Journal of Universal Computer Science, vol. 6, no. 5 (2000), 490-502
submitted: 9/9/99, accepted: 24/3/00, appeared: 28/5/00 *© Springer Pub. Co.*

A New Approach to Communicating X-Machines Systems

Horia Georgescu
(Faculty of Mathematics, Bucharest University, Romania
email: hg@oroles.cs.unibuc.ro)

Cristina Vertan
(Faculty of Mathematics, Bucharest University, Romania
email: hg@oroles.cs.unibuc.ro)

Abstract: This paper presents a new model for the specification of communicating X-machine systems (CXMS). In previous papers, systems of X-machines have been implemented in two ways: using an unique X-machine which simulates the concurrent behaviour of several processes [1], or using several X-machines which communicate through asynchronous channels [2].
This article introduces an X-machine system for which the communication between components is done through synchronous channels. The model supposes that each X-machine has a local memory, an input and an output tape. The X-machines act simultaneously. The states of each component of the system are partitioned into processing and communicating states. Passing messages between the X-machines involves only communicating states. It is shown that, taking advantage of the behaviour of X-machines, communication using channels may be implemented, thus providing a synchronous message passing.

Key Words: Communicating X-machine systems, concurrent processes, communication using channels.

Category: F.1.2, H.2.4, D.1.3

1 Introduction

The X-machines were introduced by Eilenberg [4] in 1974, but little research has been done in this field, until Holcombe [7] used them as basis for a possible specification language. His paper was the starting point for further research. A lot of results were proved, mainly concerning their generative power and their use for verification and testing.

Very little attention has been paid to the way in which many X-machines may be integrated into a system and how they can communicate.

In [1], the behaviour of a system of grammars is studied, by simulating their concurrent action with a single X-machine. Words of a given language are placed on the input tape. At any time, a single grammar is active; afterwards, it can be used again or the control may be passed to another grammar. The language generated by the system is the language of terminal strings obtained as output. Relations between languages used as input and the corresponding generated languages are studied and results concerning the generative power of the grammars are obtained.

The first attempt to introduce communicating X-machines is due to Barnard [2]. In this article a communicating X-machine is a typed finite state machine that can communicate with other communicating X-machines via channels; these channels connect ports attached on each of the X-machines. A modular system is developed. The communicating X-machine model encapsulates dynamic and functional behaviour, as well as the data model, in one process specification. Message passing using channels is not necessarily synchronous.

In this paper, the above ideas are further studied. The X-machines act simultaneously as in [2], but for message passing (synchronous) channels are used. Communication between X-machines uses input and output ports, but also a communication matrix, introduced for this purpose. Each X-machine has its own local memory, input and output tape. Any X-machine may pass messages to any other one. The states of each component of the system are partitioned into processing and communicating states. Passing messages between the X-machines involves only communication states; for functions emerging from a communicating state, the local memory may be only observed, but never changed. In this way, internal behaviour and communication are separated. In the third section, (synchronous) channels are introduced as a way of communication between X-machines. Basic operations for sending and receiving messages are implemented.

The suitability of the new approach is proved by case studies specifying problems occurring in the concurrent processing area.

2 Communicating X-Machine Systems

For any set A, A^ϵ denotes the set $A \cup \{\epsilon\}$, where ϵ is the empty sequence. $A^\star$ denotes the free monoid generated by A.

Definition 1. A *stream X-machine* is a tuple $X = (\Sigma, \Gamma, Q, M, \Phi, F, I, T, m^0)$, where:

- Σ and Γ are finite sets, called the *input* and *output alphabets* respectively;
- Q is the finite set of *states*;
- M is a (possibly infinite) set called *memory;*
- Φ is a finite set of *partial functions* of the form: $f : M \times \Sigma^\epsilon \to \Gamma^\epsilon \times M$;
- F is the *next state* partial function $F : Q \times \Phi \to 2^Q$;
- I and T are the sets of *initial* and *final* states;
- m^0 is the *initial memory* value.

In practice M will usually be a product $\Omega_1^* \times \ldots \times \Omega_n^*$, where Ω_i are finite alphabets. It is sufficiently general [9] to model many common types of machines from finite state machines (where the memory is trivial) to Turing machines (where the memory is a model of the tape).

If, in a state, several functions can be applied, one of them is chosen randomly.

We define a *configuration* of the X-machine by (m, q, s, g) where $m \in M, q \in Q, s \in \Sigma^\star, g \in \Gamma^\star$. A machine computation starts from an initial configuration, which form is $(m^0, q^0, s^0, \epsilon)$ where $q^0 \in I$ and $s^0 \in \Sigma^\star$ is the input sequence.

Definition 2. *The output corresponding to an input sequence.* A change of configuration, denoted by $\vdash$:

$$(m, q, s, g) \vdash (m', q', s', g')$$

is possible if:

- $s = \sigma s'$, $\sigma \in \Sigma^\epsilon$
- there is a function $f \in \Phi$ emerging from q and reaching q', $q' \in F(q, f)$, so that $f(m, \sigma) = (\gamma, m')$ and $g' = g\gamma$, $\gamma \in \Gamma^\epsilon$.

$\overset{\star}{\vdash}$ denotes the reflexive and transitive closure of $\vdash$.

For any $s \in \Sigma^\star$, the output corresponding to this input sequence, computed by the stream X-machine X is defined as:

$$X(s) = \{g \in \Gamma^\star | \exists m \in M, q^0 \in I, q \in T, \text{so that } (m^0, q^0, s, \epsilon) \overset{\star}{\vdash} (m, q, \epsilon, g)\}$$

Example 1. The following example presents a simple X-machine and illustrates the necessity of introducing X-machine systems for modeling concurrent processes. The *Producer-Consumer problem*, described below, may originate from quite different areas including databases, communication protocols, operating systems, computer architecture. Distinguished items are produced, delivered to a buffer and finally consumed. The buffer is assumed to have a limited capacity. The constraints are the following:

- produce must always precede consume;
- the consumer takes the items from the buffer in the same order they were placed, i.e. the buffer is a queue;
- reading from an empty buffer must be avoided;
- writing in a full buffer must be avoided too.

We will suppose that these items are lowercase characters. The sequence of characters is ended by a special symbol #. The producer stops after sending the character #, and the consumer stops after receiving #. The consumer transforms the received characters, different from the special symbol #, in the uppercase form. The result must contain the characters produced, as well as the characters received and modified by the consumer.

In figure 1 the diagram for an X-machine simulating these activities is presented.

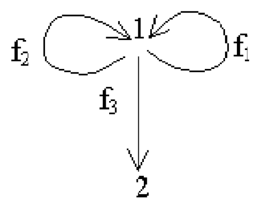

Figure 1: Simulating the Producer - Consumer problem with one X-machine

The X-machine corresponding to the diagram is the following:
$X = (\Sigma, \Gamma, Q, M, I, T, \Phi, F, m^0)$ with

- $\Sigma = \{a, b, \ldots, z, \#\}$
- $\Gamma = \{a, b, \ldots, z, A, B, \ldots, Z\}$

- $Q = \{1, 2\}$
- M is a queue, denoted with $\mathcal{L}$, of finite size n, with elements in Σ
- $I = \{1\}$
- $T = \{2\}$
- $\Phi = \{f_1, f_2, f_3\}$ where:

$$f_1(m, c) = \begin{cases} (c, mc), & \text{if } |\mathcal{L}| < n,\ c \in \Sigma \setminus \{\#\} \\ \uparrow & \text{otherwise} \end{cases}$$

$$f_2(cm, \cdot) = \begin{cases} (uppercase(c), m), & \text{if } |\mathcal{L}| > 0,\ c \in \Sigma \setminus \{\#\} \\ \uparrow & \text{otherwise} \end{cases}$$

$$f_3(\epsilon, \#) = (\epsilon, \epsilon)$$

- $F(1, f_1) = \{1\}$, $F(1, f_2) = \{1\}$, $F(1, f_3) = \{2\}$
- m^0 is the empty queue.

Obviously this is only a simulation of the real case. For a realistic solution for most applications, it is common to *decouple* the producer and the consumer by interposing a *buffer* between them. It follows that three X-machines are necessary, corresponding to the producer, to the buffer and to the consumer.

The producer's X-machine repeatedly takes items from its input tape and send them (if possible) to the buffer; it writes nothing on its output tape.

The consumer's X-machine repeatedly takes (if possible) items from the buffer and "consumes" them; it takes nothing from its input tape and places nothing on its output tape.

The X-machine corresponding to the buffer repeatedly chooses between receiving or sending an item and outputs it.

It is quite obvious that a communication between the X-machines should be provided. Firstly, we will add to each X-machine an input port and an output port. The system has now the form in figure 2.

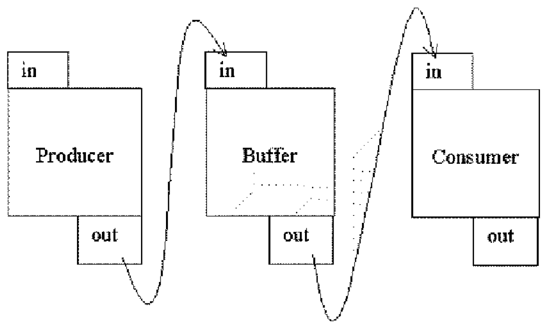

Figure 2: A system of X-machines for Producer-Consumer problem

We add also a so-called *communication matrix* (here of order 3×3) where the items to be transmitted from one X-machine to another are temporary stored in. Let us suppose that the Producer (the X-machine with number 1) produces an item i; it means that it must send it to the Buffer (the X-machine with number 2). It follows that the producer has to move item (letter) i to its output port and from there to C_{12}, while the Buffer has to copy the value of C_{12} into its input port.

Of course, several problems are still to be solved, mainly concerning the synchronization between the actions of the X-machines in the system and a proper handling of the common matrix (memory) C, due to the fact that several X-machines can access the same element of C.

Following the example above, we will introduce the definition for Communicating X-machines systems. The example will be reviewed in section 3 and a complete solution will be provided.

In the sequel we will assume that each X-machine P_i has, besides the information in the standard definition 1, an input port in_i and an output port out_i. They are two additional distinct memory locations used for receiving and sending messages; their values may be those of M_i or the special symbol λ.

Definition 3. A *Communicating X-Machines System* ($CXMS$ for short) with n components is a 4-tuple $CXMS_n = ((P_i)_{i=1,\dots,n}, C, C^0, O)$, where:

- P_i is the X-machine with number i, $P_i = (\Sigma_i, \Gamma_i, Q_i, M_i, in_i, out_i, F_i, I_i, T_i, m_i^0)$;
- C is a matrix of order $n \times n$, used for communication between the X-machines;
- C^0 is the initial content of C;
- O is the output tape of the system; at any time, the content of O has the form $(g_1, \dots, g_n) \in \Gamma_1^\star \times \dots \Gamma_n^\star$. For any i, O_i will denote the output tape of the i^{th} component of the system.

For each pair (i, j) with $i, j \in \{1, ..., n\}$, $i \neq j$, C_{ij} is used as a temporary buffer for passing "messages" from the X-machine P_i to the X-machine P_j. Initially, $C_{ij} = C_{ij}^0$ has one of the values λ or @, as passing messages from P_i to P_j is intended or not. For all i, $C_{ii} =$ @ because an X-machine never passes a message to itself.

$$C_{ij} = \begin{cases} \lambda \text{ if communication between } P_i \text{ and } P_j \text{ is allowed} \\ @ \text{ otherwise} \end{cases}$$

The actual messages passed from an X-machine to another can not be λ, @ or \$, which are used for special purposes, further described.

The mechanism of passing a value (message) will be explained in detail later. For the moment, we mention only that each X-machine P_i can access (read from or write into) only the i^{th} column and i^{th} row of the communication matrix. We will denote this accessibility domain by $+_i$. At any time, a location C_{ij} can contain a single piece of information.

Remark. For sake of simplicity, in the above definition we suppose that all messages passed from any X-machine to any other one have the same type. This does not restrict the generality of the model, since any message could begin with a flag indicating the type of the message. This flag could be used by the receiver in order to decode correctly the message.

In each X-machine P_i there are two kinds of states: $Q_i = Q_i^{'} \cup Q_i^{"}, Q_i^{'} \cap Q_i^{"} = \emptyset$, where $Q_i^{'}$ contains *processing states* and $Q_i^{"}$ contains *communicating states*. In the diagrams below, any state x will be represented as $\underline{x}$ (if it is an processing state), as $\underline{\underline{x}}$ (if it is a communicating state) or as x (if it can be either an processing or a communicating state). The final states are always processing states.

Let $\underline{\underline{x}}$ be a communicating state of the X-machine P_i, let $f_1, \ldots, f_k \in \Phi_i$ be the functions emerging from it and let $y_1, \ldots, y_k \in Q_i$ be their destinations, as in Fig. 3. Then any function f_s depends on $+_i$, M_i, in_i, out_i and may change

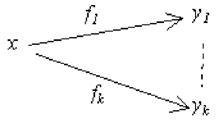

Figure 3: States and functions emerging from them.

the values of $+_i$, in_i, out_i. This can be done in three ways:

1) a value is moved from out_i to C_{ij}, for some $j \neq i$:
if $condition_s$ **and** $C_{ij} = \lambda$ **and** $out_i \neq \lambda$
then $C_{ij} \leftarrow out_i$, $out_i \leftarrow \lambda$
where $condition_s$ depends on M_i ;
2) a value is moved from C_{ji} to in_i, for some $j \neq i$:
if $condition_s$ **and** $C_{ji} \notin \{\lambda, @\}$
then $in_i \leftarrow C_{ji}, C_{ji} \leftarrow \lambda$
where $condition_s$ depends on M_i ;
3) under some condition, some elements of $+_i$ are modified:
if $condition_s$ **then** $modify +_i$,
where $condition_s$ depends on M_i, in_i and out_i and the modifications consist only in changing some elements of $+_i$ into @, λ or \$.

Remark. For functions emerging from a communication state, the local memory may be only observed, but not changed.
If more than one of the functions $f_1, \ldots, f_k$ may be applied, one of them is chosen arbitrarily to act. If none of these functions may be applied, the X-machine does nothing (so it does not change the state).
Let now $\underline{x}$ be an processing state, which is not a final one, of the X-machine P_i, let $f_1, \ldots, f_k \in \Phi_i$ be the functions emerging from it and let $y_1, \ldots, y_k \in Q_i$ be their destinations, as in Fig. 3. Then any function f_s depends on $\Sigma_i^{\epsilon}, M_i, in_i, out_i$ and is meant to (partially) change the content of M_i, in_i, out_i and possibly add some information to the local output tape O_i. If more than one of the functions $f_1, \ldots, f_k$ may be applied, one of them is chosen arbitrarily to act. If none of these functions may be applied, the X-machine blocks and so does the entire system; the content of the output tape is not significant in this case.

Remark. The system starts with all X-machines in their initial states, $C = C^0$, $M_i = m_i^0$, $in_i = \lambda$ and $out_i = \lambda$ for all $i \in \{1, \ldots, n\}$. The X-machines act simultaneously. The system stops successfully when all X-machines reach final states; in this case the result is the n-tuple $(g_1, \ldots, g_n)$, $g_i \in \Gamma_i$, $i \in \overline{1,n}$.

From the remarks above, it follows that a $CXMS$ is nondeterministic. The nondeterminism is provided in two ways:

- by means of the communicating states and the matrix C;
- by means of the processing states' behaviour of each X-machine.

Definition 4. An X-machine P_i in a $CXMS$ is called *deterministic with respect to processing states* (for short *PS-deterministic*) if:

1) $F_i : Q_i \times \Phi_i \to Q_i \;\; \forall i = 1, \ldots, n$;
2) for any processing state, any content of the local memory and of the two additional memory locations (*in* and *out*), exactly one function can be applied.

Remark. Mutual exclusion or other synchronization are assumed when working with the common memory C.

3 Communicating X-machines systems using channels

The mechanism introduced above assures only a low level of synchronization. In this paragraph we will introduce channels as a higher level of synchronization. The mechanism resembles that found in Occam (INMOS, 1984) and the formalism CSP (see [6]). The CXM systems prove to be a natural way for implementing intercommunication between the components, namely through *channels*.

The classical communication through channels is described further. It involves *send* and *receive* operations; the operations on each channel are synchronized. Each channel has a single sender and a single receiver. Whichever process arrives at a channel operation first, will be blocked until the process at the other end of the channel reaches the complementary operation. When both processes are ready, a rendezvous is said to take place, with data passing from the output of the sender to the input of the receiver. Only after this message passing is complete can the two processes act further.

We will simulate this kind of synchronous communication for X-machines. The special symbol \$, mentioned in section 2, will be used namely as an acknowledging signal that the message was received. Let us suppose that we intend to send a message from P_i to P_j (i.e. the content of out_i to in_j, of course using C_{ij}) in the same way as a transmission through channels is done (see [3], [5]). The functions emerging from a communication state in P_i are restricted to send/receive a message to/from P_j and have respectively the following two constructs:

a) **when** $condition \Rightarrow j \,!\, out_i$
for some $j \neq i$, where $condition$ depends only on M_i;
b) **when** $condition' \Rightarrow j \,?\, in_i$
for some $j \neq i$, where $condition'$ depends only on M_i;

In fact these are *macrofunctions*. Their diagrams are showed in Fig. 4 a and Fig. 4 b, where $-$ stands for no action:

f_1: **if** *condition* **and** $C_{ij} = \lambda$ **and** $out_i \neq \lambda$
then $C_{ij} \leftarrow out_i$; $out_i \leftarrow \lambda$

f_2: **if** $C_{ji} = \$$
then $C_{ji} \leftarrow \lambda$

g_1: **if** *condition'* **and** $C_{ij} \neq \lambda$
then $in_j \leftarrow C_{ij}$; $C_{ij} \leftarrow \lambda$

g_2: $C_{ji} \leftarrow \$$

g_3: **if** $C_{ji} = \lambda$
then $-$

It is important to stress the fact that the conditions appearing in a) and b) are included in f_1 and g_1. In this way, even if the condition is fulfilled, it does not mean that the function may be chosen without further checking.

Assumption. We will assume that it is the programer's duty to ensure, when the CXM system is working, that for each channel operation the complementary one is provided. The situation when two X-machines try simultaneously to send or simultaneously to receive messages between them has to be avoided.

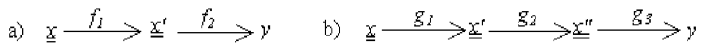

Figure 4: State diagrams for implementing channels for intercommunication between the components of a $CXMS$.

Theorem 1. *Under the above assumption, the communication between X-machines using the simulation of channels described above works correctly.*

Proof. Let us suppose that the two X-machines involved in communication are P_i and P_j. Initially $C_{ij} = C_{ji} = \lambda$. We recall that only P_i and P_j can modify C_{ij} and C_{ji}. Let us assume that P_i chooses to send a value to P_j and *condition* = *condition'* = *true*. Then, at the beginning of the sending operation, the only possible sequence is: f_1, g_1, g_2, f_2. Afterwards, two cases are possible:
I) If P_j executes g_3 then the send and receive operations are completed.
II) There is a delay in P_j before the execution of g_3 (P_j sleeps for a while). During this delay $C_{ji} = \lambda$.
According to the possible actions of P_i, the following cases have to be studied:

1) P_i will not communicate again with P_j; when P_j awakes it will execute g_3.

2) P_i tries again to send a value to P_j (i.e. $out_i \neq \lambda$), *condition* = *true* and $C_{ij} = \lambda$. P_i is blocked on f_2, so when P_j awakes it will execute g_3. Following

the assumption that to every *send* operation a *receive* is associated, P_j will try again to receive a value from P_i. This value will be received while executing the function g_1; then P_j will execute g_2, so there $C_{ji} \leftarrow \$$. P_i can now resume execution.

3) P_i tries to receive a value from P_j. Consequently it will try to execute the following sequence of functions:

g_1: **if** *condition''* **and** $C_{ji} \neq \lambda$ **then** $in_i \leftarrow C_{ji};\ C_{ji} \leftarrow \lambda$

g_2: $C_{ij} \leftarrow \$$

g_3: **if** $C_{ij} = \lambda$ **then** $-$

As $C_{ji} = \lambda$, P_i will be blocked on g_1; when P_j awakes, it will execute g_3.

Corollary 1*. Under the assumption in theorem 1, deadlock can not occur during the actual message passing.*

Example 2 . The Producer-Consumer problem with bounded buffer. The problem will be modeled by means of a $CXMS$ with 3 components: P_1, P_2 and P_3. The initial form C^0 of the communication matrix C is:

$$\begin{matrix} @ & \lambda & @ \\ \lambda & @ & \lambda \\ @ & \lambda & @ \end{matrix}$$

P_1 corresponds to the producer. m_1^0 contains the items that the producer takes from the input tape to place them into the buffer. We have $I_1 = \{\underline{1}\}$ and $T_1 = \{\underline{5}\}$. No information is added to the local output tape. The state transition diagram for P_1 appears in Fig. 5 a.

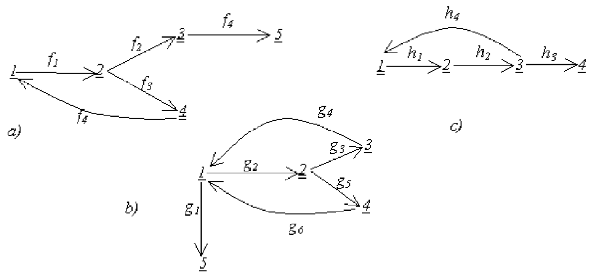

Figure 5: The Producer-Consumer problem. State transition diagrams for: a)P_1; b)P_2 c)P_3

f_1: $out_1 \leftarrow \text{first}(M_1)$;
$M_1 \leftarrow \text{tail}(M_1)$;

f_2: **if** $out_1 = \#$ **then** $-$

f_3: **if** $out_1 \neq \#$ **then** $-$

f_4: $2\,!\,out_1$

P_3 models the activities of the consumer. $m_3^0 = \emptyset$, $I_3 = \{\underline{1}\}$ and $T_3 = \{\underline{4}\}$. No information is added to the local output tape. The state transition diagram is showed in Fig. 5 c.

h_1: $2\,?\,in_3$

h_2: in_3 is "consumed"

h_3: **if** $in_3 = \#$ **then** $-$

h_4: **if** $in_3 \neq \#$ **then** $-$

The X-machine P_2 implements the activities concerning the buffer. Let max be the size of the queue Q, and ok an integer variable initialized with 2. Variable ok will decrease to 1 after the character z is received from P_1 and will decrease to 0 when the same character, in uppercase form, is sent to P_3. The internal memory of P_2 includes Q, max, ok, $last$ and nr, where nr is the current number of items in Q and $last$ has the following meaning: whenever in state $\underline{\underline{2}}$ a sending operation is chosen to be performed, $last$ has the same value as out_2 ; initially, $last = \lambda$, $nr = 0$. We have $I_2 = \{\underline{1}\}$ and $T_2 = \{\underline{5}\}$. "$\Leftarrow$" is the operator used for extracting an item from the queue Q, while "$\Rightarrow$" is the operator used for adding an item to the same queue. Of course the operator "$\Rightarrow$" has not to be confused with "=>" appearing in the functions emerging from a communication state.
Additionally, the X-machine P_2 uses its output tape for keeping record of the items received and sent; therefore, we chose to transform each item of the queue, different from the special character #, in the uppercase form, before sending it to the output tape and to the Consumer. The state transition diagram appears in Fig. 5 b.

g_1: **if** $ok = 0$ **then** $-$

g_2: **if** $ok \neq 0$ **then** $-$

g_3: **when** $ok = 2$ **and** $nr < max$
=> $1\,?\,in_2$

g_5: $3\,!\,out_2$

g_4 : **if** $inn_2 \neq \#$ **then** $ok \leftarrow ok - 1$
else add in_2 to O_2;
$in_2 \Rightarrow Q$; $nr \leftarrow nr + 1$;
if $out_2 = \lambda$
then $out_2 \Leftarrow Q$; $nr \leftarrow nr - 1$;
  if $out_2 \neq \#$
  then $out_2 \leftarrow upper(out_2)$;
  $last \leftarrow out_2$

g_6 : **if** $last = \#$ **then** $ok \leftarrow ok - 1$
else add $last$ to O_2;
if $nr > 0$
then $out_2 \Leftarrow Q$; $nr \leftarrow nr - 1$;
  if $out_2 \neq \#$
  then $out_2 \leftarrow upper(out_2)$;
  $last \leftarrow out_2$

Example 3 Finding the first n prime numbers. We will introduce a CXM system with $n+2$ components, in fact a pipeline of X-machines labeled with $P_0, P_1, \ldots, P_{n+1}$.
The main activity of the X-machine P_0 is to pump the numbers 2,3,... to P_1. The complete activity of P_0 will be described below.
For $i = 1, \ldots, n$, the X-machine P_i does the following: the first number it receives from P_{i-1} is stored as a witness value and added to its output tape. For the following numbers it receives, it checks if these are primes "from its point of view", i.e. if the witness value does not divide them; if so, the number is passed to P_{i+1} (for further checking), otherwise it is discarded. Obviously, the witness values of $P_1, \ldots, P_n$ are the first n prime numbers.
The X-machine P_{n+1} acts as follows:

- receives a number from P_n;
- sends the value -1 to P_0;
- successively receives numbers from P_n until the received value is -1.

We describe now the complete activity of P_0. At each step, it chooses to send a number to P_1 or to receive, if possible, a value from P_{n+1}. When receiving the value -1 from P_{n+1}, it sends it to P_1 and stops.
The X-machines P_1, $P_2, \ldots,$ P_n, P_{n+1} will stop, in this order, after receiving the value -1.
The initial form C^0 of the communication matrix C is:

$$\begin{array}{cccccc}
@ & \lambda & @ & \ldots & @ & \lambda \\
\lambda & @ & \lambda & \ldots & @ & @ \\
@ & \lambda & @ & \ldots & @ & @ \\
. & . & . & . & . & . \\
@ & @ & @ & \ldots & @ & \lambda \\
\lambda & @ & @ & \ldots & \lambda & @
\end{array}$$

For the X-machine P_0, m_0 contains a variable i initialized with 2, and a boolean variable *ok* initalized with $false$. We have $I_0 = \{\underline{1}\}$ and $T_0 = \{\underline{6}\}$. No information is added to O_0. The state diagram appears in Fig. 6 a, where:

f_1: **if** not ok **then** $out_0 \leftarrow i$

f_2: $1\,!\,out_0$

f_3: $in_0\,?\,n+1$

f_4: $i \leftarrow i+1$

f_5 : $ok \leftarrow true$; $out_0 \leftarrow in_0$

f_6 : **if** ok **then**$-$

Remark. The conditions "**if** $C_{01} = \lambda$" in f_2 and "**if** $C_{n+1,0} \neq \lambda$" in f_3 are implicit, so that in fact the choice between these two functions is not completely non-deterministic.

For each $i = 1, \ldots, n$, the internal memory M_i of the X-machine P_i contains two cells x (the "witness value") and y. $T_i = \{\underline{8}\}$ and $I_i = \{\underline{1}\}$. The witness value is added to the output tape O_i. The state diagram is shown in Fig. 6 b, where:

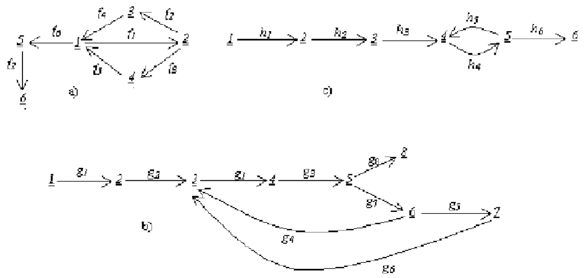

Figure 6: Finding the first n prime numbers. The state transition diagrams for: a) P_0; b) P_i, $i = 1, \ldots, n$; c) P_{n+1}

g_1: $i-1\,?\,in_i$	g_2: $x \leftarrow in_i$; add x to O_i
g_3: $y \leftarrow in_i$	g_4: **if** y mod $x = 0$ **then** $-$
g_5: **if** y mod $x \neq 0$ **then** $out_i \leftarrow y$	g_6: $i+1\,!\,out_i$
g_7: **if** $y \neq -1$ **then** $-$	g_8: **if** $y = -1$ **then** $-$

The internal memory of the X-machine P_{n+1} is void. We have $I_{n+1} = \{\underline{1}\}$ and $T_{n+1} = \{\underline{6}\}$. No information is added to O_{n+1}. The state transition diagram appears in Fig. 6 c, where:

h_1: $n\,?\,in_{n+1}$	h_2: $out_{n+1} \leftarrow -1$
h_3: $0\,!\,out_{n+1}$	h_4: $n\,?\,in_{n+1}$
h_5: **if** $in_{n+1} \neq -1$ **then** $-$	h_6: **if** $in_{n+1} = -1$ **then** $-$

4 Conclusions

In this paper we have presented a new formal specification for systems of communicating X-machines, as an extension of the X-machine model. Each X-machine has its own internal memory and two additional memory locations used for sending and receiving messages, as well as an output tape. It enables us to distinguish between processing and communicating states. In this way internal and external behaviour can be studied separately, which seems to be adequate for our further work. Of course, at the same time some X-machines may be in processing states while the others are in communicating states.

It is shown that the communication between the X-machines in the system can be achieved through channels, providing a synchronous message passing, so that most of the problems that appear in concurrent programming may be modeled by $CXMS$ (see theorem 1 and its corrolary).

We are currently working on designing an automatic method which, for any deterministic $CXMS$, generates a concurrent program written in a Pascal-FC style language (see [3], [5]).

Further work will include verification and testing. These have to be done separately for the internal and external behaviour of the components of the system. Techniques presented in [8] have to be adapted and developed. Reachability aspects have to be studied for both behaviours too.

References

[1] Bălănescu, T., Georgescu, H., Gheorghe, M. : Stream X-machines with underlying distributed grammars, *Informatica* (to appear).

[2] Barnard, J., Whitworth, J., Woodward, M. : Communicating X-machines, *Journal of Information and Software Technology*, Vol. 38, no. 6, (1996).

[3] Burns, A., Davies, G. : *Concurrent Programming*, Addison Wesley, 1993.

[4] Eilenberg, S. : *Automata, Languages and Machine*, Vol. A, Academic Press, 1974.

[5] Georgescu H.: *Concurrent Programming*, Ed. Tehnică, Bucharest, (in Romanian), 1997.

[6] Hoare, A. : *Communicating Sequential Processes*, Prentice Hall, 1985.

[7] Holcombe, M. : X-machines as a basis for dynamic system specification, *Software Engineering Journal* 3 (1988), 69 - 76.

[8] Holcombe, M., Ipate, F. : *Correct Systems : Building a Business Process Solution*, Springer Verlag, Berlin, 1998.

[9] Ipate, F., Holcombe, M. : Another Look at Computability, *Informatica*, 20(1996), 359-372.