Acta Informatica 29, 499–522 (1992)

Acta Informatica



# Analysis of a class of communicating finite state machines

**Wuxu Peng**[1] **and S. Purushothaman**[2]*

[1] Department of Computer Science, Southwest Texas State University, San Marcos, TX 78666, USA
[2] Department of Computer Science, The Pennsylvania State University, University Park, PA 16802, USA

Received January 19, 1989 / May 14, 1992

**Abstract.** The *reachability*, *deadlok detection* and *unboundedness detection* problems are considered for the class of *cyclic one-type message* networks of communicating finite state machines. We show that all the three problems are effectively solvable by (a) constructing canonical execution event sequences which belong to a context-free language, and (b) showing that the reachability sets are semilinear. Our algorithms have polynomial complexity in terms of size of a global structure of a network, called the *shuffle-product*. The relationships between general Petri nets and the class of communicating finite state machines considered here are also explored.

## 1 Introduction

A network of communicating finite state machines (NCFSM) consists of a set of finite state machines which communicate asynchronously with each other over (potentially) unbounded FIFO channels by sending and receiving typed messages. As a concurrency model, NCFSM has been widely used to specify and validate communications protocols [1, 13, 10].

In validating systems specified by NCFSMs, we are interested in a number of safety properties such as *freedom from deadlocks*, *freedom from unspecified receptions*, and *freedom from unboundedness*. These safety problems are of interest in checking that the specification of a network is free of logical errors. It is known that the above mentioned safety problems and the reachability problem are undecidable for general NCFSMs [1]. Therefore considerable effort has been devoted to finding classes of networks with decidable safety properties [13, 9].

In this paper we restrict our attention to a class of networks called *cyclic one-type message* (COTM) networks. Informally a NCFSM is a *one-type message* (OTM) network, if all messages communicated in one direction between a pair of processes are all of the same type. A NCFSM is a *cyclic* network if the

* Supported in part by NSF CCR-9004121

topology of the network is a simple cycle. In OTM networks, as all messages communicated over a channel are all of the same type, unspecified reception errors can not occur. We will, therefore, restrict our attention to the *deadlock detection problem* (DDP), the *unboundedness detection problem* (UBDP) and the *reachability problem* (RP).

We are interested in the class of COTM networks for several reasons. First, the ring-structured network topology is widely used in local area networks. Secondly, by collapsing the message types in a general network $N$, we can obtain an OTM network $N'$. As indicated in [13], the absence of deadlocks in $N'$ provides a sufficent condition for asserting the absence of deadlocks in the original network $N$.

Here is a brief historical review. In [2], a procedure, similar to the coverability graph construction in Petri Nets, was provided for checking deadlocks in OTM networks. It was later shown by Yu and Gouda in [13] that deadlocks could be detected in a much simpler way for two-machine OTM networks which do not have any *mixed states*. Their algorithm took time $O(m_1^3 m_2^3)$, where $m_1$ and $m_2$ are the number of states in the two machines. Yu and Gouda [14] proposed an algorithm that detects unboundedness in two-machine OTM networks. However, the complexity of that algorithm is not clear. In [3], it was shown that the deadlock detection problem for OTM networks is PSPACE-complete.

In this paper we first extend the work in [13] by presenting an algorithm that detects deadlocks in two-machine OTM networks that may contain mixed states. Our algorithm takes time $O(m_1^4 m_2^4)$. Using the same framework, we then show how to solve RP, DDP, and UBDP for the class of COTM networks. The complexity of our algorithm is polynomial in terms of the size of a global structure, called the shuffle-product, of the network.

The class of OTM networks are equivalent to general Petri nets. The class of COTM networks, which is a subclass of the OTM networks, is therefore a subclass of Petri nets too. The development of all the three analyses hinges on the fact that the *reachability set* is a *semilinear set* for COTM networks. We show this by constructing canonical execution sequences that are members of a context-free language. Other classes of petri nets that are known to have semilinear reachability sets, such as persistent nets [7] and symmetric petri nets [5], are not comparable to COTM. Furthermore, unlike [7, 6], the proof technique used here does not make use of the solution to the coverability problem for Petri nets.

The remainder of this paper is organized as follows. In Sect. 2 we provide necessary notations and definitions. Sections 3, 4 and 5 form the central part of the paper. In Sect. 3, an algorithm for detecting deadlocks in two-machine OTM networks is presented. Section 4 extends this algorithm to provide an algorithm, named CANCEL, that computes canonical execution sequences. With that algorithm, in Sect. 5 we discuss how to solve DDP, UBDP, and RP for COTM networks. In Sect. 6 we provide translations that show the equivalence of Petri nets and OTM NCFSMs. Concluding remarks are given in Sect. 7.

## 2 Definitions

A network of communicating finite state machines (NCFSM) consists of a set of finite state processes that *send* messages to, and *receive* messages from, each other. The communication medium from one process to another is assumed

to be error-free and a potentially unbounded FIFO buffer. A *send event* causes a message to be enqueued in a FIFO buffer, while a receive event dequeues a message from a buffer. Consequently, send events can never block whereas receive events can. The definition of general NCFSMs can be found in the literature, such as in [10]. In this paper we are interested in networks where all messages from one particular process to another are all of the same type. Such networks will be referred to as one type message (OTM) networks. Note that the FIFO buffers of OTM networks behave like counters.

Let $I=\{1, 2, \ldots, n\}$, where $n\geqq 2$ is some constant denoting the number of processes in a network. Let $N=\langle P_1, P_2, \ldots, P_n\rangle$ be a network of $n$ processes. Use $C_{i\to j}$ to denote the (potentially) unbounded buffer that holds messages that process $P_i$ sends to $P_j$. It is assumed that any process $P_i$ cannot send messages to itself. Define $TG(N)$ to be a directed graph, called the topology graph of $N$, where the nodes in $TG(N)$ are the names of the processes in $N$ and an edge from $P_i$ to $P_j$ indicates the fact that process $P_i$ can directly send messages to $P_j$.

Let $s_{i,j}$ denote the event of process $P_i$ sending a message to $P_j$ (i.e. appending a message to $C_{i\to j}$) and $r_{j,i}$ the event of process $P_j$ receiving a message sent by process $P_i$ (i.e. removing a message from $C_{i\to j}$). Let $\Sigma_i^-$ be the set of send events in process $P_i$, i.e. $\Sigma_i^-=\{s_{i,j}|j\in I \;\&\; P_i\to P_j\in TG(N)\}$. Let $\Sigma_i^+$ be the set of receive events in process $P_i$, i.e. $\Sigma_i^+=\{r_{i,j}|j\in I \;\&\; P_j\to P_i\in TG(N)\}$. Use the abbreviation $\Sigma_i^\pm=\Sigma_i^-\cup\Sigma_i^+$ to denote the set of all events of process $P_i$ and $\Sigma^\pm=\bigcup_{i\in I}\Sigma_i^\pm$ to denote the set of all events of the network.

**Defintion 2.1** A communicating finite state machine $P_i$ can be specified as a four-tuple $(S_i, \Sigma_i^\pm, \delta_i, p_{0i})$, where

- $S_i$ is the set of local states of process $P_i$.
- $p_{0i}\in S_i$ is the start state of process $P_i$.
- $\Sigma_i^\pm$ is the alphabet of events.
- $\delta_i$: $S_i\times\Sigma_i^\pm\to 2^{S_i}$ is the transition function. $\delta_i(p, s_{i,j})$ is the set of states that process $P_i$ could move to from state $p$ after sending a message to process $P_j$. $\delta_i(p, r_{i,j})$ is the set of states that process $P_i$ could move to from state $p$ after receiving a message sent by process $P_j$. □

A transition $p'\xrightarrow{s_{i,j}}p$ ($p'\xrightarrow{r_{i,j}}p$, resp.) in $P_i$ is called a *send edge* (*receive edge*, resp.). A state $p$ in $P_i$ is said to be a *send* (*receive*, resp.) *state* if all of its outgoing edges are send (receive, resp.) edges. $p$ is said to be a *mixed state* if it has both outgoing send and receive edges. Define $RV(P)$ to be the set of receive states of CFSM $P$.

Let $N=\langle P_1, P_2, \ldots, P_n\rangle$ be a network of communicating finite state machines and $\mathcal{N}$ the set of non-negative integers. The semantics of the entire network can be captured by the set of global states that the network can be in. As any two messages in the same buffer cannot be differentiated, a global state consists of a collection of local states, one for each process in the network, and the number of messages in each buffer. More formally, a global state is a tuple $[p_1, p_2, \ldots, p_n, c_{2\to 1}, c_{3\to 1}, \ldots c_{n\to 1}, c_{1\to 2}, c_{3\to 2}, \ldots, c_{n\to 2}, \ldots, c_{(n-1)\to n}]$, where $p_i\in S_i$ and $c_{i\to j}\in\mathcal{N}$. Let $V_N$ be the cartesian-product of the sets $S_1, \ldots, S_n$, i.e. $V_N=S_1\times\ldots\times S_n$, and let $C_N=\mathcal{N}^{n(n-1)}$. A global state therefore belongs to $V_N\times C_N$.

**Definition 2.2** A one-type message (OTM) network of communicating finite state machines is a tuple $N=\langle P_1, \ldots, P_n\rangle$, where each $P_i$, $(i\in I)$ is a CFSM. The initial state of $N$ is $[\langle p_{0i}\rangle_{i\in I}, \langle c_{i\to j}\rangle_{i,j\in I}]$, where $c_{i\to j}=0$ $(i\neq j)$.

Let $[\langle p_i\rangle_{i\in I}, \langle c_{i\to j}\rangle_{i,j\in I}]$ be a global state. The global state transition function

$$\delta_N : (V_N \times C_N) \times \Sigma^{\pm} \to 2^{V_N \times C_N}$$

is the least partial function defined as:

1. If $p_i'\in\delta_i(p_i, s_{i,j})$ then $[\langle p_k'\rangle_{k\in I}, \langle c_{k\to l}'\rangle_{k,l\in I}]\in\delta_N([\langle p_k\rangle_{k\in I}, \langle c_{k\to l}\rangle_{k,l\in I}], s_{i,j})$, where

$$p_k'=\begin{cases} p_i' & \text{if } k=i \\ p_k & \text{otherwise}\end{cases} \quad \text{and} \quad c_{k\to l}'=\begin{cases} c_{k\to l}+1 & \text{if } (k,l)=(i,j) \\ c_{k\to l} & \text{otherwise}\end{cases}.$$

2. If $p_i'\in\delta_i(p_i, r_{i,j})$ and $c_{j\to i}>0$ then $[\langle p_k'\rangle_{k\in I}, \langle c_{k\to l}'\rangle_{k,l\in I}]\in\delta_N$ $([\langle p_k\rangle_{k\in I}, \langle c_{k\to l}\rangle_{k,l\in I}], r_{i,j})$, where

$$p_k'=\begin{cases} p_i' & \text{if } k=i \\ p_k & \text{otherwise}\end{cases} \quad \text{and} \quad c_{k\to l}'=\begin{cases} c_{k\to l}-1 & \text{if } (k,l)=(j,i) \\ c_{k\to l} & \text{otherwise}\end{cases} \quad \square$$

A network $N$ whose topology graph $TG(N)$ is a simple cycle will be called a *cyclic network*. Of special interest in this paper are cyclic OTM (COTM) networks.

*Notations*

- $[\mathbf{v}, \mathbf{c}]$ will be used to denote a global state, where $\mathbf{v}=\langle p_i\rangle_{i\in I}\in V_N$ and $\mathbf{c}=\langle c_{i\to j}\rangle_{i,j\in I}\in C_N$.
- $[\mathbf{v}_0, \mathbf{c}_0]$ will be used to denote the initial state.
- $\mathbf{v}|_i$ will be used to denote the $i^{\text{th}}$ component of $\mathbf{v}$.
- The terms *channel, buffer* and *queue* will be used interchangeably.

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

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

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

*Notations*

- We will write $\delta_N^*([\mathbf{v}_0, \mathbf{c}_0], e)$ as $\delta_N^*(e)$.
- We will drop the subscripts in $\delta_N$ and $\delta_N^*$ if no confusion arises.
- As $\forall a\in\Sigma^{\pm}$ $\delta^*([\mathbf{v}, \mathbf{c}], a)=\delta([\mathbf{v}, \mathbf{c}], a)$, we will use $\delta$ instead of $\delta^*$.
- Define $|e|_g$ as the number of times $g$ occurs in $e$, where $e$ is a string or a path in a graph.

Define the reachability set, $RS(N)$, to be the set of global states that can be reached in the network $N$, i.e., $RS(N)=\bigcup_{e\in(\Sigma^{\pm})^*} \delta(e)$. For any word $e\in(\Sigma^{\pm})^*$, we use $prefix(e)$ to denote the set of prefixes of $e$, i.e. $prefix(e)=\{e' \mid \exists e''\in(\Sigma^{\pm})^*$:

$e'\,e''=e\}$. This definition is also extended to *prefix*$(L)$ for any language $L$. Let $RV(N)$ be the set of receive state tuples $\mathbf{v}=\langle p_i\rangle_{i\in I}$, where each $p_i\in S_i$ is a receive state.

In the rest of the paper we will be dealing with sequences of events that a network of communicating finite state machines will go through. To facilitate the presentation we will use the following abbreviations:

- A word $e$ in $(\Sigma^{\pm})^*$ is called *an event sequence.*
- An event sequence $e\in(\Sigma^{\pm})^*$ is *feasible,* if

$$\forall e'\in prefix(e)\ \forall i,j\in I\ |e'|_{r_{i,j}}\leqq|e'|_{s_{j,i}}$$

- An event sequence $e$ is *executable,* if $\delta(e)$ is defined, i.e., $\delta(e)\neq\emptyset$.
- An event sequence is *stable* if (a) it is feasible and (b) it contains the same number of send and receive events of all types.

As we are dealing with interleaved execution of the processes, there could be two or more execution sequences which are reorderings of each other. The execution of such event sequences will have the same effect on the network. Thus, these interleavings should be considered *equivalent.* Formally, we have:

**Definition 2.3** Two event sequences $e_1$ and $e_2$ are *equivalent,* denoted as $e_1\simeq e_2$, iff

(1) $e_1$ and $e_2$ are reorderings of each other, and
(2) $\delta(e_1)=\delta(e_2)$. □

Much like the definition of the finite control part of a Pushdown Automaton (PDA) or of a Turing machine, we now provide a definition of the finite control part of a NCFSM called *shuffle-product.*

**Definition 2.4** Let $N=\langle P_1,\ldots,P_n\rangle$ be a NCFSM. The shuffle-product of $N$ with respect to a set $F\subseteq V_N$, written as $\otimes N(F)$, is a non-deterministic finite state automaton (NFA) $(V_N,\Sigma,T,\mathbf{v}_0,F)$, where

1. $\mathbf{v}_0=[p_{01},p_{02},\ldots,p_{0n}]\in V_N$ is the start state.
2. The transition function $T$: $V_N\times\Sigma^{\pm}\to 2^{V_N}$ is defined as

$$\mathbf{v}'\in T(\mathbf{v},a),\quad \text{where } a\in\Sigma_i^{\pm},\quad \text{iff } \mathbf{v}'|_j=v|_j\,(j\in I\ \&\ j\neq i)\ \text{ and }\ \mathbf{v}'|_i\in\delta_i(v|_i,a).$$

3. $F$ is the set of final states. □

The shuffle-product with respect to some $F\subseteq V_n$ defines a regular set $L(\otimes N(F))$, which is the set of all event sequences that leads the network from state $\mathbf{v}_0$ to $\mathbf{v}\in F$. Note that the semantics of each word $e\in L(\otimes N(F))$ is left undefined. In particular, many of the words in $L(\otimes N(F))$ are even not feasible. However $L(\otimes N(V_N))$ does capture all the executable event sequences of the network $N$, as expressed by the following proposition.

**Proposition 2.1** *Let $N$ be a network of processes over the alphabet $\Sigma^{\pm}$, then*

$$\textit{Exceution sequences of } N=L(\otimes N(V_N))\cap \textit{Feasible sequences of } N.$$

In the treatment that follows we will identify the finite control part of a shuffle-product $\otimes N(F)$ with a graph (a customary practice). Such a graph is called the *shuffle-product graph*. Recall from [4] that a Dyck language over $k$ distinct pairs of parenthesis $[_1, ]_1, \ldots, [_n, ]_n$ is the context-free language generated by the context-free grammar $G_k$, where $G_k$ has productions

$$S \to SS | [_1 S]_1 | [_2 S]_2 | \cdots | [_k S]_k | \varepsilon.$$

For a directed graph $G$, we will call a path in $G$ a *Dyck* (*PDyck*, resp.) path if the word induced by the path, i.e. the word formed by concatenating the lables of edges on the path, is a string in a Dyck language $L(prefix(L)$, resp.).

We use the notation $u \xrightarrow{\varepsilon^*} v$ to denote a path from $u$ to $v$ all of whose edges are labeled by $\varepsilon$ and $u \xrightarrow{*} v$ to denote an arbitrary path from $u$ to $v$.

We are now ready to formally define the safety properties that were informally introduced in Sect. 1.

**Definition 2.5** Let $N$ be a network of processes and $[\mathbf{v}, \mathbf{c}] \in RS(N)$ be a reachable global state.

1. $[\mathbf{v}, \mathbf{c}]$ is a *deadlock* state if $(\mathbf{v} \in RV(N)) \,\&\, (\mathbf{c} = \mathbf{c}_0)$. Namely every machine is in a local receive state and all channels are empty.
2. The communication of $N$ is *bounded* if there exists a bound $K$ on the length of the buffers in any reachable state, i.e., $\exists K \in \mathcal{N} \forall [\mathbf{v}, \mathbf{c}] \in RS(N) \, \forall i, j \in I(c_{i \to j} \leqq K)$. Otherwise we say that the communication of $N$ is *unbounded*. □

For a COTM network $N$, the problems of interests in this paper are:

*Reachability problem* (RP): Is a given state $[\mathbf{v}, \mathbf{c}]$ a reachable state in $N$?
*Deadlock detection problem* (DDP): Is a given network $N$ free of deadlocks?
*Unbounded detection problem* (UBDP): Is the communication for a given network $N$ bounded?

## 3 DDP of two-machine OTM networks

We will set the ball rolling by considering, in this section, the DDP for two-machine OTM networks. We will provide an algorithm to capture the set of all *fair* reachable states of such networks. In the next two sections we will extend the algorithm given here to deal with COTM networks.

Let $N = \langle P_1, P_2 \rangle$ be a two-machine OTM network. The global states of such a network can be captured by tuples of the form $[p_1, p_2, c_{2 \to 1}, c_{1 \to 2}]$, where $c_{2 \to 1}$ and $c_{1 \to 2}$ are non-negative integers. A reachable global state in a two-machine OTM network is said to be a *fair reachable state* provided $c_{2 \to 1} = c_{1 \to 2}$. To check for deadlocks, by definition, we are interested in finding all fair reachable states of the form $[p_1, p_2, 0, 0]$.

To generate fair reachable states, it is enough to consider execution sequences that are *fair*. An execution sequence is, by definition, an arbitrary interleaving of the actions of the two processes. On the other hand, in a fair execution sequence, every even length prefix of the execution sequence has an equal number

of steps from both processes. Put differently, in each step we force both processes to each make a move. We will show shortly that to check for deadlocks, it is sufficient to capture all fair execution sequences. To see that a fair execution sequence will always lead from one fair state to another, consider the following:

- The initial state $[p_{01}, p_{02}, 0, 0]$, as the buffers are empty initially, is a fair state.
- Let $[p, q, n, n]$ be a fair state. The combined effect of a single move made by each process from a fair state is illustrated in the following table:

| $P_1$'s Move | $P_2$'s Move | State reached after both moves |
|---|---|---|
| $p' \in \delta_1(p, s_{12})$ | $q' \in \delta_2(q, s_{21})$ | $[p', q', n+1, n+1]$ |
| $p' \in \delta_1(p, r_{12})$ | $q' \in \delta_2(q, s_{21})$ | $[p', q', n, n]$ |
| $p' \in \delta_1(p, r_{12})$ | $q' \in \delta_2(q, r_{21})$ | $[p', q', n-1, n-1]$ |
| $p' \in \delta_1(p, s_{12})$ | $q' \in \delta_2(q, r_{21})$ | $[p', q', n, n]$ |

The set of all fair reachable execution sequences can be captured by the following *fair reachable graph.*

**Definition 3.1** A fair reachable graph $FRG[P_1, P_2] = (V, E)$, corresponding to a two-machine OTM network $N = \langle P_1, P_2 \rangle$, is a graph over the parenthesis $\{s, r\}$, defined as follows. The nodes $V$ is a subset of the cartesian-product of the local states of $P_1$ and $P_2$, and the edges $E$ are labeled by $s$, $r$ or $\varepsilon$. More formally,

- $[p_{01}, p_{02}] \in V$.
- If $[p, q] \in V$, $p' \in \delta_1(p, s_{12})$ and $q' \in \delta_2(q, s_{21})$ then $[p', q'] \in V$ and $[p, q] \xrightarrow{s} [p', q']$ is an edge in $E$. This corresponds to the situation where each process can send a message to the other.
- If $[p, q] \in V$, $p' \in \delta_1(p, r_{12})$ and $q' \in \delta_2(q, s_{21})$ then $[p', q'] \in V$ and $[p, q] \xrightarrow{\varepsilon} [p', q']$ is an edge in $E$. This corresponds to the situation where process $P_2$ sends process $P_1$ a new message, which is immediately picked up by $P_1$.
- If $[p, q] \in V$, $p' \in \delta_1(p, r_{12})$ and $q' \in \delta_2(q, r_{21})$ then $[p', q'] \in V$ and $[p, q] \xrightarrow{r} [p', q']$ is an edge in $E$. This corresponds to the situation where both processes pick up a message from their respective queues.
- If $[p, q] \in V$, $p' \in \delta_1(p, s_{12})$ and $q' \in \delta_2(q, r_{21})$ then $[p', q'] \in V$ and $[p, q] \xrightarrow{\varepsilon} [p', q']$ is an edge in $E$. This captures the situation where process $P_1$ sends process $P_2$ a new message, which is immediately picked up by $P_2$. □

The Dyck paths (treat $s$ as the left parenthesis and $r$ as the right parenthesis) in a fair reachable graph are important as they correspond to fair execution paths of the original network. Furthermore, Dyck paths correspond to fair reachable states. The following lemma and its corollary make these claims precise (a similar lemma is made use of in [13]).

**Lemma 3.1** *Let $N=\langle P_1, P_2\rangle$ be a* OTM *network and let $FRG[P_1, P_2]=(V, E)$ be its fair reachable graph. The fair state $[p_1, p_2, n, n]$ is reachable iff there exists a PDyck path $u$ from $[p_{01}, p_{02}]$ to $[p_1, p_2]$ in $FRG[P_1, P_2]$ and $|u|_s - |u|_r = n$.*

*Proof If part*: By the construction of the fair reachable graph, given a PDyck path $u$ from $[p_{01}, p_{02}]$ to $[p_1, p_2]$ in $FRG[P_1, P_2]$, $|u|_s - |u|_r = n$, it is easy to recover the corresponding execution path leading from $[p_{01}, p_{02}, 0, 0]$ to $[p_1, p_2, n, n]$ in $N$.

*Only if part*: Let $e_1, e_2, \ldots e_k$ be an execution path in the network $N$ leading from $[p_{01}, p_{02}, 0, 0]$ to a fair state $[p_1, p_2, n, n]$. Among these $k$ edges, let $\#_s(P_1)$ be the number of sending edges of process $P_1$, $\#_s(P_2)$ be the number of sending edges of $P_2$, $\#_r(P_1)$ be the number of receive edges of $P_1$, and $\#_r(P_2)$ be the number of receive edges of $P_2$. As $\#_s(P_1) = \#_r(P_2) + n$ and $\#_s(P_2) = \#_r(P_1) + n$, the path $e_1, e_2, \ldots e_k$ has the same number of edges from $P_1$ and $P_2$. Let $a_1, a_2, \ldots a_l$ be the edges of $P_1$ appearing in $e_1, e_2, \ldots e_k$, and let $b_1, b_2, \ldots b_l$ be the edges of $P_2$ appearing in the same order in $e_1, e_2, \ldots e_k$. Now a path $u$ can be constructed in the $FRG[P_1, P_2]$, by considering the edges $a_i$ and $b_i$ in pairs. By induction on $k$, it can be easily shown that $u$ is a PDyck path, and $|u|_s - |u|_r = n$. □

From Lemma 3.1, we can immediately obtain the following corollary.

**Corollary 3.1** *Let $N=\langle P_1, P_2\rangle$ be a* OTM *and let $FRG[P_1, P_2]=(V, E)$ be its fair reachable graph. For a receive node $[p_1, p_2]\in V$, $[p_1, p_2, 0, 0]$ is a reachable deadlock state in $N$ iff there exists a Dyck path from $[p_{01}, p_{02}]$ to $[p_1, p_2]$ in $FRG[P_1, P_2]$.*

Given a OTM network $N=\langle P_1, P_2\rangle$, in order to check for deadlocks in it, we have to (a) construct $FRG[P_1, P_2]$ and (b) check if there are any Dyck paths from the start state $[p_{01}, p_{02}]$ to any of the receive nodes in $FRG[P_1, P_2]$. Since we are dealing with a single pair of parenthesis $s$ and $r$, we have (a) a path $u$ is a PDyck path provided $\forall w \in prefix(u)\ |w|_s \geqq |w|_r$, and (b) $u$ is a Dyck path provided it is PDyck and $|u|_s = |u|_r$. In the following, we give an $O(m_1^4 m_2^4)$ algorithm to check if there is a Dyck path in $FRG[P_1, P_2]$.

For the fair reachable graph $FRG[P_1, P_2]=(V, E)$, we define a non-deterministic finite automaton $A_1=(V, \{s, r\}, \delta_{A_1}, \mathbf{v}_0, F)$ where

(1) $\mathbf{v}' \in \delta_{A_1}(\mathbf{v}, a)$, for $a \in \{s, r, \varepsilon\}$, if the edge $\mathbf{v} \xrightarrow{a} \mathbf{v}' \in E$.

(2) $F = RV(N)$, the set of final states, is the set of receive state tuples that are present in $V$.

The NFA $A_1$ has $|V| \leqq m_1 m_2$ states. For each state $\mathbf{v}$, $\delta_{A_1}(\mathbf{v}, a)$ contains at most $m_1 m_2$ elements, where $a \in \{s, r, \varepsilon\}$. Let $L(A_1)$ be the language accepted by $A_1$. It is easy to see from the above definition and Corollary 3.1 that $L(A_1)$ contains all (if any at all) fair execution sequences that lead from $\mathbf{v}_0$ to some $\mathbf{v} \in F$. Clearly, we need to test if $L(A_1) \cap D = \emptyset$ where $D$ is the Dyck language over the alphabet $\{s, r\}$. We can do so by constructing a PDA $A$ such that $L(A) = L(A_1) \cap D$. One could achieve this by constructing a PDA $A_2$ that accepts $D$ and then constructing a PDA $A$ such that $L(A) = L(A_1) \cap L(A_2)$. But we can define a PDA $A$ directly without constructing $A_2$, as $A_2$ can have at most

**Input:** The description of a two-machine OTM network $N=\langle P_1, P_2\rangle$.
**Output:** A report as whether $N$ is free of deadlocks.
1. Construct the fair reachable graph $FRG[V, E]$ for $N$;
2. Construct the NFA $A_1$ from $FRG[V, E]$;
3. Construct the pushdown automaton $A$ that accepts (by empty stack) the language $L(A_1)\cap D$;
4. Transform $A$ into a context-free grammar $G$, such that $L(G)=L(A)$;
5. Check if the starting symbol $S$ of the grammar $G$ is a useful symbol. If it is, then declare that $N$ has deadlocks; otherwise declare $N$ free of deadlocks.

**Fig. 1.** Algorithm TWO_MACHINE_DEADLOCK

two states. Given the NFA $A_1=(V,\{s,r\},\ \delta_{A_1}, \mathbf{v}_0, F)$, we define the PDA $A=(V,\{s,r\},\ \{Z_1, Z_0\},\ \delta, \mathbf{v}_0, Z_0, \emptyset)$ and the corresponding grammar $G=(\{[\mathbf{p}, Z, \mathbf{q}]\,|\,\mathbf{p}, \mathbf{q}\in V, Z\in\{Z_0, Z_1\}\},\ \{s,r\},\ S, P)$ as follows:

| | $A_1$'s transitions | $A$'s transitions | $G$'s productions |
|---|---|---|---|
| (1) | $\mathbf{p}\in\delta_{A_1}(\mathbf{q}, s)$ | $(\mathbf{p}, Z_1 Z_0)\in\delta(\mathbf{q}, s, Z_0)$ | $[\mathbf{q}, Z_0, \mathbf{v}]\to s[\mathbf{p}, Z_1, \mathbf{u}][\mathbf{u}, Z_0, \mathbf{v}]$, $\forall \mathbf{u}, \mathbf{v}\in V$ |
| | | $(\mathbf{p}, Z_1 Z_1)\in\delta(\mathbf{q}, s, Z_1)$ | $[\mathbf{q}, Z_1, \mathbf{v}]\to s[\mathbf{p}, Z_1, \mathbf{u}][\mathbf{u}, Z_1, \mathbf{v}]$, $\forall \mathbf{u}, \mathbf{v}\in V$ |
| (2) | $\mathbf{p}\in\delta_{A_1}(\mathbf{q}, r)$ | $(\mathbf{p}, \varepsilon)\in\delta(\mathbf{q}, r, Z_1)$ | $[\mathbf{q}, Z_1, \mathbf{p}]\to r$ |
| (3) | $\mathbf{p}\in\delta_{A_1}(\mathbf{q}, \varepsilon)$ | $(\mathbf{p}, Z_0)\in\delta(\mathbf{q}, \varepsilon, Z_0)$<br>$(\mathbf{p}, Z_1)\in\delta(\mathbf{q}, \varepsilon, Z_1)$ | $[\mathbf{q}, Z_0, \mathbf{u}]\to[\mathbf{p}, Z_0, \mathbf{u}]$, $\forall \mathbf{u}\in V$<br>$[\mathbf{q}, Z_1, \mathbf{u}]\to[\mathbf{p}, Z_1, \mathbf{u}]$, $\forall \mathbf{u}\in V$ |
| (4) | | $(\mathbf{u}, \varepsilon)\in\delta(\mathbf{u}, \varepsilon, Z_0), \forall \mathbf{u}\in F$ | $[\mathbf{u}, Z_0, \mathbf{u}]\to\varepsilon, \forall \mathbf{u}\in F$ |
| (5) | | | $S\to[\mathbf{v}_0, Z_0, \mathbf{q}], \forall \mathbf{q}\in V$ |

**Proposition 3.1** *The* PDA $A$ *given above accepts (by empty stack) the language* $L(A_1)\cap D$. *The grammar* $G$ *given above has* $O(m_1^4 m_2^4)$ *productions and generates* $L(A)$.

*Proof* Let $w$ be an input string to the PDA $A$. Initially the stack only contains the initial stack symbol $Z_0$. A stack symbol $Z_1$ is pushed onto the stack whenever a letter $s$ in $w$ is read, and a stack symbol $Z_1$ is popped whenever a letter $r$ in $w$ is scanned. $A$ can empty the stack only when the top stack symbol is $Z_0$ and it is in a final state $\mathbf{u}$ of $A_1$. It follows that when the stack is empty, the sequence of letters that $A$ has read is a string in the Dyck language $D$. Moreover, that sequence can lead from the start state $\mathbf{v}_0$ of $A_1$ to the final state $\mathbf{u}$ of $A_1$. From this, it is easy to see that if $w$ is accepted by $A$, then $w\in L(A_1)\cap D$.

The grammar $G$ is constructed from $A$ by applying standard transformations that can be found elsewhere [4]. The number of type (1) transitions in $A$ is bounded by $O(m_1^2 m_2^2)$. Since each of the type (1) transitions can introduce at most $m_1^2 m_2^2$ produtions, the number of type (1) productions is bounded by $O(m_1^4 m_2^4)$. Similarly, it is easy to see that the number of type (2), (3), (4) and

(5) productions is bounded by $m_1^2 m_2^2$, $2m_1^3 m_2^3$, $m_1 m_2$, and $m_1 m_2$ respectively. Altogether, the number of productions in $G$ is bounded by $O(m_1^4 m_2^4)$. □

We note that the right hand side of each production in $G$ has at most three symbols. Once $G$ has been constructed, deadlock detection can be carried out by checking whether the start symbol $S$ of $G$ is a useful symbol. Because such a check can be done in time $O(m_1^4 m_2^4)$, the whole process takes time $O(m_1^4 m_2^4)$. The entire algorithm is summarized in Fig. 1.

## 4 Canonical execution sequences of cyclic OTM networks

The concept of fair reachability is built upon the fact that the network has only two machines. Therefore, this concept and the ensuing algorithm presented in the previous section are not directly applicable to OTM networks if $n>2$, i.e. if the OTM network has more than two machines. Furthermore, that algorithm can only solve DDP. In this section we show that in order to solve DDP, UBDP, and DDP for COTM networks, it is only necessary to capture certain canonical execution sequences, which will be shown to form a context-free language.

Let $N$ be a COTM network. Without loss of generality, assume that $P_i$ can send to $P_{i+1}$ and receive from $P_{i-1}$, $2 \leqq i \leqq n-1$, $P_1$ can send to $P_2$ and receive from $P_n$, and $P_n$ can send to $P_1$ and receive from $P_{n-1}$. Note that every two-machine OTM network is a COTM network. In the remainder of this paper, we will freely use the notation $s_{i,i+1}$ and $r_{i,i-1}$ to denote a send event or a receive event of machine $P_i$, where by convention $i+1$ should be understood as $(i \bmod n)+1$ and $i-1$ as if $(i=1)$ *then* $n$ *else* $i-1$.

Define $Dyck(1 \ldots n)$ as the Dyck language with $n$ pairs of parentheses $s_{1,2}$, and $r_{2,1}$, ..., $s_{i,i+1}$ and $r_{i+1,i}$, ..., and $s_{n,1}$ and $r_{1,n}$. Define $PDyck(1 \ldots n)$ as the language where each word $w \in PDyck(1 \ldots n)$ is a prefix of some word $x \in Dyck(1 \ldots n)$.

We state the following simple properties of Dyck and PDyck languages here, which will be used later in the proof of our main theorem in this section. The proof of this lemma is quite straightforward, and hence omitted.

**Lemma 4.1** (1) *If* $w \in \mathrm{Dyck}(1 \ldots n)$, *then* $s_{i,i+1}\, w\, r_{i+1,i} \in \mathrm{Dyck}(1 \ldots n)$.
(2) *If* $w_1$ *and* $w_2$ *are both in* $\mathrm{Dyck}(1 \ldots n)$, *then* $w_1 w_2$ *is also in* $\mathrm{Dyck}(1 \ldots n)$.
(3) *If* $w_1 \in \mathrm{PDyck}(1 \ldots n)$ *and* $w_2 \in \mathrm{Dyck}(1 \ldots n)$, *then* $w_1 w_2$ *and* $w_2 w_1$ *are both in* $\mathrm{PDyck}(1 \ldots n)$.
(4) *If* $w_1 \in \mathrm{PDyck}(1 \ldots n)$ *and* $w_2 \in \mathrm{PDyck}(1 \ldots n)$, *then* $w_1 w_2$ *and* $w_2 w_1$ *are both in* $\mathrm{PDyck}(1 \ldots n)$.

COTM networks enjoy the following nice property:

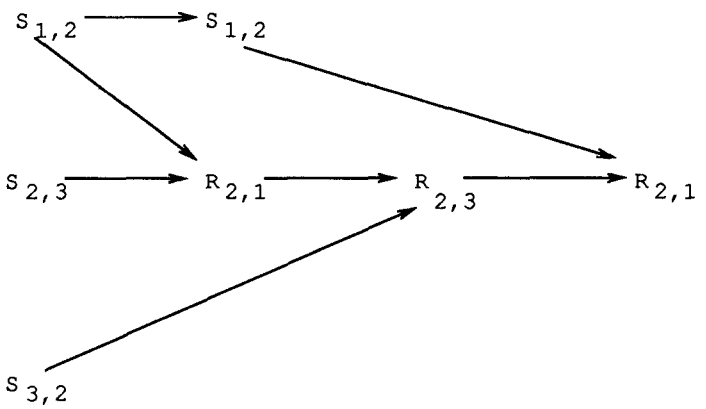
For every feasible event sequence $e$, there exists another sequence $e' \simeq e$, such that $e' \in PDyck(1 \ldots n)$.

It is this property that enables us to efficiently analyze COTM networks for RP, UBDP, and DDP. To facilitate the formal proof of this porperty, we introduce the concept of event dependence graph (EDG) for an event sequence $e$.

Let $e$ be an event sequence, and let $e_i$ be the projection of $e$ over machine $i$, i.e., the actions of process $P_i$ in the sequence $e$. The EDG for $e$ is a directed graph $\mathrm{EDG}(e)=(V_e, E_e)$, where $V_e=\{a \mid a$ is an event in $e\}$ and $E_e=E_1 \cup E_2$, where

$E_1 = \{a \to a' | \exists i$ where $a, a' \in e_i$ and $a$ is an immediate predecessor of $a'$ in $e_i\}$,
$E_2 = \{s_{i,j} \to r_{j,i} |$ where $s_{i,j}$ is the $k^{\text{th}}$ send event in $P_i$ to $P_j$ and $r_{j,i}$ is the $k^{\text{th}}$ receive event in $P_j$ with $P_i$ as the source$\}$.

Intuitively, $E_1$ reflects the dependence between events within a process, whereas $E_2$ captures the dependence between the transmission of a message by $P_i$ and its receipt by $P_j$. If two events $a_1$ and $a_2$ in $e$ are independent, then there should be no paths in EDG($e$) from $a_1$ to $a_2$ or vice versa. For example, let $e = s_{1,2}\, s_{3,2}\, s_{1,2}\, s_{2,3}\, r_{2,1}\, r_{2,3}\, r_{2,1}$, then EDG($e$) is as follows:




The following three lemmas describe several interesting properties of the EDG of a feasible event sequence $e$ in a general network.

**Lemma 4.2** *For any two nodes* $s_{i,i+1}$ *and* $r_{j+1,j}$ *in* EDG($e$), $1 \leqq$ *indegree* $(r_{j+1,j}) \leqq 2$ *and indegree* $(s_{i,j+1}) \leqq 1$.

*Proof* From the definition of EDG, if $s_{i,i+1}$ is the first event of $e_i$, *indegree*$(s_{i,i+1}) = 0$. Otherwise there is exactly one edge of the form $a \to s_{i,i+1}$, i.e. *indegree*$(s_{i,i+1}) = 1$, where $a$ is the immediate predecessor of $s_{i,i+1}$ in $e_i$. For a receive event $r_{j+1,j}$, if $r_{j+1,j}$ is the first event of $e_{j+1}$, *indegree*$(r_{j+1,j}) = 1$, since there could be exactly one edge of the form $s_{j,j+1} \to r_{j+1,j}$. Otherwise there are two edges that have $r_{j+1,j}$ as the tail node. One is of the form $s_{j,j+1} \to r_{j+1,j}$, the other is of the form $a \to r_{j+1,j}$, where $a$ is the immediate predecessor of $r_{j+1,j}$ in $e_{j+1}$. In this case *indegree*$(r_{j+1,j}) = 2$. □

**Lemma 4.3** *The event dependence graph for a feasible event sequence is acyclic.*

*Proof* An arc from $a$ to $b$ in EDG($e$) means that the occurrence of $b$ depends upon that of $a$. Therefore, a cycle in EDG($e$) would mean that $e$ is not feasible. □

Since EDG($e$) for every feasible event sequence $e$ is an acyclic graph, it can be sorted by topological sort. Moreover, as stated in the following lemma, each event sequence resulting from a topological sort is equivalent to $e$.

**Lemma 4.4** *For every event sequence* $e$, *if* $e'$ *is the sequence resulting from a topological sort of* EDG($e$), *then* $e' \simeq e$.

*Proof* For each $i \in I$, all event nodes in $e_i$ are connected by edges as a straight line in the order they occur in $e_i$. Let $s_{i,j}$ be the $k^{\text{th}}$ send event in machine $i$ with machine $j$ as destination and $r_{j,i}$ be the $k^{\text{th}}$ receive event in machine

$j$ with $P_i$ as the source. The edge $s_{i,j} \rightarrow r_{j,i}$ ensures that the receive event $r_{j,i}$ can not occur until the corresponding send event $s_{i,j}$ has occurred. □

As indicated earlier in this section, we want to show that for each feasible event sequence $e$ of a COTM network, there exists $e' \simeq e$ and $e' \in PDyck(1 \ldots n)$. By Lemma 4.4, we need only show that there exists a topological sort of EDG($e$) where the resulting sequence $e'$ is in PDyck$(1 \ldots n)$.

**Input:** a feasible event sequence $e$
**Output:** an event sequence $e' \simeq e$, and $e' \in$ PDyck$(1 \ldots n)$
compute $e_i$, $1 \leqq i \leqq n$; ($*e_i$ is the projection of $e$ over $P_i*$)
construct EDG($e$)$=(V_e, E_e)$ from $e$;
$e' := \varepsilon$; $x := \varepsilon$; $E := E_e$; $V := V_e$;
**while** $V \neq \emptyset$ **do**
**if** $\exists i \in I (e_i = s_{i,i+1}\, e'_i$ & $e_{i+1} = r_{i+1,i}\, e'_{i+1})$ **then** (* A direct matching action *)
$e' := e'\, s_{i,i+1}\, r_{i+1,i}$;
$e_i := e'_i$; $e_{i+1} := e'_{i+1}$;
$E := E - \{d_1, d_2 \mid d_1 = r_{i+1,i} \rightarrow a \in E_e, d_2 = s_{i,i+1} \rightarrow b \in E_e\}$;
$V := V - \{s_{i,i+1}, r_{i+1,i}\}$;
**else if** $x = x'\, s_{i,i+1}$ & $e_{i+1} = r_{i+1,i}\, e'_{i+1}$ **then** (* An indirect matching action *)
$e' := e'\, r_{i+1,i}$; $x := x'$;
$e_{i+1} := e'_{i+1}$;
$E := E - \{d \mid d = r_{i+1,i} \rightarrow a \in E_e\}$; $V := V - \{r_{i+1,i}\}$;
**else** (* A pushing action *)
Let $e_i$ be the project of $e$ such that rank$(e_i)$ is maximal;
$x := x s_{i,i+1}$;
$E := E - \{d \mid d = s_{i,i+1} \rightarrow a \in E_e\}$; $V := V - \{s_{i,i+1}\}$;
**endif**
**endwhile**

**Fig. 2.** Algorithm TPSort for sorting of an event sequence $e$

Define $rank(e_i)$ to be the difference between the number of send events in projection $e_i$ and the number of receive events in $e_{i+1}$, i.e.

$$rank(e_i) = |e_i|_{s_{i,i+1}} - |e_{i+1}|_{r_{i+1,i}}.$$

Let us consider the algorithm TPSort shown in Fig. 2. Given an input sequence $e$, a result sequence $e'$ is constructed by TPSort based on EDG($e$). In each iteration of the loop, one (by an indirect matching action or pushing action) or two events (by a direct matching action) are removed from $e$ and added to $e'$. We show in the following that the output event sequence $e'$ of this algorithm is in $PDyck(1 \ldots n)$.

**Lemma 4.5** *If during some iteration of the* **while** *loop there exist projections beginning with receive events, then a direct or indirect matching action can always be performed.*

*Proof* By induction on the number $k$ of pushing actions.

If before the first iteration of the **while** loop each projection $e_i$ is either $\varepsilon$ or begins with a send event, obviously the first action has to be a pushing action. If, on the other hand, at the beginning of the algorithm some projection, say $e_l$, begins with a receive event, i.e. $e_l = r_{l,l-1}\, e'_l$, then it is easy to verify that a direct matching action is always possible (otherwise $e$ would be nonfeasible, a contradiction). A similar argument shows that before the first pushing action could be performed, a direct matching actions is always possible as long as there exist projections beginning with receive events. It is easy to see that if the first pushing action pushes some send event, say $s_{i,i+1}$, onto $x$, then $rank(e_i)$ must be maximal among all the projections. Note that no indirect matching action is possible if $x = \varepsilon$.

Assume that the lemma holds before performing the $k^{\text{th}}$ pushing action, where $k > 0$. Consider the situation after the $k^{\text{th}}$ pushing action is performed. Assume to the contrary that there exist projections beginning with receive events, but no direct or indirect matching actions can be performed. If $x = \varepsilon$ we can immediately conclude that the input sequence $e$ is not feasible, a contradiction. Therefore assume that $x = x'\, s_{j,j+1}$. Because the network is COTM (cyclic and one-type-message), it is easy to verify that the above scenario is possible if and only if the following four conditions are all met:

a. If $e_i = s_{i,i+1}\, e'_i$, then either $e_{i+1} = \varepsilon$ or $e_{i+1} = s_{i+1,i+2}\, e'_{i+1}$.
b. If $e_i = r_{i,i-1}\, e'_i$, then either $e_{i-1} = \varepsilon$ or $e_{i-1} = r_{i-1,i-2}\, e'_{i-1}$. Moreover, $i \neq j+1$ (otherwise an indirect matching action would be possible because $x = x'\, s_{j,j+1}$).
c. If there exist projections beginning with send events, then there must exist some projection $e_l = s_{l,l+1}\, e'_l$, and $e_{l+1} = \varepsilon$.
d. There must exist some projection $e_t = r_{t,t-1}\, e'_t$, and $e_{t-1} = \varepsilon$.

Because the input event sequence is feasible, there must exist some send event $s_{t-1,t}$ in $x$. Write $x = x'\, s_{t-1,t}\, x''\, s_{j,j+1}$, where $x''$ does not contain any $s_{t-1,t}$. Denote by $rank(e_i, s_{m,m+1})$ the rank of projection $e_i$ just before some send event $s_{m,m+1}$ was pushed onto $x$. The induction hypothesis would imply that $rank(e_{t-1}, s_{t-1,t}) \geqq rank(e_l, s_{t-1,t})$. This says that $|e_{t-1}|_{s_{t-1,t}} - |e_t|_{r_{t,t-1}} \geqq |e_l|_{s_{l,l+1}} - |e_{l+1}|_{r_{l+1,t}}$ at the time the $s_{t-1,t}$ was pushed onto $x$. However, as currently $e_l = s_{l,l+1}\, e'_l$, $e_{l+1} = \varepsilon$, it must be the case that $rank(e_l, s_{t-1,t}) > 0$. Because $e_t = r_{t,t-1}\, e'_t$, $e_{t-1} = \varepsilon$, and $x''$ does not contain any $s_{t-1,t}$, it must be the case that $rank(e_{t-1}, s_{t-1,t}) < 0$. Therefore a contradiction arises. Hence the above scenario can never happen. □

**Lemma 4.6** *Algorithm* TPSort *correctly performs a topological sort of* EDG($e$). *Moreover, for any feasible input event sequence* $e$, $e' \in \text{PDyck}(1 \ldots n)$, *where* $e'$ *is the output sequence of the algorithm.*

*Proof* We observe that whenever a direct matching action occurs, the matched send event $s_{i,i+1}$ is the first send event in $e_i$, and the matched receive event $r_{i+1,i}$ is the first receive event $e_{i+1}$. By Lemma 4.2 $indegree(s_{i,i+1}) = 0$ since the only possible edge incident to $s_{i,i+1}$ has been removed. Similarly $indegree(r_{i+1,i}) \leqq 1$ and the only possible edge incident to the $r_{i+1,i}$ at this moment is the one from the $s_{i,i+1}$. Therefore each direct matching action observes the rule of topological sort. Similarly it is easy to see that an indirect or pushing action also observes the topological sort rule.

Termination of the algorithm is guaranteed by Lemma 4.5. Because each event is output only exactly once following the rule of topological sort, it follows that Algorithm TPSort indeed performs a topological sort of $EDG(e)$.

It remains to prove that $e' \in PDyck(1 \ldots n)$. We show this by induction on the number of iterations of the **while** loop.

Initially $e'$ is in $PDyck(1 \ldots n)$ because $e'$ is initialized as $\varepsilon$ before the **while** loop.

Assume that the conclusion is true after the $k^{\text{th}}$ iterations, $k \geqq 1$. Consider $e'$ after the $(k+1)^{\text{th}}$ iteration. There are four cases to consider, depending upon the three different actions taken inside the loop.

**Case 1** A direct matching action is performed. Then some $s_{i,i+1}\, r_{i+1,i}$ is appended to $e'$. By Lemma 4.1, the new $e'$ is still in $PDyck(1 \ldots n)$.

**Case 2** A pushing action is performed. Then some send event $s_{i,i+1}$ is pushed onto $x$ and appended to $e'$. Again by Lemma 4.1, the new $e'$ is still in $PDyck(1 \ldots n)$.

**Case 3** An indirect matching action is performed. This is a more complex situation that needs some explanation. Let us write $e'$ before the indirect matching action as $e' = e''\, s_{i,i+1}\, e'''$, where the $s_{i,i+1}$ is picked up to match the $r_{i+1,i}$. After the event $s_{i,i+1}$ was pushed onto $x$, it will not be popped until it is matched with the current $r_{i+1,i}$. Hence all the matching actions after the current $s_{i,i+1}$ was pushed onto $x$ could only take place in the substring $e'''$. In other word, there is no receive event in $e'''$ that could have been matched with some send event in $e''$ before the current matching action. Furthermore, Lemma 4.5 implies that all the send events in $e'''$ must have been matched. These two observations together with the fact that $e'$ is in $PDyck(1 \ldots n)$ implies that both $e''$ and $e'''$ are in $PDyck(1 \ldots n)$. By Lemma 4.1, both $s_{i,i+1}\, e'''\, r_{i+1,i}$ and $e''\, s_{i,i+1}\, e'''\, r_{i+1,i}$ are in $PDyck(1 \ldots n)$. □

It is easy to see that if the input event sequence to Algorithm TPSort is stable, then the output sequence is in $Dyck(1 \ldots n)$. We state this observation in the following corollary.

**Corollary 4.1** *If the input event sequence e of Algorithm* TPSort *is stable, then the output sequence* $e' \simeq e$ *and* $e \in \mathrm{Dyck}(1 \ldots n)$.

From Lemma 4.6 and corollary given above, we can immediately obtain the following theorem.

**Theorem 4.1** *Let N be a* COTM *network. For every feasible event sequence e of N, there exists another sequence* $e' \simeq e$ *and* $e' \in \mathrm{PDyck}(1 \ldots n)$. *In particular, if e is stable, then* $e' \in \mathrm{Dyck}(1 \ldots n)$.

Let $A = \otimes N(V_N)$ be the shuffle-product with respect to $V_N$ (cf. Definition 2.4), and $L(A)$ the language accepted by $A$. We can see that each string in $L(A)$ either is executable, or nonfeasible. This is so because it is impossible for a sequence from $L(A)$ to be feasible but not executable. By Theorem 4.1, for each feasible event sequence $e \in L(A)$, there exists another sequence $e' \simeq e$ and $e' \in PDyck(1 \ldots n)$. It should be clear that every string in $PDyck(1 \ldots n)$ is also feasible.

Recall that $RV(N) \subseteq V_N$ is the set of receive state tuples of $N$. Let $A_1 = \otimes N(RV(N))$ be the shuffle-product with respect to $RV(N)$, and $L(A_1)$ be the language accepted by $A_1$. With these definitions, we are now ready to give the main theorems in this paper. The correctness proofs for the following two

theorems are omitted since they follow directly from Corollary 4.1, Theorem 4.1, and the above definitions.

**Theorem 4.2** *Let $N$ be a* COTM *network. Let $L = L(A) \cap \mathrm{PDyck}(1 \ldots n)$, where $A = \otimes N(V_N)$ is the shuffle-product with respect to $V_N$. $L$ contains only executable event sequences of the* COTM *network $N$. Furthermore, for each executable event sequence $e \in L(A)$, there exists at least one event sequence $e' \in L$ and $e' \simeq e$.*

**Theorem 4.3** *Let $N$ be a* COTM *network. Let $L' = L(A_1) \cap \mathrm{Dyck}(1 \ldots n)$, where $A_1 = \otimes N(\mathrm{RV}(N))$ is the suffle-product with respect to* RV($N$). *The language $L'$ is empty iff $N$ is free of deadlocks.*

These two theorems provide the theoretical basis for algorithms to solve the three safety problems in next section. Notice that since context-free languages are closed under intersection with regular languages, the languages $L$ in Theorem 4.2 and $L'$ in Theorem 4.3 are context-free.

## 5 Algorithms for solving DDP, UBDP, and RP of COTM

Based on Theorem 4.2 and 4.3 we present, in this section, algorithms that solve the DDP, UBDP, and DDP for COTM networks. In Sect. 5.1, we present an efficient algorithm for DDP. In Sect. 5.2, a general algorithm is given to solve all the three problems of interest.

### *5.1 An algorithm for DDP in COTM networks*

Let $N = \langle P_1, \ldots, P_n \rangle$ be a COTM network, where $P_i$ has $m_i$ local states. Recall that $k = m_1 m_2 \ldots m_n$. By Theorem 4.3, to detect deadlocks, we only need to check if the context-free language $L(\otimes N(RV(N))) \cap Dyck(1 \ldots n)$ is empty. In the same spirit as the algorithm for deadlock detection in Sect. 3, we present in Fig. 3 an algorithm that solves DDP for COTM networks. To see the complexity of Algorithm COTM_DEADLOCK, we again use the following table to illustrate the transformations.

| | $A_1$'s transitions | $A$'s transitions | $G$'s productions |
|---|---|---|---|
| (1) | $\mathbf{p} \in \delta_{A_1}(\mathbf{q}, s_{i,j})$ | $(\mathbf{p}, Z_{i,j}\, Z_0) \in \delta(\mathbf{q}, s_{i,j}, Z_0)$ | $[\mathbf{q}, Z_0, \mathbf{v}] \to s_{i,j}[\mathbf{p}, Z_{i,j}, \mathbf{u}][\mathbf{u}, Z_0, \mathbf{v}]$, $\forall \mathbf{u}, \mathbf{v} \in V_N$ |
| | | $(\mathbf{p}, Z_{i,j}\, Z_{k,l}) \in \delta(\mathbf{q}, s_{i,j}, Z_{k,l})$ | $[\mathbf{q}, Z_{k,l}, \mathbf{v}] \to s_{i,j}[\mathbf{p}, Z_{i,j}, \mathbf{u}][\mathbf{u}, Z_{k,l}, \mathbf{v}]$, $\forall \mathbf{u}, \mathbf{v} \in V_N$ |
| (2) | $\mathbf{p} \in \delta_{A_1}(\mathbf{q}, r_{i,j})$ | $(\mathbf{p}, \varepsilon) \in \delta(\mathbf{q}, r_{i,j}, Z_{j,i})$ | $[\mathbf{q}, Z_{j,i}, \mathbf{p}] \to r_{i,j}$ |
| (3) | | $(\mathbf{p}, \varepsilon) \in \delta(\mathbf{p}, \varepsilon, Z_0)$, $\forall \mathbf{p} \in RV(N)$ | $[\mathbf{p}, Z_0, \mathbf{p}] \to \varepsilon, \forall \mathbf{p} \in RV(N)$ |
| (4) | | | $S \to [\mathbf{v}_0, Z_0, \mathbf{q}], \forall \mathbf{q} \in RV(N)$ |

**Proposition 5.1** *Let $N = \langle P_1, P_2, \ldots, P_n \rangle$ be a* COTM *network. The* PDA *$A$ given in Fig. 3 accepts (by empty stack) the language $L(A_1) \cap \mathrm{Dyck}(1 \ldots n)$. The grammar $G$ has $O(k^3 k_1 n^2)$ productions and generates $L(A)$, where $k = m_1 m_2 \ldots m_n$, $k_1 = m_1 + m_2 + \ldots + m_n$.*

**Input:** The description of a COTM network $N = \langle P_1, P_2, \ldots, P_n \rangle$.
**Output:** A report as whether $N$ is free of deadlocks.
1. Construct NFA $A_1 = \otimes N(RV(N))$;
2. Construct the PDA $A$ that accepts (by empty stack) the language $L(A_1) \cap Dyck(1 \ldots n)$;
3. Transform $A$ into a context-free grammar $G$, such that $L(G) = L(A)$;
4. Check if the starting symbol $S$ of the grammar $G$ is a useful symbol. If it is, then declare that $N$ has deadlocks; otherwise declare $N$ free of deadlocks.

**Fig. 3.** Algorithm COTM_DEADLOCK

*Proof* The argument that $A$ accepts $L(A_1) \cap Dyck(1 \ldots n)$ is similar to the proof of Proposition 3.1 and hence omitted.

There are $n+1$ stack symbols $Z_{1,2}, Z_{2,3}, \ldots, Z_{n-1,n}, Z_{n,1}$ and $Z_0$. $A$ has at most $O(kk_1 n^2)$ type (1) transitions. Each of these transitions can introduce $O(k^2)$ productions in $G$. Hence the number of type (1) productions in $G$ is bounded by $O(k^3 k_1 n^2)$. It is easy to see that $G$ has $kk_1 n$ type (2) productions, $k$ type (3) productions, and $k$ type (4) productions. Altogether, the number of productions in $G$ is bounded by $O(k^3 k_1 n^2)$. Moreover, the length of the right hand side of each production is at most three. □

It is easy to see that the complexity of Algorithm COTM_DEADLOCK is dominated by the time taken by Step (3) and (4). From Proposition 5.1, Step (3) takes $O(k^3 k_1 n^2)$. Because checking whether the starting symbol $S$ is useless can be done in time proportional to the length of productions, the whole algorithm needs time $O(k^3 k_1 n^2)$.

### *5.2 Solving DDP, UBDP, and RP for COTM networks*

We first provide a lemma that characterizes the relationship between the languages $PDyck(1 \ldots n)$ and $Dyck(1 \ldots n)$. The proof for this lemma is omitted to allow us to concentrate on the main theme.

**Lemma 5.1** *Each word* $y \in \mathrm{PDyck}(1 \ldots n)$ *can be written as* $y = y_1 x_1 \ldots y_k x_k y_{k+1}$, *where* $y_i \in (\Sigma^-)^*$, $1 \leqq i \leqq k+1$, *and* $x_i \in \mathrm{Dyck}(1 \ldots n)$, $1 \leqq i \leqq k$.

Let us call a path $p$ in the shuffle-product graph a *PDyck path* (or *Dyck path*, resp.) if the event sequence $e$ induced by $p$ is in $PDyck(1 \ldots n)$ (or in $Dyck(1 \ldots n)$, resp.). Consider an executable event sequence $e$ that is induced by a path $p$ in the shuffle-product $\otimes N$. Theorem 4.2 says that for each event sequence $e$, we can capture by a context-free language at least one executable event $e' \in PDyck(1 \ldots n)$ that is equivalent to $e$. This amounts to saying that for each event sequence $e$ induced by a path $p$, there is a sequence $e' \simeq e$ that is induced by a PDyck path $p'$.

Let $e'$ be the event sequence induced by a PDyck path. Since $e' \in PDyck(1 \ldots n)$, by Lemma 5.1, we can write $e' = y_1 x_1 \ldots y_k x_k y_{k+1}$, where $x_i \in Dyck(1 \ldots n)$, $1 \leqq i \leqq k$, and $y_i \in \Sigma^*$, $1 \leqq i \leqq k+1$. As each word $x_i \in Dyck(1 \ldots n)$ is a sequence of paired send and receive events, we can cancel each paired send and receive events as follows: Starting from the innermost paired send and receive events, replace them by $\varepsilon$. We then repeat this cancellation procedure to the resulting

**Input:** The description of a COTM network $N=\langle P_1, \ldots, P_n\rangle$.
**Output:** A directed graph $G=(V, E')$ such that for each executable event sequence $e$ that leads to a global state $[\mathbf{v}, \mathbf{c}]$, there exists a path $p$: $\mathbf{v}_0 \xrightarrow{*} \mathbf{v}$ where $\mathbf{c}$ can be obtained from the labels of path $p$.

1. Compute the shuffle-product graph $\otimes N=(V, E)$;
2. **while** $\exists$ edges $\mathbf{v}_1 \xrightarrow{s_{i,i+1}} \mathbf{v}_2$ **and** $\mathbf{v}_3 \xrightarrow{r_{i+1,i}} \mathbf{v}_4$ **and** $\mathbf{v}_2 \xrightarrow{\varepsilon^*} \mathbf{v}_3$ **and** $\mathbf{v}_1 \xrightarrow{\varepsilon^*} \mathbf{v}_4 \notin G$ **do**
   $E := E \cup \{\mathbf{v}_1 \xrightarrow{\varepsilon} \mathbf{v}_4\}$;
3. $E' := E - \{e \mid e: \mathbf{v} \xrightarrow{r_{i+1,i}} \mathbf{v}' \in E\}$

**Fig. 4.** Algorithm CANCEL

string until all the paired send and receive events are canceled and are replaced by $\varepsilon$ strings.

The significance of above cancellation procedure is the following. Assume that an event sequence $e' = y_1 x_1 \ldots y_k x_k y_{k+1} \in PDyck(1 \ldots n)$ leads the network to a global state $[\mathbf{v}, \mathbf{c}]$. Each send event in $x_i$ has a paired receive event in $x_i$. Therefore, the net effect to the channel of this pair of events is zero. Only the send events in $y_i$ actually contribute to the channel contents in $c$. If we can capture all these $y_i$'s, we can effectively solve the RP.

Let us get back to the shuffle-product graph. Employing the idea of cancelling paired events in a PDyck word, we can pair a send edge $\mathbf{v}_1 \xrightarrow{s_{i,i+1}} \mathbf{v}_2$ with a receive edge $\mathbf{v}_2 \xrightarrow{r_{i+1,i}} \mathbf{v}_3$ by adding an $\varepsilon$ edge $\mathbf{v}_1 \xrightarrow{\varepsilon} \mathbf{v}_3$. Next we can pair a send edge $\mathbf{v}' \xrightarrow{s_{j,j+1}} \mathbf{v}_1$ with a receive edge $\mathbf{v}_3 \xrightarrow{r_{j+1,j}} \mathbf{v}''$ by adding an $\varepsilon$ edge $\mathbf{v}' \xrightarrow{\varepsilon} \mathbf{v}''$. We observe that adding $\varepsilon$ edges in this way does not change the reachability set of the network, since an $\varepsilon$ edge means that one (or more) pair of send and receive events have been paired, and they can not contribute to the channel contents at the final state leading to by the event sequence $e'$.

Let us call a path $p$ from node $\mathbf{v}_1$ to node $\mathbf{v}_2$ in $G$ an *$\varepsilon$ path*, denoted by $\mathbf{v}_1 \xrightarrow{\varepsilon^*} \mathbf{v}_2$, if all the labels on $p$ are $\varepsilon$. The discussion given above provides the motivation for Algorithm CANCEL presented in Fig. 4.

**Theorem 5.1** *Algorithm CANCEL terminates. For a* COTM *network* $N=\langle P_1, \ldots, P_n\rangle$, *it needs time* $O(k^4 k_2 m)$, *where* $k_2 = m_1 m_2 + \ldots + m_{n-1} m_n + m_n m_1$, $k = m_1 m_2 \ldots m_n$, $m = \max(m_i : i \in I)$, *and* $m_i$ *is the number of states in machine* $P_i$.

*Proof* Termination is guaranteed, since the shuffle-product graph has a finite number of send and receive edges.

Since the network is COTM, each state in machine $P_i$ has at most $2m_i$ edges. Among them at most $m_i$ are send (to $P_{i+1}$) edges and $m_i$ are receive (from $P_{i-1}$) edges.

It is clear that $k$ is the number of nodes in the shuffle-product. Consider a node $\mathbf{v}$ in the shuffle-product graph. There are at most $m_i$ send edges with

label $s_{i,i+1}$ exiting from $\mathbf{v}$, for any $i \in I$. Each such send edges can be matched with at most $km_{i+1}$ receive edges with label $r_{i+1,i}$, i.e. the number of possible matches is bounded by $m_i m_{i+1} k$ for any $i$ at $\mathbf{v}$. Hence the number of possible matches at $\mathbf{v}$ is bounded by $k_2 k$, and the number of possible matches in the shuffle-product is bounded by $k^2 k_2$.

By carefully maintaining appropriate data structures (cf. [10]), we can ensure that (1) Each iteration of the **while** loop will match one pair of edges except during the last iteration; (2) when trying to find a receive edge $\mathbf{v}_3 \xrightarrow{r_{i+1,i}} \mathbf{v}_4$ that matches a send edge $\mathbf{v}_1 \xrightarrow{s_{i,i+1}} \mathbf{v}_2$, we only check those receive edges with labels $r_{i+1,i}$; and (3) checking the existence of an $\varepsilon$ path between any two nodes for a matching action takes no more than $k$ steps. As a result, each matching action will take no more than $k^2 m$ steps. Hence the $O(k_2 k^4 m)$ time complexity follows. □

The following lemma characterizes the relationship between Dyck paths in the shuffle-product graph $\otimes N$ and $\varepsilon$ paths in the graph $G$.

**Lemma 5.2** *Let* $\mathbf{v}_1$ *and* $\mathbf{v}_2$ *be two nodes in the shuffle-product. There exists a Dyck path* $p$: $\mathbf{v}_1 \xrightarrow{*} \mathbf{v}_2$ *where* $e' = x_1 \ldots x_k$ *is the event sequence induced by* $p$, *if and only if there exists an* $\varepsilon$ *path* $p'$: $\mathbf{v}_1 \xrightarrow{\varepsilon^*} \mathbf{v}_2$ *in the output graph* $G$, *of algorithm CANCEL*

*Proof* ($\Rightarrow$): By induction on $|e'|$ ($|e'|$ is the length of $e'$). If $|e'| = 0$, i.e. the path $p$ is a single node, the conclusion trivially holds. Assume that the conclusion is true for $|e'| = 2k$, $k > 0$. Consider the case $|e'| = 2(k+1)$, i.e. a path $p$ of length $2(k+1)$. There are two cases.

**Case 1** $e' = e'' e'''$, where $|e''| \neq 0$ and $|e'''| \neq 0$, and both $e''$ and $e'''$ are in $Dyck(1 \ldots n)$. Let us write $p$ as $p$: $\mathbf{v}_1 \xrightarrow{*} \mathbf{v}' \xrightarrow{*} \mathbf{y}_2$, where $p_1$: $\mathbf{v}_1 \xrightarrow{*} \mathbf{v}'$ induces the event sequence $e''$, and $p_2$: $\mathbf{v}' \xrightarrow{*} \mathbf{v}_2$ induces the event sequence $e'''$. By induction hypothesis, there exist path $p'_1$: $\mathbf{v}_1 \xrightarrow{\varepsilon^*} \mathbf{v}'$, and $p'_2$: $\mathbf{v}' \xrightarrow{\varepsilon^*} \mathbf{v}_2$, hence the existence of a path $p$: $\mathbf{v}_1 \xrightarrow{\varepsilon^*} \mathbf{v}_2$ follows.

**Case 2** $e' = s_{i,i+1} e'' r_{i+1,i}$, where $e''$ is in $Dyck(1 \ldots n)$. Write $p$ as $p$: $\mathbf{v}_1 \xrightarrow{s_{i,i+1}} \mathbf{v}' \xrightarrow{*} \mathbf{v}'' \xrightarrow{r_{i+1,i}} \mathbf{v}_2$, where $p_1$: $\mathbf{v}' \xrightarrow{*} \mathbf{v}''$ is the path that induces $e''$. By induction hypothesis, there exist path $p'_1$: $\mathbf{v}' \xrightarrow{\varepsilon^*} \mathbf{v}''$. The paring action of Algorithm CANCEL that pares the send edge $\mathbf{v}_1 \xrightarrow{s_{i,i+1}} \mathbf{v}'$ and the receive edge $\mathbf{v}'' \xrightarrow{r_{i+1,i}} \mathbf{v}_2$ will introduce such an $\varepsilon$ edge $\mathbf{v}_1 \xrightarrow{\varepsilon} \mathbf{v}_2$ if an $\varepsilon$ path $v_1 \xrightarrow{\varepsilon^*} v_2$ is not already in $E$.

($\Leftarrow$): By induction on the number of $\varepsilon$ edges on an $\varepsilon$ path and the details are omitted. □

Let $A(F) = (V_N, \Sigma^-, \delta, \mathbf{v}_0, F)$ be a nondeterministic finite state automaton induced by the output path $G$, i.e. a NFA with $V_N$ as the set of states, $\Sigma^{\pm}$ as the finite alphabet, $\mathbf{v}_0$ as the initial state, and $F \subseteq V_N$ as the set of final states, and the transition function $\delta$ being defined by the edges of $G$. From Lemma 5.2, we can obtain the following corollary, which allows one to check for deadlocks.

Let $A(F)=(V_N, \Sigma^-, \delta, \mathbf{v}_0, F)$ be a nondeterministic finite state automaton induced by the output path $G$, i.e. a NFA with $V_N$ as the set of states, $\Sigma^{\pm}$ as the finite alphabet, $\mathbf{v}_0$ as the initial state, and $F \subseteq V_N$ as the set of final states, and the transition function $\delta$ being defined by the edges of $G$. From Lemma 5.2, we can obtain the following corollary, which allows one to check for deadlocks.

**Lemma 5.3** *Let $N$ be a* COTM *network, and $L(A(RV(N)))$ be the language accepted by the* NFA *$A(RV(N))$ as defined above. The network $N$ has deadlock states iff $\varepsilon \in L(A(RV(N)))$.*

The following lemma presents the relationship between PDyck paths in the shuffle-product graph $\otimes N(V_N)$ and certain paths in the output $G$ of algorithm CANCEL.

**Lemma 5.4** *Let $\mathbf{v}$ be a node in the shuffle-product $\otimes N(V_N)$. There exists a PDyck path $p$: $\mathbf{v}_0 \xrightarrow{*} \mathbf{v}$ where $e'=y_1 x_1 \ldots y_k x_k y_{k+1}$ is the event sequence induced by $p$, $y_i \in (\Sigma^-)^*$, $x_i \in \mathrm{Dyck}(1 \ldots n)$, if and only if there exists a path $p'$: $\mathbf{v}_0 \xrightarrow{*} \mathbf{v}$ in the output graph $G$, where the sequence of labels induced by $p'$ is $y_1 \ldots y_{k+1}$.*

*Proof* To show the lemma in ($\Rightarrow$) let $p_i = \mathbf{v}_i \xrightarrow{*} \mathbf{v}'_i$, $1 \leqq i \leqq k$, be the Dyck path that induces the event sequence $x_i$. By Lemma 5.2, for each such path, there exists an $\varepsilon$ path $p'_i$: $\mathbf{v}_i \xrightarrow{\varepsilon^*} \mathbf{v}'_i$. Let $q_i$: $\mathbf{v}'_{i-1} \xrightarrow{*} \mathbf{v}_i$, $1 < i \leqq k$, be the path that induces the event sequence $y_i$, $q_0$: $\mathbf{v}_0 \xrightarrow{*} \mathbf{v}_1$ be the path that induces the sequence $y_1$, and $q_{k+1}$: $\mathbf{v}'_k \xrightarrow{*} \mathbf{v}$ be the path that induces the sequence $y_{k+1}$. The path $q_1 p'_1 \ldots q_k p'_k q_{k+1}$ satisfies the requirement of the lemma.

A similar argument establishes the other direction too. □

Note that any path in the final graph $G$ has only send edges or $\varepsilon$ edges.

Let $e$ be an execution sequence such that $[\mathbf{v}, \mathbf{c}] \in \delta(e)$. By Theorem 4.1 and Lemma 5.1 there exists a path in $\otimes N(V_N)$ such that $e' \simeq e$, $e'=y_1 x_1 \ldots y_{k+1}$, $y_i \in (\Sigma^-)^*$ and $x_i \in Dyck(1 \ldots n)$. We, therefore, have $\forall i \in I$: $|e|_{s_{i,i+1}} - |e_{r_{i+1,i}}| = |e'|_{s_{i,i+1}} - |e'|_{r_{i+1,i}} = |y_1 \ldots y_{k+1}|_{s_{i,i+1}}$. By Lemma 5.4, we now have our main theorem in this section.

**Theorem 5.2** *Let $N$ be a* COTM *network. Let $L(A(\{\mathbf{v}\}))$ be the language of the* NFA *(or equivalently, the set of all paths from $\mathbf{v}_0$ to $\mathbf{v}$ in $G$) as defined above. A global state $[\mathbf{v}, \langle c_{i \to i+1} \rangle_{i \in I}]$ is reachable iff $\exists u \in L(A(\{\mathbf{v}\}))$ such that $\forall i$ $c_{i \to i+1} = |u|_{s_{i,i+1}}$.*

The following corollary allows one to check for unboundedness.

**Corollary 5.1** *Let $N$ be a* COTM. *Let $L(A(\{\mathbf{v}\}))$ be the language of the* NFA *as defined above. The communication of $N$ is bounded iff $L(A(\{\mathbf{v}\}))$ is bounded.*

Given the result graph $G$ of Algorithm CANCEL, in order to check for deadlock one need to check if there are $\varepsilon$-paths from the distinguished start node to some receive node. It is easy to see that the complexity of Algorithm CANCEL would dominate the complexity of checking for deadlock, as even

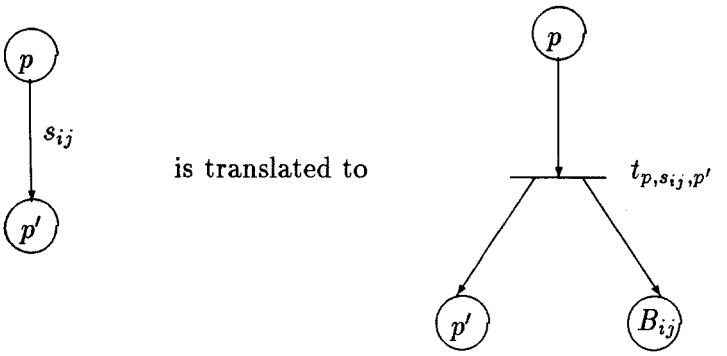



**Fig. 5.** Translation of a send edge of an OTM CFSM

the naive way of checking for $\varepsilon$ paths by using a transitive closure style algorithm would be less costly than Algorithm CANCEL. Similar comments about complexity hold for checking unboundedness, as it involves (a) removing $\varepsilon$-edges from $G$ to obtain $G'$ in a manner similar to converting NFA's with $\varepsilon$-transition to an NFA without $\varepsilon$-transitions, and (b) checking for cycles in $G'$. To summarize, the complexity of checking for deadlock and unboundedness is dominated by the cost of Algorithm CANCEL, which takes time polynomial in the size of the product of the component finite state machines, or is exponential in the number of processes in the system.

On the other hand, given graph $G$, the result of Algorithm CANCEL, it is easy to see that the reachability problem is reducible to checking for membership in a semilinear set. As the membership problem for semi-linear sets is known to be NP-complete [5], the reachability problem is in NP.

## 6 Equivalence of OTM networks and Petri nets

In this section we show that OTM networks and Petri nets are equivalent. First we recall the definition of Petri nets [12].

**Definition 6.1** A Petri Net $N$ is given by $(P, T, I, O, M_0)$ where

- $P$ is the set of places.
- $T$ is the set of transitions.
- $I \subseteq P \times T$.
- $O \subseteq T \times P$.
- $M_0 : P \to \mathcal{N}$ is the initial marking. □

As usual we will use ${}^0t({}^0p)$ and $t^0(p^0)$ to denote the preset and postset of a transition $t$ (a place $p$, respectively). A transition $t$ is *enabled* under a marking $M$ if $M(p)>0$ for all $p \in {}^0t$. An enabled transition can *fire* by removing one token from each of its incoming places and putting one token into each of its outgoing places. A marking $M'$ is *one-step reachable* from a marking $M$ by firing a transition $t$ if

$$M'(p)=\begin{cases} M(p)+1 & \text{if } p\in t^0 \text{ and } p\notin {}^0t \\ M(p)-1 & \text{if } p\in {}^0t \text{ and } p\notin t^0 \\ M(p) & \text{otherwise} \end{cases}$$

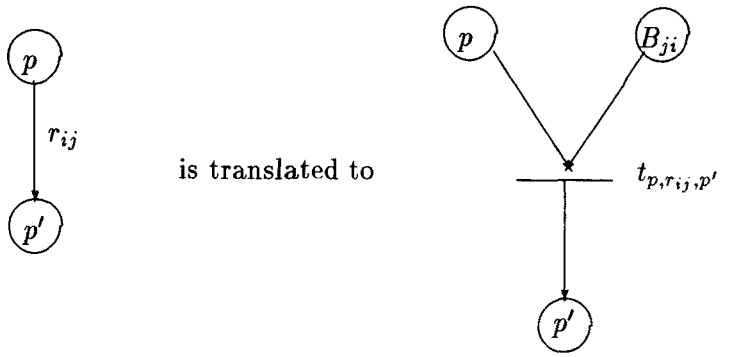



**Fig. 6.** Translation of a receive edge of an OTM CFSM

*6.1 Translation of OTM NCFSM to Petri nets*

Let $N=\langle P_1, \ldots, P_n\rangle$ be a NCFSM, where each $P_i$ is a CSFM $(S_i, \Sigma_i^{\pm}, \delta_i, p_{0i})$. We construct a Petri net $Q=(P, T, I, O, M_0)$ as follows:

- $P=\bigcup_i S_i \cup \bigcup_{i,j}\{B_{ij}\}$. Each of the places $B_{ij}$ will be used to capture the channel $C_{i\to j}$.
- The set of transitions $T=\bigcup_{i\in 1\ldots n}\{t_{p,a,q'} \mid p'\in\delta_i(p,a)\}$.
- A send edge $p \xrightarrow{s_{ij}} p'$ in process $P_i$ is translated into three arcs in $Q$: $(p, t_{p,s_{ij},p'})\in I$ and $(t_{p,s_{ij},p'}, p')$, $(t_{p,s_{ij},p'}, B_{ij})\in O$. The translation is depicted in Fig. 5.
- A receive edge $p \xrightarrow{r_{ij}} p'$ in process $P_i$ is also translated into three arcs in $Q$: $(B_{ji}, t_{p,r_{ij},p'})$, $(p, t_{p,r_{ij},p'})\in I$ and $(t_{p,r_{ij},p'}, p')\in O$. The translation is pictorially given in Fig. 6.
- The initial marking $M_0$ is defined as $M_0(p)=\begin{cases}1 & \text{if } p=p_{0i} \text{ for some } i\\ 0 & \text{otherwise}\end{cases}$

Let $\mathcal{N}$ be the set of non-negative integers. For a reachable global state $[\mathbf{v}, \mathbf{c}]$ in a OTM NCFSM $N$, a marking $M$ in $Q$ corresponding to $[\mathbf{v}, \mathbf{c}]$ should reflect both the program pointers in $\mathbf{v}$ and the channel contents in $\mathbf{c}$. Notice that by the above translation, a place $p$ in $Q$, which corresponds to the state $p\in P_i$, will have a token in it precisely when proces $P_i$ is in state $p$. The number of tokens in the place $B_{ij}$ will be equal to the number of messages in the channel $c_{i\to j}$, a component of the vector $\mathbf{c}$. Therefore there exists a one-one mapping between global states in $N$ and markings in $Q$. More precisely, we define a mapping $M_{[\mathbf{v},\mathbf{c}]}: P\to\mathcal{N}$, as follows:

$$M_{[\mathbf{v},\mathbf{c}]}(q)=\begin{cases}1 & \text{if } q\in P_i \text{ and } \mathbf{v}|_i=q\\ 0 & \text{if } q\in P_i \text{ and } \mathbf{v}|_i\neq q\\ c|_{i\to j} & \text{if } q \text{ is the place } B_{ij}\end{cases}$$

**Lemma 6.1** *Let $N$ be a* OTM NCFSM *and $Q$ a Petri net obtained from the above translation. A state $[\mathbf{v}, \mathbf{c}]$ is reachable in $N$ iff the marking $M_{[\mathbf{v},\mathbf{c}]}$ is reachable in $Q$.*

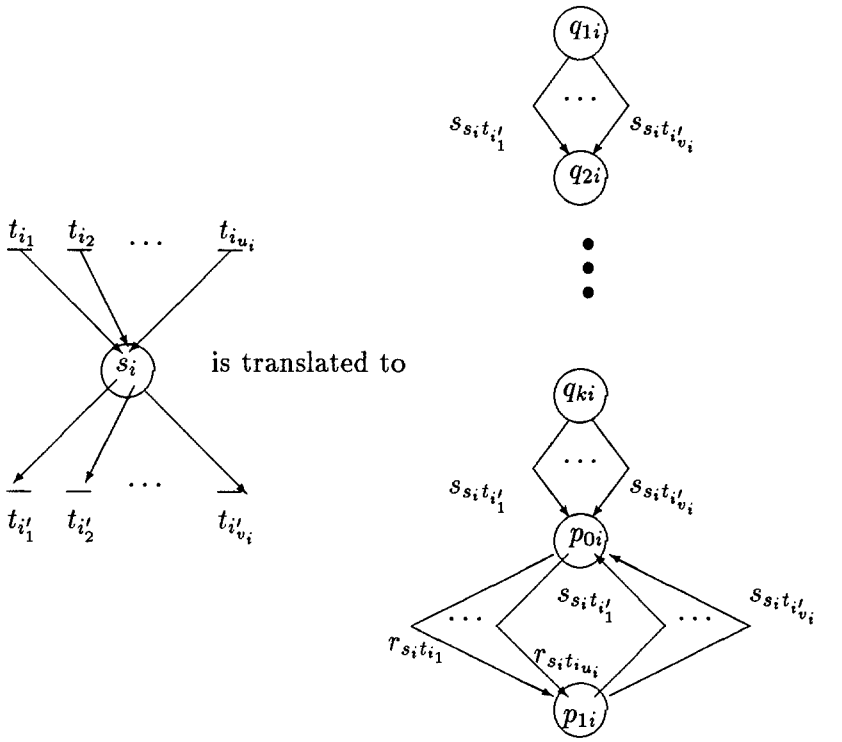



**Fig. 7.** Translating a place in a Petri net to a OTM CFSM

### 6.2 *Translation of Petri nets to OTM NCFSM*

We now provide the translation in the opposite direction. Let $N=(S, T, I, O, M_0)$ be a Petri net. Assume $S=\{s_1, s_2, \ldots, s_m\}$ and $T=\{t_1, t_2, \ldots, t_n\}$.

We will construct a OTM NCFSM $Q$ which has $m+n$ processes. Call a process corresponding to place $s_i$ (transition $t_j$) as $P_{s_i}$ ($P_{t_j}$, respectively). Thus $Q=\langle P_{s_1}, P_{s_2}, \ldots, P_{s_m}, P_{t_1}, P_{t_2}, \ldots, P_{t_n}\rangle$ where the first $m$ processes model the places of the Petri net and the last $n$ processes model the transitions of the Petri net.

Let $s_i$ be a place such that ${}^0s_i=\{t_{i_1}, t_{i_2}, \ldots t_{i_{u_i}}\}$ and $s_i^0=\{t_{i'_1}, t_{i'_2}, \ldots t_{i'_{v_i}}\}$. Let $M_0(s_i)=k$ be the initial number of tokens in $s_i$. The place $s_i$ is translated into a CFSM $P_{s_i}$, as shown in Fig. 7. Notice that the number of states of the form $q_{ji}$ depends on the number of tokens in $s_i$ in the initial marking $M_0$.

A transition $t_i$, where ${}^0t_i=\{s_{i_1}, s_{i_2}, \ldots, s_{i_{u_i}}\}$ and $t_j^0=\{s_{i'_1}, s_{i'_2}, \ldots, s_{i'_{v_j}}\}$, is translated into a CFSM $P_{t_i}$, as illustrated in Fig. 8.

Intuitively, a channel $C_{s\to t}$, where $s$ is a place and $t\in s^0$ a transition in the Petri net, is intended to hold tokens needed from $s$ for $t$ to fire. Similarly, a channel $C_{t\to s}$, where $t$ is a transition and $s\in t^0$ a place, is intended to contain tokens emitted by $t$ to $s$ as a result of $t$'s firing. The firing of a transition $t$ in the Petri net is simulated by three sequences of send and receive events:

- For each place $s\in{}^0t$, if the channel $C_{s\to t}$ is empty, a send event from machine $P_s$ to machine $P_t$ must occur;
- A sequence of receive events by $P_t$ from machines $P_s$, $s\in{}^0t$;
- A seqeunce of send events from $P_t$ to $P_s$, $s\in t^0$.

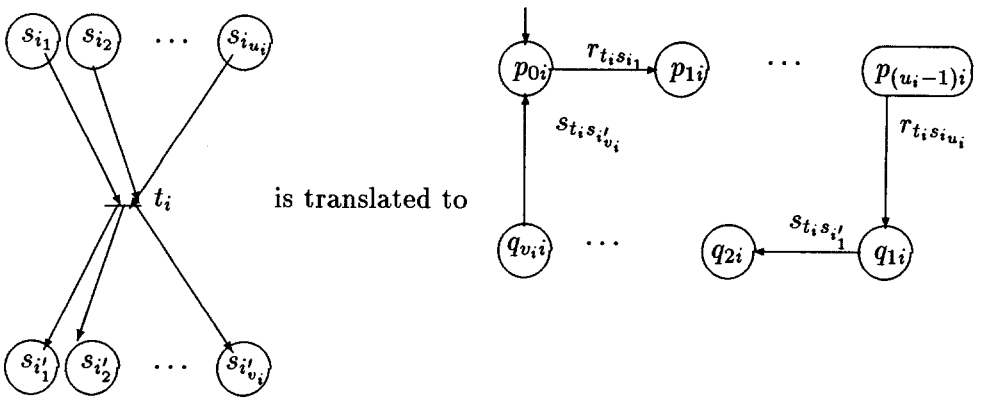



**Fig. 8.** Translating a transition in a Petri net to a OTM CFSM

Let us call the state $p_{0i}$ of a machine in both translations given above a *home state.* Intuitively, if machine $P_t$ is in its home state, where $t$ is a transition in the Petri net, then $P_t$ has not started the simulation of a firing of transition $t$ yet.

We are now ready to formally justify our translation:

**Lemma 6.2** *Let $Q$ be a Petri Net and $N$ a* OTM NCFSM *obtained from the above translation. A marking $M$ is reachable in $Q$ iff there exists a reachable global state* $[\mathbf{v}, \mathbf{c}]$ *of $N$ where each local state in* $\mathbf{v}$ *is a home state, and the number of tokens in $M(s_i)$ is equal ot the sum of the number of messages in channels $c_{s_i \to t_j}$ and $c_{t_l \to s_i}$ where $t_j \in s_i^0$ and $t_l \in {}^0 s_i$.*

Combining the above two lemmas, we have our main theorem of this section:

**Theorem 6.1** *The class of* OTM NCFSMs *and the class of general Petri Nets are equivalent in terms of their expressive power.*

## 7 Discussion

We have considered three problems – DDP, UBDP, and RP – for the class of COTM networks in this paper. Cyclic networks are a common topology among communication networks. We have shown how the reachability analysis, the deadlock and unboundedness analysis can be solved for COTM networks. In particular, we showed that the reachability sets of COTM networks are semilinear. We have also showed that OTM networks are equivalent to Petri nets.

It is clear that the class of OTM networks is equivalent to Petri nets. However, COTM networks do not correspond to any of the known subclasses of Petri nets that have semilinear reachability sets (for example, persistent nets, conflict-free nets [7], symmetric nets [6], free-choice nets, etc.). This can be easily seen by following the translation from OTM NCFSMs to Petri nets shown in the last section. Therefore, our effective construction of semilinear languages

for the reachability sets of COTM networks reveals another subclass of Petri nets with semilinear reachability sets. The problem of identifying the class of Petri nets which correspond to COTM networks is worthy of further study.

The complexity of Algorithms COTM_DEADLOCK and CANCEL are polynomial in the size of the shuffle-product graph of a network, but exponential in the number of machines. But (a) given that COTM do not correspond to any known subclass of petri nets and (b) given the lower bound result in [8], the exponential complexity is not surprising.

For DDP, it is interesting to compare the complexity of Algorithm COTM_DEADLOCK with the scheme expressed in Lemma 5.3. Recall that $k_1 = m_1 + m_2 + \ldots + m_n$, $k_2 = m_1 m_2 + m_2 m_3 + \ldots + m_n m_1$, $k = m_1 m_2 \ldots m_n$ and $m = \max(m_1, \ldots, m_n)$. It is easy to see that Algorithm COTM_DEADLOCK is superior to that scheme unless $k_1 n^2$ is larger than $k k_2 m$.

For two-machine OTM networks, Algorithm TWO_MACHINE_DEADLOCK takes time $O(m_1^4 m_2^4)$, while Algorithm COTM_DEADLOCK needs time $O(m_1^3 m_2^3 (m_1 + m_2))$. It appears at surface that the latter is more efficient than the former. However, the complexity $O(m_1^4 m_2^4)$ is obtained by assuming that the number of nodes in the fair reachable graph is $O(m_1 m_2)$. In practice, we do not expect this to happen frequently.

*Acknowledgements.* The authors would like to thank the anonymous referee who pointed out an error in an earlier proof of the Lemma 4.6.

## References

1. Brand, D., Zafiropulo, P.: On communicating finite-state machines. JACM **30**(2), 323–342 (1983)
2. Cunha, P.R., Maibaum, T.: A synchronization calculus for message oriented programming. In Proceedings of II International Conference on Distributed Computing Systems. April 1981, pp. 433–445
3. Gouda, M., Gurari, E., Lai, T.-H., Rosier, L.E.: On deadlock detection in systems of communicating finite state machines. Comput. Artif. Intell. **6**(3), 209–228 (1987)
4. Hopcroft, J., Ullman, J.: Introduction to automata theory, languages and computation. Reading: Addison-Wesley 1979
5. Huynh, D.: The complexity of semi-linear sets. Elekron. Informationsverarb. Kybernet. **18**, 291–338 (1982)
6. Huynh, D.: The complexity of the equivalence problem for commutative semigroups and symmetric vector addition systems. STOC **17**, 405–412 (1985)
7. Landweber, L., Robertson, E.: Properties of conflict-free and persistant Petri nets. JACM **25**(3), 352–364 (1978)
8. Lipton, R.: The reachability problem requires exponential space. Research Report 62, Department of Computer Science, Yale University, January 1976
9. Pachl, J.: Reachability problems for CFSMs. Research Report CS-82-12, University of Waterloo, 1982
10. Peng, W., Purushothaman, S.: Data flow analysis of communicating finite state machines. ACM TOPLAS, July 1991
11. Räuchle, T., Toueg, S.: Exposure to deadlock for communicating proceeses is hard to detect. Inf. Process. Lett. **21**, 63–68 (1985)
12. Reisig, W.: Petri nets: An introduction. Berlin Heidelberg New York Springer 1982
13. Yu, Y.T., Gouda, M.G.: Deadlock detection for a class of communicating finite-state machines. IEEE Trans. Commun. COM-**30**(12), 2514–2518 (1982)
14. Yu, Y.T., Gouda, M.G.: Unboundedness detection for a class of communicating finite state machines. Inf. Process. Lett. **17**, 235–240 (1983)