_{2k-1}'$. Therefore, when the receive edge $\overline{v_{2k-1}} \xrightarrow{+g} \overline{v_{2k}}$ is checked against the send edge $\overline{v_1^1} \xrightarrow{-g} \overline{v_1^2}$, the edge $\overline{v_2^1} \xrightarrow{-g_1} \overline{v_2^2}$ would be added to $first[\bar{v}]$. This completes the proof. □

We now show that our unspecified reception analysis is sound.

THEOREM 5.2. (*Detecting unspecified receptions in strict FIFO networks*). *Let $\bar{v}$ be a node in V_N. If there is a global state of the form $[\bar{v}, z_1 g z_2]$, where, for some $i, j \in I$, $g \in \sum_{i,j}$, and z_1 do not contain any message from $\sum_{i,j}$, then a send edge of the form $\overline{v'} \xrightarrow{-g} \overline{v''}$ is in $first[\bar{v}]$.*

PROOF. From Lemma 5.1, if $[\bar{v}, z_1 g z_2]$ is a reachable global state, then there exists another reachable global state $[\bar{v}, g z_1 z_2]$. By Lemma 5.2, if $[\bar{v}, g z_1 z_2]$ is a reachable global state, then there exists some send edge $\overline{v'} \xrightarrow{-g} \overline{v''}$ in $first[\bar{v}]$. □

The significance of Theorem 5.2 is as follows: From Proposition 3.1, if $[\bar{v}, z]$ is a reachable global state of the shuffle-product, then $[\bar{v}, c]$ is a reachable global state of the network, where $c_{i,j} = h_{j,i}(z), i, j \in I$. Theorem 5.2 says that if $c_{i,j} = g c_{i,j}'$ then there exists a send edge $\overline{v'} \xrightarrow{-g} \overline{v''}$ in $first[\bar{v}]$. Therefore, to check if N is free of unspecified receptions at a node $\bar{v}$, we merely need to look at the set $first[\bar{v}]$. Our next theorem concerns deadlock detections.

THEOREM 5.3. (*Detecting deadlocks in strict FIFO networks*). *If node $[\bar{v}, \varepsilon]$ is reachable, then $empty[\bar{v}] = 1$ after Algorithm 5.1.*

PROOF. $empty[\bar{v}]$ can be set to 1 only when a receive edge $\overline{v'} \xrightarrow{+g} \bar{v}$ is matched against an edge $\overline{v_1} \xrightarrow{-g} \overline{v_2}$ in $first[\overline{v'}]$ where the node $\overline{v'}$ is in the ε-closure of the node $\overline{v_2}$.

Let $\bar{v} \in V_N$ and $[\bar{v}, \varepsilon]$ be a reachable global state in the shuffle-product; that is, there exists an execution sequence

$$es: [\overline{v_0}, \varepsilon] \xrightarrow{*} [\overline{v'}, z'] \xrightarrow{b} [\bar{v}, \varepsilon].$$

The event b must be a receive event. Assume that $b = +g \in +\sum_{i,j}$ for some $i, j \in I$, and $z' = g$. By Lemma 5.2, there exists a path $r: \overline{v_1} \xrightarrow{-g} \overline{v_2} \xrightarrow{\varepsilon^*} \overline{v'}$ in G' and the edge $e: \overline{v_1} \xrightarrow{-g} \overline{v_2} \in first[\overline{v'}]$. Therefore, when the edge $\overline{v'} \xrightarrow{+g} \bar{v}$ is checked against e at lines 2.3–2.5, a call to procedure *node-jump* at line 3 in procedure *propagation* will set $empty[\bar{v}]$ to 1. □

For each node $\bar{v} \in V_N$, define $head[\bar{v}] = \{g \mid \overline{v_1} \xrightarrow{-g} \overline{v_2} \in first[\bar{v}]\}$. The *restriction* of $head[\bar{v}]$ over machine P_i, notated as $head[\bar{v}] \setminus P_i$, is defined as $head[\bar{v}] \setminus P_i = \{g \mid g \in head[\bar{v}]$ and $\exists j \in I(g \in \sum_{j,i})\}$.

It is clear from Theorem 5.3 that to check for deadlocks we only need to look at the Boolean array *empty*, while from Theorem 5.2, to check for unspecified receptions, we only need the collection of message set *head*. From the definition of unspecified reception, we can get the following two important corollaries:

COROLLARY 5.1. *If* $first[\bar{v}] = \emptyset$ *and* $empty[\bar{v}] = 0$, *after Algorithm* 5.1, *then* $\bar{v}$ *is an unreachable state.*

COROLLARY 5.2. *Let* $N = \langle P_1, \ldots, P_n \rangle$ *be a NCFSM. A receive node* p_i *in machine* P_i $(1 \le i \le n)$ *is free of unspecified receptions if for all* $\bar{v} \in V_N$, *where* p_i *is a component state in* $\bar{v}$, $RMsg(p_i) \supseteq head[\bar{v}] \setminus P_i$.

Theorem 5.3 and Corollary 5.2 suggest the following algorithm for detecting unspecified receptions and deadlocks in a NCFSM $N = \langle P_1, \ldots, P_n \rangle$:

Algorithm 5.2 Detecting deadlocks and unspecified receptions in a NCFSM.

1 Construct the shuffle-product graph $SPG(N) = (V_N, E)$.

2 Apply Algorithm 5.1 to $SPG(N)$. Let $G' = (V_N, E')$ be the output graph.

3 For each receive state tuple $\bar{v} \in V_N$, if $empty[\bar{v}] = 0$, declare that $[\bar{v}, \varepsilon]$ is not a reachable global state (and hence not a deadlock state).

4 For each node $\bar{v} \in V_N$, construct $head[\bar{v}] \setminus P_i$ for all $i \in I$.

5 **for** i:= 1 to n **do**
  for each receive node $p \in P_i$ **do**
    if $RMsg(p) \supseteq head[\bar{v}] \setminus P_i$ for all $\bar{v} \in V_N$ where p is a component in $\bar{v}$ **then**
      Declare p free of unspecified receptions.

6 If each receive node $p \in P_i$ is free of unspecified receptions, then declare that P_i free of unspecified receptions. Otherwise, state that P_i probably has unspecified receptions.

7 If all P_i in the network are free of unspecified receptions, then declare N to be free of unspecified receptions. Otherwise, state that N probably has unspecified receptions.

Example 4. Consider the NCFSM $N_2 = \langle P_2, Q_2 \rangle$ shown in Figure 2. Nodes 3 and 4 in P_2 are receive nodes. $RMsg(3) = \{c\}$; $RMsg(4) = \{d\}$. Nodes 7 and 8 in Q_2 are receive nodes. $RMsg(7) = \{a\}$; $RMsg(8) = \{b\}$. There are 16

send edges in $SP(N_2)$. Let

$$e_1 = [1,5] \xrightarrow{-a} [1,6], \qquad e_2 = [1,5] \xrightarrow{-a} [2,5],$$
$$e_3 = [1,6] \xrightarrow{-b} [1,7], \qquad e_4 = [1,6] \xrightarrow{-a} [2,6],$$
$$e_5 = [2,5] \xrightarrow{-a} [2,6], \qquad e_6 = [2,5] \xrightarrow{-b} [3,5],$$
$$e_7 = [1,7] \xrightarrow{-a} [2,7], \qquad e_8 = [2,6] \xrightarrow{-b} [2,7],$$
$$e_9 = [2,6] \xrightarrow{-b} [3,6], \qquad e_{10} = [3,5] \xrightarrow{-a} [3,6],$$
$$e_{11} = [1,8] \xrightarrow{-a} [2,8], \qquad e_{12} = [2,7] \xrightarrow{-b} [3,7],$$
$$e_{13} = [3,6] \xrightarrow{-b} [3,7], \qquad e_{14} = [4,5] \xrightarrow{-a} [4,6]$$
$$e_{15} = [2,8] \xrightarrow{-b} [3,8], \qquad e_{16} = [4,6] \xrightarrow{-b} [4,7].$$

By running Algorithm 5.1, we can get

$$empty[3,7] = empty[3,8] = empty[4,7] = empty[4,8] = 0.$$

And also,

$$first[3,5] = first[3,6] = first[3,7] = \{e_2, e_{10}, e_7, e_{14}, e_{11}, e_1\},$$
$$first[3,8] = \{e_5, e_6, e_{12}, e_{15}\},$$
$$first[4,5] = first[4,6] = first[4,7] = \{e_{13}, e_{16}, e_3, e_4\},$$
$$first[4,8] = \{e_8, e_9\} \quad \text{and}$$
$$first[1,7] = first[2,7] = \{e_2, e_{10}, e_7, e_{14}, e_{11}, e_1\},$$
$$first[1,8] = first[2,8] = \{e_5, e_6, e_{12}, e_{15}\}.$$

Therefore,

$$head[3,5] = head[3,6] = head[3,7] = \{a, c\},$$
$$head[3,8] = \{b, c\},$$
$$head[4,5] = head[4,6] = head[4,7] = \{a, d\},$$
$$head[4,8] = \{b, d\},$$
$$head[1,7] = head[2,7] = \{a, c\},$$
$$head[1,8] = head[2,8] = \{b, c\}.$$

By Algorithm 5.2, N_2 is free of unspecified receptions and deadlocks. □

Approximate algorithms provide approximate solutions. In our case, Algorithm 5.2 gives a sufficient condition for the deadlock and unspecified reception problems in the following sense: If a network N suffers from deadlocks at node $\bar{v}$, then it will set $empty[\bar{v}]$ to 1. Likewise, if N has unspecified receptions at $\bar{v}$, then $first[\bar{v}]$ will capture a send edge $\overline{v'} \xrightarrow{-g} \overline{v''}$ such that

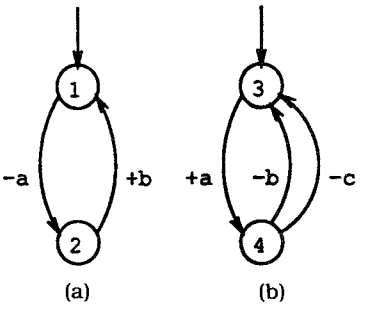


Fig. 7. Network with unspecified receptions. (a) P_3; (b) Q_3.

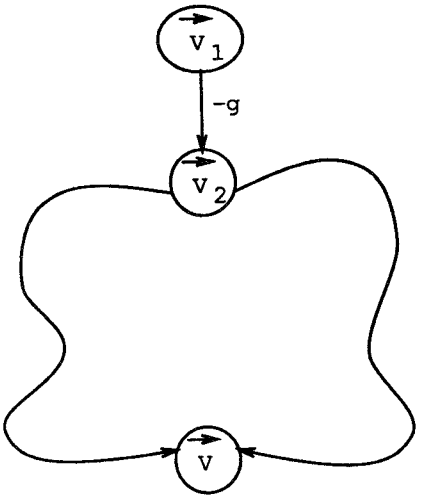


Fig. 8. Finite state automaton defined by a send edge $\overline{v_1} \xrightarrow{-g} \overline{v_2}$ in $first[\bar{v}]$.

$g \in RMsg(\overline{v_i})$, where $\overline{v_i}$ is a local receive state in machine P_i. The following example illustrates this point:

Example 5. The NCFSM $N_3 = \langle P_3, Q_3 \rangle$ shown in Figure 7, suffers from unspecified receptions at node [2, 3]. By running Algorithm 5.1 on this example, we find that the edge $[2,4] \xrightarrow{-c} [2,3] \in first[2,3]$.

5.8 Approximate Solutions to Data Flow Equations

We show in this subsection that the least fix-points to the data flow equations are approximated by regular languages.

Let $G' = (V_N, E')$ be the output graph of Algorithm 5.1, and let $\bar{v}$ be a node in G'. For each edge e: $\overline{v_1} \xrightarrow{-g} \overline{v_2} \in first[\bar{v}]$, define a finite state automaton (with ε-edges) $M_e = (V_N, \Sigma, \delta_e, \overline{v_2}, \{\bar{v}\})$. The transition function δ_e is defined by the edges in G'. Formally, for $\overline{v'} \in V_N$, $\delta_e(\overline{v'}, a) = \{\overline{v''} \mid \text{the edge } \overline{v'} \xrightarrow{a} \overline{v''} \in E'\}$, where $a \in \{\varepsilon\} \cup \Sigma$. The finite state automaton M_e is illustrated in Figure 8. Let $L(M_e)$ be the language accepted by M_e. Let $L_e(\bar{v}) = gL(M_e)$.

Define

$$L(\bar{v}) = \bigcup_{e \in first[\bar{v}]} L_e(\bar{v}).$$

Let $L_{fix}(\bar{v})$ be the actual solution to the data flow equations at node $\bar{v}$; that is, $L_{fix}(\bar{v}) = \{z \mid [\overline{v_0}, \varepsilon] \xrightarrow{*} [\bar{v}, z]\}$.

THEOREM 5.4. (*Inclusion theorem for strict FIFO networks*). *For each node* $\bar{v}$ *in the output graph* $G' = (V_N, E')$ *of Algorithm* 5.1,

$$L_{fix}(\bar{v}) - \{\varepsilon\} \subseteq L(\bar{v}).$$

PROOF. Follows from Lemma 5.2 and the definition of $L(\bar{v})$. □

Theorem 5.4 effectively states that we have approximated the least fix-point by a collection of regular languages $L(\bar{v})$. What is more significant is the following: We never systematically generate any portion of the reachability set in order to solve the deadlock and unspecified reception problems. Furthermore, these regular languages need not be explicitly constructed for the purpose of verification.

6. APPROXIMATE SOLUTIONS FOR QUASI-FIFO NETWORKS

In this section we extend the results of the last section to quasi-FIFO networks. To make the presentation simpler, we use the notation $\forall i, j \in I$ to mean that, for all $i, j \in I$ such that $P_i \rightarrow P_j$ is an edge in the topology graph $TG(N)$.

6.1 Heuristics

The soundness of Algorithm 5.1 is based on Theorem 5.2, which, in turn, is established on the assumption that the network under consideration is strict FIFO. However, the single buffer in a quasi-FIFO network need not be a FIFO queue. The following example should clarify this point:

Example 6. Consider the three-machine network $N_7 = \langle P_1, P_2, P_3 \rangle$, as shown in Figure 9. Only the relevant parts of each of the three machines are given to demonstrate the problem. Machine P_1 sends message a to P_3 and then attempts to receive message b from P_3. Machine P_2 sends c to P_1, then sends message d to P_3, and then attempts to receive f from P_3. P_3 first attempts to receive message d from P_2, then sends message b to P_1, and finally sends f to P_2.

The event sequence e: $(-a)(-c)(-d)$ in the execution path

$$[1,4,8,\varepsilon] \xrightarrow{-a} [2,4,8,a] \xrightarrow{-c} [2,5,8,ac] \xrightarrow{-d} [2,6,8,acd]$$

will lead to global state $[2,6,8,acd]$. However, it should be clear that even

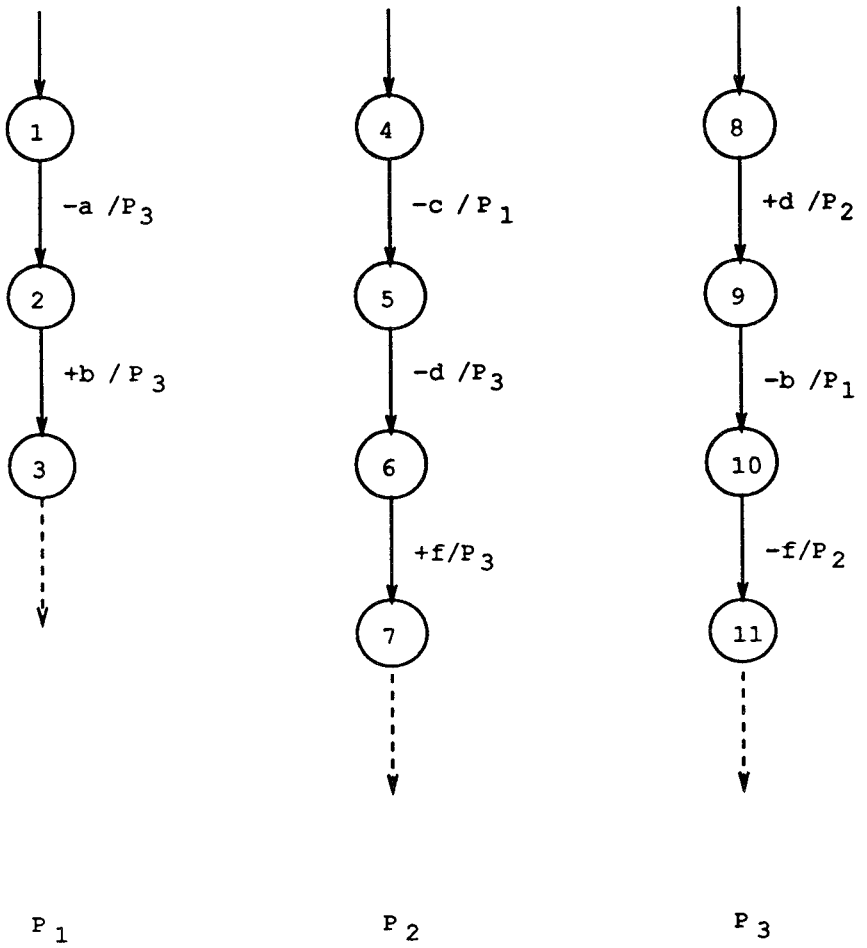


Fig. 9. Quasi-FIFO network $N_7 = \langle P_1, P_2, P_3 \rangle$.

though $a, c \notin \Sigma_{2,3}$, there is no event sequence $e' \simeq e$ that can lead to the global state [2, 6, 8, *dac*].

The problem is caused by the fact, that although P_2 sends c to P_1 before it sends d to P_3, P_3 has to pick up the message d before P_1 can pick up b in order for the whole network to proceed. Although the message d is the first message in the channel $P_2 \rightarrow P_3$, it has to be placed after the message c in the shuffle-product's single buffer.

For quasi-FIFO networks, as the single buffer is no longer a strict FIFO queue, the general heuristics used in the last section has to be modified as follows: *To detect unspecified receptions at state $\bar{v}$, we need to capture all send events $\overline{v'} \stackrel{g}{\rightarrow} \overline{v''}$, where $g \in \Sigma_{i,j}$ for some $i, j \in I$, such that the execution of these send events causes the message g to appear as the first message in the channel $P_i \rightarrow P_j$.*

Based on this observation, for each node $\bar{v} \in V_N$, we associate an array

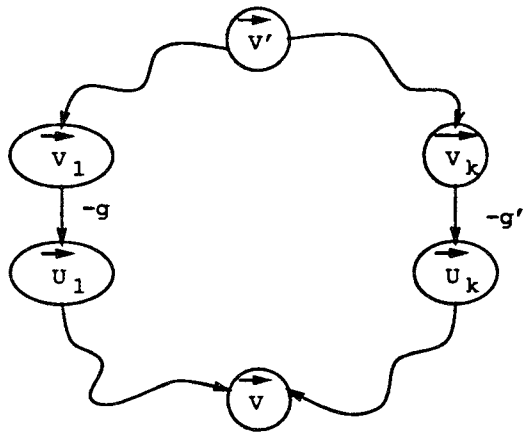


Fig. 10. Illustration of the function *grab-first*$(\bar{v}, i, j, \overline{v'})$.

first$[\bar{v}, i, j]$ $(i, j \in I)$, which is intended to include all send events $\overline{v'} \overset{-g}{\rightarrow} \overline{v''}$ as defined in the above heuristics.

To make the presentation more concise, we define a function *grab-first*$(\bar{v}, i, j, \overline{v'})$ as follows:

$$\textit{grab-first}(\bar{v}, i, j, \overline{v'}) = \left\{ \overline{v_1} \overset{-g}{\rightarrow} \overline{v_2} \colon -g \in -\sum_{i,j} \middle| \exists r \colon \overline{v'} \overset{s^*}{\rightarrow} \overline{v_1} \in SPG(N) \,\&\, zf_{i,j}(r) = \varepsilon \,\&\, \overline{v_2} \overset{s^*}{\rightarrow} \bar{v} \in SPG(N) \right\}.$$

Put differently, *grab-first*$(\overline{v'}, i, j, \bar{v})$ captures all of the first send event on some *send* path from $\overline{v'}$ to $\bar{v}$ that contributes a message to the channel $P_i \rightarrow P_j$. Figure 10 illustrates the idea behind the function *grab-first*$(\bar{v}, i, j, \overline{v'})$. We can initialize *first*$[\bar{v}, i, j]$ by *grab-first*$(\bar{v}, i, j, \overline{v_0})$.

6.2 Information Propagation

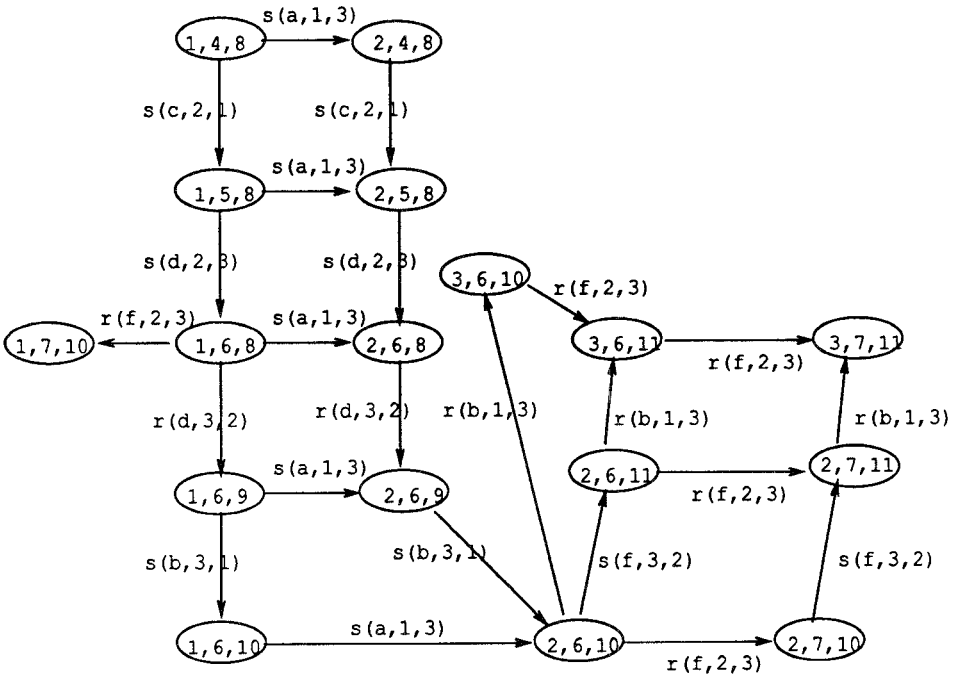
In Algorithm 6.1, to be presented shortly, as in the case of Algorithm 5.1, a graph $G' = (V_N, E')$ is first initialized, where E' contains only the send edges in $SPG(N)$. Algorithm 6.1 then repeatedly matches receive edges of the form $e_1 \colon \bar{v} \overset{+g}{\rightarrow} \overline{v_1}$ with some send edge $e_2 \colon \overline{v_2} \overset{-g}{\rightarrow} \overline{v_3}$ in *first*$[\bar{v}, i, j]$. This process continues until none of the first sets can be augmented.

Information propagation is much more complex for quasi-FIFO networks. Let us use Example 6 for our illustration. Figure 11 shows a portion of the shuffle-product $SP(N_7)$, where $s(g, i, j)(r(g, i, j)$, resp.) denotes the event that machine P_i sends message g to (receives messages g from, resp.) P_j. To simplify the presentation, we use the mnemonic names for nodes and edges, as given in Figure 11.

Initially, we have

$$\textit{first}[\overline{v_2}, 1, 3] = \{e_1\};$$

$$\textit{first}[\overline{v_3}, 2, 1] = \{e_2\};$$



Notation

$\overline{v_1} = [1,4,8]$	$\overline{v_2} = [2,4,8]$	$\overline{v_3} = [1,5,8]$	$\overline{v_4} = [2,5,8]$	$\overline{v_5} = [1,6,8]$
$\overline{v_6} = [2,6,8]$	$\overline{v_7} = [1,6,9]$	$\overline{v_8} = [1,7,10]$	$\overline{v_9} = [2,6,9]$	$\overline{v_{10}} = [1,6,10]$
$\overline{v_{11}} = [2,6,10]$	$\overline{v_{12}} = [3,6,10]$	$\overline{v_{13}} = [2,6,11]$	$\overline{v_{14}} = [2,7,10]$	$\overline{v_{15}} = [3,6,11]$
$\overline{v_{16}} = [2,7,11]$	$\overline{v_{17}} = [3,7,11]$			

$e_1 = \overline{v_1} \xrightarrow{s(a,1,3)} \overline{v_2}$	$e_2 = \overline{v_1} \xrightarrow{s(c,2,1)} \overline{v_3}$	$e_3 = \overline{v_2} \xrightarrow{s(c,2,1)} \overline{v_4}$	$e_4 = \overline{v_3} \xrightarrow{s(a,1,3)} \overline{v_4}$
$e_5 = \overline{v_3} \xrightarrow{s(d,2,3)} \overline{v_5}$	$e_6 = \overline{v_4} \xrightarrow{s(d,2,3)} \overline{v_6}$	$e_7 = \overline{v_5} \xrightarrow{s(a,1,3)} \overline{v_6}$	$e_8 = \overline{v_5} \xrightarrow{r(d,3,2)} \overline{v_7}$
$e_9 = \overline{v_5} \xrightarrow{r(f,2,3)} \overline{v_8}$	$e_{10} = \overline{v_6} \xrightarrow{r(d,3,2)} \overline{v_9}$	$e_{11} = \overline{v_7} \xrightarrow{s(a,1,3)} \overline{v_9}$	$e_{12} = \overline{v_7} \xrightarrow{s(b,3,2)} \overline{v_{10}}$
$e_{13} = \overline{v_9} \xrightarrow{s(b,3,1)} \overline{v_{11}}$	$e_{14} = \overline{v_{10}} \xrightarrow{s(a,1,3)} \overline{v_{11}}$	$e_{15} = \overline{v_{11}} \xrightarrow{r(b,1,3)} \overline{v_{12}}$	$e_{16} = \overline{v_{11}} \xrightarrow{s(f,3,2)} \overline{v_{13}}$
$e_{17} = \overline{v_{11}} \xrightarrow{r(f,2,3)} \overline{v_{14}}$	$e_{18} = \overline{v_{12}} \xrightarrow{s(f,3,2)} \overline{v_{15}}$	$e_{19} = \overline{v_{13}} \xrightarrow{r(b,1,3)} \overline{v_{15}}$	$e_{20} = \overline{v_{13}} \xrightarrow{r(f,2,3)} \overline{v_{16}}$
$e_{21} = \overline{v_{14}} \xrightarrow{s(f,3,2)} \overline{v_{16}}$	$e_{22} = \overline{v_{15}} \xrightarrow{r(f,2,3)} \overline{v_{17}}$	$e_{23} = \overline{v_{16}} \xrightarrow{r(b,1,3)} \overline{v_{17}}$	

Fig. 11. Portion of the shuffle-product of quasi-FIFO networks.

$$first[\overline{v_4}, i, j] = \begin{cases} \{e_1, e_4\} & \text{if } (i,j) = (1,3), \\ \{e_2, e_3\} & \text{if } (i,j) = (2,1), \\ \emptyset & \text{otherwise;} \end{cases}$$

$$first[\overline{v_5}, i, j] = \begin{cases} \{e_2\} & \text{if } (i,j) = (2,1), \\ \{e_5\} & \text{if } (i,j) = (2,3), \\ \emptyset & \text{otherwise;} \end{cases}$$

$$first[\overline{v_6}, i, j] = \begin{cases} \{e_1, e_4, e_7\} & \text{if } (i,j) = (2,3), \\ \{e_2, e_3\} & \text{if } (i,j) = (2,1), \\ \{e_5, e_6\} & \text{if } (i,j) = (1,3), \\ \emptyset & \text{otherwise;} \end{cases}$$

All other $first[\overline{v_k}, i, j] = \emptyset$

At node $\overline{v_5}$, as $first[\overline{v_5}, 2, 3] = \emptyset$, we cannot use the edge e_9 to propagate information to node $\overline{v_8}$. However, $first[\overline{v_5}, 2, 3] = \{e_5\}$. Hence, we can match e_5 against e_8 and propagate information to $\overline{v_7}$. Similarly, $\overline{v_6}$, $first[\overline{v_6}, 3, 2] = \{e_5, e_6\}$. Therefore, we can match e_5 or e_6 against e_{10} and change the information in $first[\overline{v_9}]$. Now the central question is, how should we propagate the information about the first sets?

To envision the problem, assume that the send edge e_5 is used to match the receive edge e_{10}. We should be able to modify the *first* array for state $\overline{v_7}, \overline{v_9}, \overline{v_{11}}, \overline{v_{13}}$. Before the match takes place, for all $i, j \in I$, $first[\overline{v_7}, i] = first[\overline{v_9}, i, j] = first[\overline{v_{11}}, i, j] = first[\overline{v_{13}}, i, j] = \emptyset$. After the match, we should have the following:

$$first[\overline{v_7}, i, j] = \begin{cases} \{e_2\} & \text{if } (i,j) = (2,1), \\ \emptyset & \text{otherwise;} \end{cases}$$

$$first[\overline{v_9}, i, j] = \begin{cases} \{e_1, e_4, e_7\} & \text{if } (i,j) = (1,3), \\ \{e_2, e_3\} & \text{if } (i,j) = (2,1), \\ \emptyset & \text{otherwise;} \end{cases}$$

$$first[\overline{v_{11}}, i, j] = \begin{cases} \{e_1, e_4, e_7\} & \text{if } (i,j) = (1,3), \\ \{e_2, e_3\} & \text{if } (i,j) = (2,1), \\ \{e_{13}\} & \text{if } (i,j) = (3,1), \\ \emptyset & \text{otherwise;} \end{cases}$$

$$first[\overline{v_{13}}, i, j] = first \begin{cases} \{e_1, e_4, e_7\} & \text{if } (i,j) = (1,3), \\ \{e_2, e_3\} & \text{if } (i,j) = (2,1), \\ \{e_{13}\} & \text{if } (i,j) = (3,1), \\ \{e_{16}\} & \text{if } (i,j) = (3,2), \\ \emptyset & \text{otherwise;} \end{cases}$$

There are three main difficulties: The first is that although the edge e_2 contributes the message c to the channel $P_2 \rightarrow P_1$ and occurs before the edge e_5 it is still necessary to place e_2 in $first[\overline{v_9}, 2, 1]$, $first[\overline{v_{11}}, 2, 1]$, and $first[\overline{v_{13}}, 2, 1]$ after the match. The second difficulty is that even though edges like e_7 seem unrelated to the channel $P_2 \rightarrow P_3$ they also have to be propagated. The third difficulty is the necessity to put edges like e_{13} to the *first* sets of relevant nodes after matching e_5 with e_{10}.

Our algorithm for quasi-FIFO networks, Algorithm 6.1, has three procedures (*type-one-propagation*, *type-two propagation*, and *type-three-propagation*)

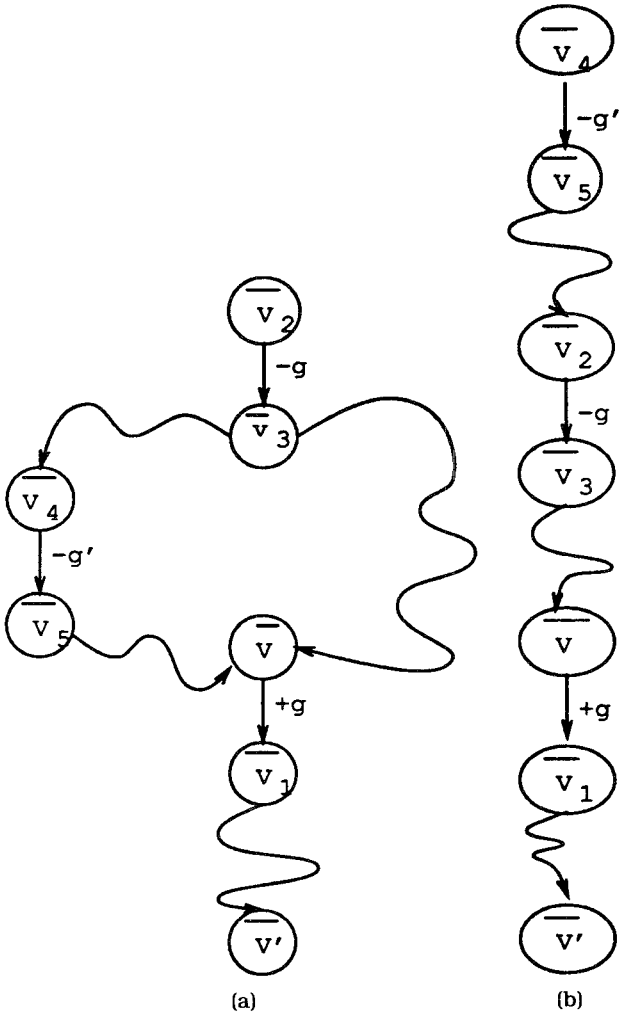


Fig. 12. Propagation of information in quasi-FIFO networks.

to solve each of the three problems. Figure 12 illustrates the main ideas behind procedures *type-one-propagation* and *type-two-propagation*. Procedure *type-one-propagation* will put all edges of the form e: $\overline{v_4} \xrightarrow{-g'} \overline{v_5}$ in Figure 12a, where $g, g' \in \sum_{i,j}$, into *first*$[\overline{v'}, i, j]$. Procedure *type-two-propagation* will put two types of edges into *first*$[\overline{v'}, k, l]$. The first type of edge is of the form e: $\overline{v_4} \xrightarrow{-g'} \overline{v_5}$ in Figure 12a, where $g' \in \sum_{k,l}$, $g \in \sum_{i,j}$ and $(i, j) \neq (k, l)$. The second type of edge is shown in Figure 12b. Let us take the shuffle-product in Figure 11 as an example. After matching e_{10} with e_5, procedure *type-two-propagation* will add the edge e_2 (the second type) to *first*$[\overline{v_9}, 2, 1]$, *first*$[\overline{v_{11}}, 2, 1]$, and *first*$[\overline{v_{13}}, 2, 1]$, and the edge e_7 (the first type) to *first*$[\overline{v_9}, 1, 3]$, *first*$[\overline{v_{11}}, 1, 3]$, and *first*$[\overline{v_{13}}, 1, 3]$. If any *first*$[\bar{v}, k, l]$ is changed during the

invocation of these two procedures, *jump-channel*$[k, l]$ will be set to 0, indicating that for the node $\bar{v}$ the channel $P_k \rightarrow P_l$ should not be propagated by the procedure *type-three-propagation*.

Procedure *type-three-propagation* is designed to propagate edges like e_{13} to *first*$[\overline{v_{11}}, 3, 1]$ and *first*$[\overline{v_{13}}, 3, 1]$. Since the channel $P_3 \rightarrow P_1$ has not been changed either by *type-one-propagation* or by *type-two-propagation*, the edge e_{13} should be in the first sets *first*$[\overline{v_{11}}, 3, 1]$ and *first*$[\overline{v_{13}}, 3, 1]$.

6.3 Detecting Nodes with Empty Buffers

The variable *empty*, as defined in the last section, is still associated with each node $\bar{v} \in V_N$. *empty*$[\bar{v}]$ will be set to 1 if the state $\bar{v}$ can be reached with all channels $P_i \rightarrow P_j$ being empty. As we are dealing with individual channels separately, we introduce another Boolean array *empty-flag*$[\bar{v}, i, j]$. A value of 1 in *empty-flag*$[\bar{v}, i, j]$ denotes that the channel $P_i \rightarrow P_j$ is empty when the global state tuple is $\bar{v}$. Hence, *empty*$[\bar{v}]$ will be set to 1 if at any time during the verification *empty-flag*$[\bar{v}, i, j]$ is found to be 1 for all $i, j \in I$.

When a send edge $\overline{v_1} \xrightarrow{-g} \overline{v_2}$ is matched against a receive edge $\overline{v_3} \xrightarrow{+g} \overline{v_4}$, where $g \in \sum_{i,j}$, an ε-edge $\overline{v_3} \xrightarrow{\varepsilon} \overline{v_4}$ is added. If $\overline{v_3}$ is reachable from $\overline{v_2}$ through a sequence of edges of the form $\overline{v'} \xrightarrow{a} \overline{v''}$, where either $a = -g \notin \sum_{i,j}$ or $a = \varepsilon$, we set *empty-flag*$[\bar{v}, i, j]$ to 1. Let us say such a path from $\overline{v_2}$ to $\overline{v_3}$ *excludes* $P_i \rightarrow P_j$. This can be justified by observing that a send edge $\overline{v'} \xrightarrow{-g'} \overline{v''}$ on any path from $\overline{v_2}$ to $\overline{v_3}$, where $g' \notin \sum_{i,j}$, will not contribute any messages to the channel $P_i \rightarrow P_j$. We then check if *empty-flag*$[\overline{v_4}, k, l] = 1$ for all $k, l \in I$. If so, we set *empty*$[\overline{v_4}]$ to 1.

6.4 The Algorithm

This algorithm is a counterpart of Algorithm 5.1 for strict FIFO networks. The arrays *reach*, *send-reach*, *epsc*, and *iepsc*, as defined in the last section, are also used in Algorithm 6.1. We only give a sketch here. For a detailed description and its correctness proof, readers are referred to [17]. The algorithm consists of two steps and four procedures. Step 1 initializes relevant variables according to their definitions. In particular, if *first*$[\bar{v}, i, j] \neq \emptyset$ after initialization, then *empty-flag*$[\bar{v}, i, j]$ is set to 1. Step 2 consists of a single **while** loop. It loops until the set *first*$[\bar{v}, i, j]$ cannot be augmented for any $\bar{v}, i, j$. Once a receive edge e_1: $\bar{v} \xrightarrow{+g} \overline{v_1}$ is picked up to match a send edge e_2: $\overline{v_2} \xrightarrow{-g} \overline{v_3}$, at line 2.3, where $-g \in \sum_{i,j}$, line 2.4 will test if $\overline{v_3}$ can reach $\bar{v}$ through ε-edges or send edges $\overline{v'} \xrightarrow{-g'} \overline{v''}$ where $g' \notin \sum_{i,j}$. If so, line 2.5 will set *empty-flag*$[\bar{v}, i, j]$ to 1. The Boolean array *jump-channel* is used to record whether the *first* sets for a channel should be propagated with the procedure *type-three-propagation*. Line 2.6 sets *jump-channel*$[i, j]$ to 1, presuming that all channels need to be propagated in that way. Lines 2.7 and 2.8 will set *empty*$[\bar{v}]$ to 1 if *empty-flag*$[\bar{v}, k, l] = 1$ for all $k, l \in I$, namely, if all channels are empty at state $\bar{v}$. Lines 2.9–2.12 sequentially calls the four procedures *type-one-propagation*, *type-two-propagation*, *type-three-propagation*, and *modify-eps-reach*.

Procedure *modify-eps-reach* is simple. It merely modifies the relevant *reach*, *epsc*, and *iepsc* sets of certain nodes to reflect the new reachability relation after matching the edge e_1 with e_2. Procedure *exclusive-path* is also simple. It checks if there exists a path from the end node of the send edge to the start node of the receive edge that excludes $P_i \rightarrow P_j$.

Algorithm 6.1: Compute the *first* sets and Boolean array *empty*.
Input: The shuffle-product graph $SPG(N) = (V_N, E)$, where N is quasi-FIFO.
Output: A graph $G' = (V_N, E')$. Each node $\bar{v} \in V_N$ has an array of first edge set $first[\bar{v}, i, j]$ and Boolean variable $empty[\bar{v}]$ that satisfy the following:
(1) If there is a global state of the form $[\bar{v}, c]$, where $h_{j,\iota}(z) = c_{j,\iota} = gc'_{j,\iota}$, then a send edge of the form $\overline{v_1} \overset{-g}{\rightarrow} \overline{v_2}$ will be in $first[\bar{v}, i, j]$.
(2) If the global state $[\bar{v}, \varepsilon]$ is reachable, then $empty[\bar{v}] = 1$.

Step 1. /* initialize relevant variables */

Step 2.
2.1 **while** $Q \neq \emptyset$ **do**
begin
2.2 remove the first element $\langle \bar{v}, \mathrm{i}, \mathrm{j} \rangle$ from Q;
2.3 **for** each edge e_1: $\bar{v} \overset{+g}{\rightarrow} \overline{v_1}$ and each edge e_2: $\overline{v_2} \overset{-g}{\rightarrow} \overline{v_3} \in first[\bar{v}, i, j]$ **do**
begin
2.4 **if** *exclusive-path*$(\mathrm{i}, \mathrm{j}, \overline{v_3}, \bar{\mathrm{v}})$ **then**
2.5 *empty-flag*$[\bar{v}, i, j] := 1$;
2.6 *jump-channel*$[k, l] := 1$ for all k, $l \in I$;
2.7 **if** *empty-flag*$[\bar{v}, k, l] = 1$ for all k, $l \in I$ **then**
2.8 *empty*$[\bar{v}] := 1$;
2.9 *type-one-propagation*(e_1, e_2, i, j);
2.10 *type-two-propagation*(e_1, e_2, i, j);
2.11 *type-three-propagation*$(\overline{v_1})$;
2.12 *modify-eps-reach*$(\overline{v_3}, \bar{v}, \overline{v_1})$
end
end;

7. EXAMPLES

Algorithm 5.2 provides a practical and strong tool to verify the progress properties of NCFSMs. Note that a protocol is generally designed to limit the set of cross-product states that the component machines are in simultaneously. Furthermore, protocols are designed to simplify interaction between the component processes and not to complicate the interaction. Holzmann reports that, in a typical protocol, of all the shuffle-product nodes less than 10 percent of them are reachable [10].

We present two examples in this section. The first one is a specification of X.25 call/establish clear protocol, and the other is the alternating bit protocol.

Example 7. Figure 13 shows a NCFSM $N_4 = \langle P_4, Q_4 \rangle$ that models the call establishment/clear procedure in X.25 [3, 8]. It has been reported in the

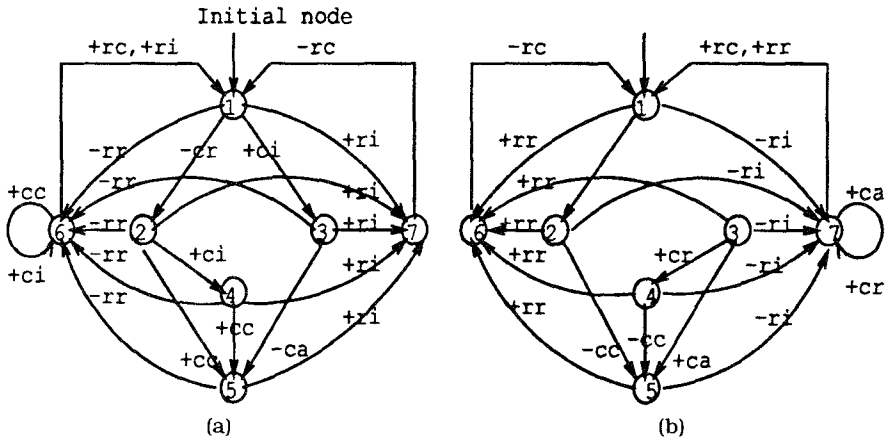


Fig. 13. Call establishment/clear protocol in X.25. (a) P_4; (b) Q_4.

literature that this specification and protocol have been hard to verify, as almost all of the shuffle-product states are reachable. Algorithm 5.2 successfully states that N_4 is free of both unspecified receptions and deadlocks.

Example 8. Figure 14 shows a NCFSM $N_5 = \langle P_5, Q_5, M_1, M_2 \rangle$ that models an alternating bit protocol [2, 16]. This was the first protocol designed and analyzed in the context of the computer networks. Machines M_1 and M_2 model the unreliable communication media through which machines P_5 and Q_5 communicate with each other.

This is a network of four machines. In particular, infinite capacity is assumed for all the four channels $P_5 \rightarrow M_1$, $M_1 \rightarrow Q_5$, $Q_5 \rightarrow M_2$, and $M_2 \rightarrow P_5$. Therefore, due to the infinite size of the set $RS(N_5)$ the conventional reachability analysis method is not applicable here. The shuffle-product $SP(N_5)$ has 1296 state tuples. Our implementation took about four seconds to verify that it has no deadlock or unspecified reception. Among the possible 1296 state tuples, our implementation declared 1152 (i.e., 80 percent) state tuples to be unreachable.

Example 9. For the network N_1 shown in Figure 1, Algorithm 5.2 asserts that it is free of both deadlocks and unspecified receptions. Again, N_1 has an infinite reachability graph; hence, conventional state exploration methods cannot be applied.

8. ABSTRACTING PARALLEL PROCESSES

The NCFSM model described here has both internal nondeterminism, as

—send events are nonblocking and
—two or more edges out of the same node can be labeled by the same event,

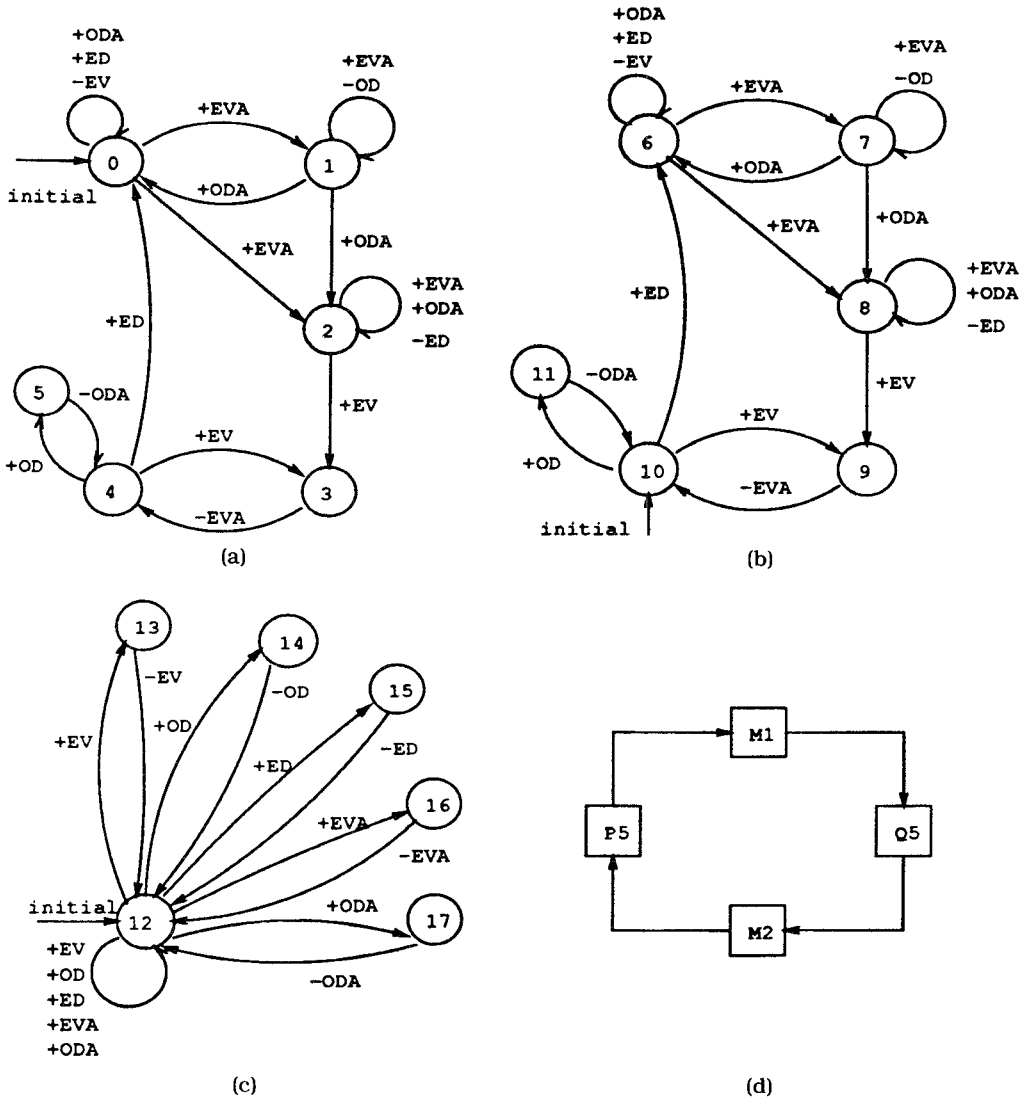
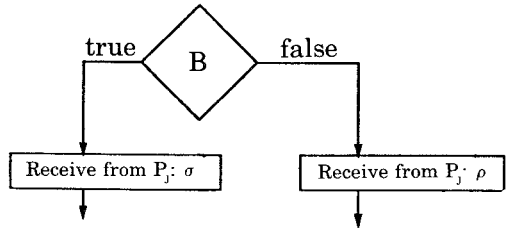


Fig. 14. NCFSM $N_4 = \langle P_5, Q_5, M_1, M_2 \rangle$. (a) P_5; (b) Q_5; (c) M_1 and M_2; (d) network topology graph.

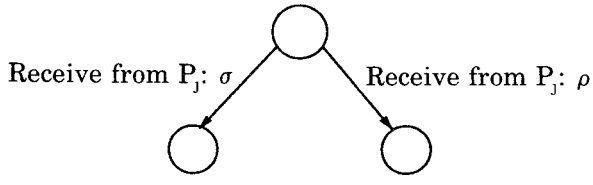
and external nondeterminism. But this does not seem to be enough to handle the following situation.

Consider two processes described in a language for parallelism that has asynchronous communication, as in NCFSMs. In order to analyze these processes statically, one would have to abstract out assignment statements and conditionals from them. But such an abstraction yields processes that cannot be captured as NCFSMs. For example, consider a process P_t that has

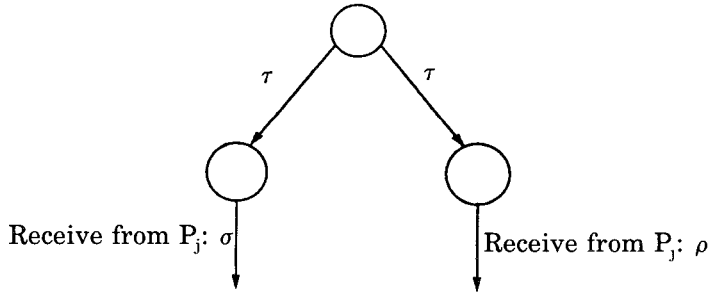
the following piece of code:



Since we cannot know at compile time which of the branches out of the conditional would be chosen, we would have to assume conservatively that both of them are equally likely. This assumption could be used to come up with the following abstraction P_i':



Such an abstraction is not quite correct. There could be an execution sequence in which B is true, but the message in front of the queue $P_j \rightarrow P_i$ is of type ρ, thereby leading to unspecified reception. Thus, the abstraction given is not quite right. The way out of this problem is to add to CFSMs a notion similar to silent (τ) transitions of CCS [15]. The semantics of a silent transition can be defined such that process P_i' can change states without changing any buffer contents. The abstraction P_i' of P_i, should therefore be as follows:



Such an enrichment to the definition of NCFSMs can be easily handled in Algorithm 5.1. It involves enforcing the following rule:

$$\text{If } \bar{v} \xrightarrow{\tau} \bar{v'}, \text{ then } first[\bar{v}] \subseteq first[\bar{v'}].$$

It is easy to see that such an extension is semantically sound. In very much the same way, the notion of zero-testing, which allows transitions to test if a buffer is empty, is added to NCFSMs. The details of experiments dealing with such features in a protocol analyzer is forthcoming [11].

9. DISCUSSION

We have proposed a new approximation method to detect unspecified receptions and deadlocks in NCFSMs. Even though we have concentrated on the nonprogress problems, we believe that the techniques proposed here can also be used for the reachability and unboundedness problems.

We have implemented Algorithms 5.1 and 5.2 using C on our departmental VAX-780 and have successfully verified more than a dozen protocols. This encouraging outcome convinces us that the approximation/data flow approach is a very lucrative way to cope with undecidable problems like the general progress problem of CFSMs.

We point out here that the $O(t_1^6 t_2^4 k^4)$ time complexity in Theorem 5.1 is very unlikely in practice. This bound has been obtained under the assumptions that (1) each iteration of the **while** loop in Step 2 can add only one sending edge into the *first* set of some node in $SPG(N)$; and (2) the CFSMs are *fully dense*, in the sense that each node in each machine can send and receive messages of every possible type to every node. As far as we know, condition (2) is not true for most practical protocols. Our experience shows that condition (1) does not hold either. For instance, for the X.25 call establishment/clear protocol $k = 7$, $t_1 = 49$, $t_2 = 14$, and our implementation take only four iterations (not $2^4 7^{20}$!) of the **while** loop and less than four seconds of CPU time to verify this fairly large protocol. We expect that the time constraint will not hamper the application of our algorithm for most practical protocols.

The approach taken in this paper has been termed *abstract interpretation* elsewhere [6, 7]. The concept of *widening operator*, as used in [6], does not seem to carry over easily to the context on hand. Nevertheless, it would be interesting to establish the relation between our work and those of Clarke [6] and Cousot [7].

The analysis described here captures the edge that contributes the *first* message in a buffer. This can be generalized to capture the first k-edges, much like the definition of $LR(k)$ grammars, to obtain a family of flow-analysis algorithms. By considering larger ks, one would, of course, obtain a sharper analysis. But, certainly, increasing k would increase the cost of the analysis.

One of the impediments to analysis of concurrent systems has been the explosion in the number of states to be analyzed. Our work also suffers from this explosion. But it is no worse than those available for dealing with NCFSMs with bounded buffers. The problem of dealing with such combinatorial explosion is a much addressed important open problem [22]. The class of networks for which our algorithm is complete is also open.

REFERENCES

1. AHO, A., SETHI, R., AND ULLMAN, J. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, Mass., 1986.
2. BARLETT, K., SCANTELBURY, R., AND WILKINSON, P. A note on reliable full-duplex transmission over half-duplex links. *Commun. ACM 12*, 5 (Dec. 1968), 260–261.

3 BOCHMANN, G. V. Finite state descriptions of communication protocols. *Computer Networks 3*, 5 (1978), 361–371.
4. BRAND, D , AND ZAFIROPULO, P On communicating finite-state machines *J. ACM 30*, 2 (1983), 323–342.
5. BROZOZOWSKI, J. A. Derivatives of regular expressions. *J ACM 11*, 4 (1964), 481–494.
6. CLARKE, E Synthesis of resource invariants for concurrent programs In *VI ACM Symposium on Principles of Programming Languages* ACM, New York, 1979, pp. 211–221.
7. COUSOT, P. *Semantic Foundations of Program Analysis*. Prentice-Hall, Englewood Cliffs, N.J., 1981, Chap 10, pp. 303–342.
8 GOUDA, M. Closed covers: To verify progress for communicating finite state machines. *IEEE Trans. Softw. Eng. SE-10*, 6 (Nov. 1985), 846–855.
9. GOUDA, M , MANNING, E., AND YU, Y. T. On the progress of communication between two finite state machines *Inf. Control 63*, 3 (1984), 308–320
10. HOLZMANN, G. *Protocols for Communication Networks*. Addison-Wesley, Reading, Mass., 1990.
11. HOLZMANN, G., AND PURUSHOTHAMAN, S. Experiments in analyzing protocols with unbounded buffers In preparation.
12. JONES, N. D., AND MUCHNICK, S. S. Flow analysis of LISP-like structures. In *Proceedings VI ACM Symposium on Principles of Programming Languages*. ACM, New York, 1979, pp. 244–256
13. KANELLAKIS, P., AND SMOLKA, S. On the analysis of cooperation and antagonism in networks of communicating processes. In *Proceedings IV ACM Symposium on Principles of Distributed Computing*. ACM, New York, 1985.
14 KUNG, H. T. On deadlock avoidance in systolic communication. In *Proceedings of the International Conference on Computer Architecture* (May 1988) IEEE Press, pp. 252–260.
15. MILNER, R. *Communication and Concurrency*. Prentice-Hall, Englewood Cliffs, N.J , 1989.
16. PACHL, J. Protocol description and analysis based on a state transition model with channel expressions. In *Protocol Specification, Testing, and Verification, VII*, H. Rubin and C. H. West, Eds. North Holland, Amsterdam, May 1987, pp. 207–219
17. PENG, W. Analysis of communicating finite state machines. Ph.D. dissertation, Dept. of Computer Science, The Pennsylvania State Univ., University Park, Pa., Dec 1990.
18. PENG, W., AND PURUSHOTHAMAN, S. Analysis of communicating finite state machines for non-progress. In *Proceedings of the 9th International Conference on Distributed Computing Systems*. (June 1990), 280–287.
19. PENG, W., AND PURUSHOTHAMAN, S. Towards data flow analysis of communicating finite state machines. In *Proceedings of the 8th ACM Symposium on Principles of Distributed Computing*. (Aug. 1989)
20 REIF, J., AND SMOLKA, S The complexity of reachability in distributed communicating processes. *Acta Inf. 25*, 3 (1988), 333–354
21. REIF, J., AND SMOLKA, S. Data flow analysis of distributed communicating processes. To appear in *Int J. Parallel Program*
22. SIFAKIS, J , ED. Automatic verification methods of finite state systems. In *Lecture Notes in Computer Science*, 407, 1989.
23. TAYLOR, R. A general-purpose algorithm for analyzing concurrent systems. *Commun. ACM* 26, 5 (1983), 362–376.
24. YU, Y. T., AND GOUDA, M. Deadlock detection for a class of communicating finite state machines. *IEEE Trans. Commun* (Dec. 1982), 2514–2518.

Received March 1989; revised October 1989 and November 1990; accepted January 1991