Formal Aspects of Computing (2000) 12: 485–500

Formal Aspects of Computing

A Structured Way to Use Channels for Communication in X-Machine Systems

Anthony J. Cowling[1], Horia Georgescu[2] and Cristina Vertan[2]
[1]Department of Computer Science, University of Sheffield, UK
[2]Faculty of Mathematics, Bucharest University, Romania

Abstract. This paper presents a new model for passing messages in communicating stream X-machine systems (CSXMS). The components are stream X-machines with ϵ-transitions, acting simultaneously. The states are partitioned into processing and communicating states. Passing messages between the X-machines involves only communicating states. A communication matrix is used as a common memory. It is shown that a structured way of using channels, namely via *select* constructs with guarded alternatives and *terminate* clause, may be implemented. An automatic scheme for writing concurrent programs in an Ada-like style, starting from a CSXMS, is proposed.

Keywords: Channel operations; Channels; Communicating systems of X-machines; X-machines

1. Introduction

The concept of X-machines was introduced by Eilenberg [Eil74], but they were not intensively studied until Holcombe used them for specification purposes [Hol88]. This led to further research, which has proved the power of the model.

The X-machine model extends the finite-state machine one. A new set Φ of basic processing functions is added. The transitions between states are performed according to these functions. Another set X is added to the model in order to characterise the internal memory of the machine. An input and an output tape are also considered. The machine evolves from one state to another according to the current state, the content of the input tape and internal memory and the function chosen to be applied. After the transition takes place a new item may be added to the output tape.

During recent years much work has been done related to the generative power of these machines and the possible use of them for testing [Cho78, HoI98, IpH96, IpH97, IpH01]. Very little attention has been paid to the possible communication between these machines and consequently their use for specification of concurrent processes. In [BGG98a, BGG98b, BGG98c, BGG01] cooperating distributed grammar systems are used for modelling their concurrent behaviour.

Correspondence and offprint requests to: Tony Cowling, Department of Computer Science, University of Sheffield, Regent Court, 211 Portobello Street, Sheffield S1 4DP, UK. Email: A.Cowling@dcs.shef.ac.uk. Or Horia Georgescu, Department of Computer Science, Faculty of Mathematics, University of Bucharest, 14 Academiei str., 70 109 Bucharest 1, Romania. Email: hg@phobos.cs.unibuc.ro.

In [BWW96] Barnard specified a model for communicating X-machines extending the X-machine model. According to [BWW96] a communicating X-machine is a typed finite state machine which can communicate with other X-machines via channels; these channels connect ports on each of the machines. However, the message passing is not necessarily synchronous and is not done in a structured way.

Another approach is proposed in [GeV00], where the communication is done through a communication matrix. Using this model in [BCG99] it is proved that communicating stream X-machine systems are equivalent (from the generative power point of view) with a single X-machine. This model does, however, rely on assumptions about the way in which the communication matrix is used, and it needs to be shown that these assumptions can be justified.

In this paper we propose a slightly different version of the communicating X-machine model described in [GeV00], and we show how this new model justifies the assumptions that are made in [BCG99]. This model allows the use of channels as a mechanism for passing messages between the X-machines, and it shows how the specific constructs for channels, such as *select* and *terminate*, can be implemented. Using this model of communication in a system, we propose an automatic scheme for the generation of a concurrent program, in which each process is associated with an X-machine.

2. Basic Definitions

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 2.1. A *stream X-machine with* ϵ*-transitions* 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* 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

together with an input tape, an output tape and some state variables.

In the above definition, the set of variables together with the states of the tapes form a tuple which is called the *configuration* of the machine.

Definition 2.2. A *configuration* of a stream X-machine with ϵ-transitions is a tuple: $x = (m, q, s, g)$ where:

- $m \in M$ is the *memory* of the machine;
- $q \in Q$ is the *state* of the machine;
- $s \in \Sigma^{\star}$ is its *input tape*; and
- $g \in \Gamma^{\star}$ is its *output tape.*

Note. Normally M will be structured as a product $\Omega_1 \times \Omega_2 \times \ldots$, where Ω_i are finite alphabets. This structure is sufficiently general [IpH96] to model many common types of machines from finite state machines (where memory is trivial) to Turing machines (where the memory is a model of a tape).

In particular, this model also corresponds to a RAM memory, where there is a tuple of individual variables $(m_1, m_2, \ldots)$ having values drawn respectively from the alphabets $\Omega_1, \Omega_2 \ldots$. For such a memory structure, the normal programming language convention will be followed, that a name such as m_1 denotes both the individual variable itself and also its current value, as the context requires. Since it is clear in each definition which names denote variables and which names denote fixed sets of values, it has not been considered necessary here to introduce separate notations to distinguish these two concepts.

The dynamics of this machine are that a computation starts from an initial configuration $(m^0, q^0, s^0, \epsilon)$ where $q^0 \in I$ and $s^0 \in \Sigma^{\star}$ is the input sequence. The computation then proceeds by a succession of transitions, each of which produces a change in the configuration of the machine by means of a step that includes the application of one of the functions $f \in \Phi$.

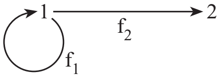

Fig. 1. State transition diagram for the X-machine with $X_0(\epsilon) = \{a^n | n \geqslant 0\}$.

Definition 2.3. For any *stream X-machine with ϵ-transitions* X, configuration (m, q, s, g) and function $f \in \Phi$, f *emerges from* q, or *is applied in* q, if $F(q, f) \neq \emptyset$. Also, if f emerges from q and $q' \in F(q, f)$, then f *reaches* q'.

Note. The above definition implies that F must be a total function, which evaluates to $\emptyset$ for any pair (q, f) in which the machine X does not have a possible next state.

Definition 2.4. A *transition* of a stream X-machine with ϵ-transitions is a change of configuration from x to x', denoted by $x \vdash x'$, where (m, q, s, g) and (m', q', s', g') are such that:

- $s = \sigma s'$, $\sigma \in \Sigma^{\epsilon}$;
- there is a function $f \in \Phi$ which emerges from q and reaches q'; and
- for this function f, $f(m, \sigma) = (\gamma, m')$, where $g' = g\gamma$, $\gamma \in \Gamma^{\epsilon}$.

If, in some configuration, there are several functions f which can be applied, then one of them is chosen randomly. Alternatively, a configuration may be reached where there is no function f that can be applied, and here there are at least two possible cases. One case, which corresponds to the behaviour that would normally be expected for such a machine, is that $q \in T$ (the set of final states). In this case the machine is said to *terminate*. The other case is that $q \notin T$, although if the machine is well behaved (in some sense that will not be discussed further here) then this case should not arise. If it does occur, then the machine is said to *block*, and this indicates that it has not carried out its computation properly.

Where a computation is carried out properly, then the output that it has computed can be defined. For this purpose, $\overset{\star}{\vdash}$ is used to denote the reflexive and transitive closure of $\vdash$.

Definition 2.5. The *output corresponding to an input sequence*. For any $s \in \Sigma^{\star}$, the output corresponding to this input sequence, computed by the stream X-machine with ϵ-transitions 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 2.1. Let us consider the stream X-machine X_0 with ϵ-transitions for which $\Sigma = \emptyset$, $\Gamma = \{a\}$, $Q = \{1, 2\}$, $M = \{m^0\}$, $\Phi = \{f_1, f_2\}$, $F(1, f_1) = \{1\}$, $F(1, f_2) = \{2\}$, $I = \{1\}$, $T = \{2\}$, and $f_1(m^0, \epsilon) = (a, m^0)$, $f_2(m^0, \epsilon) = (\epsilon, m^0)$. The diagram corresponding to X_0 is presented in Fig. 1.

Then $X_0(\epsilon) = \{a^n | n \geqslant 0\}$.

Note. Stream X-machines with ϵ-transitions are more general than stream X-machines. Indeed, there is no stream X-machine X and no input sequence s so that $X(s) = \{a^n | n \geqslant 0\}$ This does raise the issue as to whether it is valid to describe these more general machines as possessing the stream property, but discussion of this is beyond the scope of this paper. Hence, since only stream X-machines with ϵ-transitions are considered here, we will refer to them, for short, as X-machines.

Definition 2.6. *Communicating stream X-machine systems.* A communicating stream X-machine system (*CSXMS* for short) is a system: $S_n = ((X_i)_{i=1,\dots,n}, \mathscr{CM}_n, C^0)$, where:

- $X_i = (\Sigma_i, \Gamma_i, Q_i, M_i \times \mathscr{CM}_n, \Phi_i, F_i, I_i, T_i, m_i^0)$ are X-machines;
- $M = M_1 \cup \ldots \cup M_n$ is the set of memory values and $\widetilde{M} = M \cup SS$ is the set of (general) values, where SS is a set of special strings of symbols different from those in M. SS will be described later;
- $\mathscr{CM}_n$ is the set of all matrices of order $n \times n$ with elements in $\widetilde{M}$. This set defines the possible values for the global memory of the system, which is used for communication between the component X-machines, and so is referred to as the *communication matrix*. The elements of this matrix are therefore the values that are being communicated between the components of the system;
- $C^0 \in \mathscr{CM}_n$ is the initial communication matrix;
- for any i and $f \in \Phi_i$, $f : M_i \times \mathscr{CM}_n \times \Sigma_i^{\epsilon} \to \Gamma_i^{\epsilon} \times M_i \times \mathscr{CM}_n$.

Let $\Gamma = \Gamma_1 \cup \ldots \cup \Gamma_n$. A common output tape O is used by all components, and this contains sequences $g \in \Gamma^\star$.

As with an individual X-machine, the state of the global memory together with the configurations of the individual machines forms a tuple which is referred to as the configuration of the system.

Definition 2.7. A *configuration* of a CSXMS system S_n is a tuple: $z = (z_1, \ldots, z_n, C)$ where:

- $z_i = (m_i, q_i, s_i, g_i)$, $i = 1, \ldots, n$ is referred to as the *local configuration* of the machine X_i;
- the tuple $((m_i, C), q_i, s_i, g_i)$, $i = 1, \ldots, n$ is the actual configuration of the machine X_i, so that:
  - $m_i \in M_i$ is the local memory of X_i,
  - $q_i \in Q_i$ is the state of X_i,
  - $s_i \in \Sigma_i^\star$ is the current input sequence of X_i, and
  - $g_i \in \Gamma_i^\star$ is the current output sequence of X_i, the elements of which are interleaved onto the common output tape;

  and
- $C \in \mathscr{CM}_n$ is the communication matrix of the system.

For practical applications of these systems we will suppose that at any time at most one X-machine can write on the common output tape, i.e. the writing operations are serialised. At first glance, this may seem to be a prohibitive and unnatural restriction, but the serialisation problem will be solved in Section 4, after introducing channels as a mechanism for intercommunication between X-machines.

Also, in the treatment of the definition above in [BCG99], an important assumption that was made is that reading from and writing to the elements of the communication matrix in the global memory are done under mutual exclusion, although no mechanism was presented there for ensuring that this assumption was valid. For the moment this assumption will be made here too, but the way in which this mutual exclusion can be guaranteed will then be described in the next section, after the basic communication mechanism has been described here.

The standard set of special strings of symbols, as introduced in [BCG99], is $SS = \{\lambda, @\}$, where the meaning of these symbols is described in the following paragraph. The actual messages passed from one X-machine to another cannot be one of the strings in SS. Additional special strings of symbols will need to be introduced later on.

For each pair (i, j) with $i, j \in \{1, \ldots, n\}$, $i \neq j$, C_{ij} is used as a temporary buffer for passing 'messages' from the X-machine X_i to the X-machine X_j. Initially, C_{ij}^0 is one of the values λ or @, depending on whether or not messages are to be passed from X_i to X_j. For all i, $C_{ii}^0 = @$ because an X-machine never passes a message to itself. During the operation of the system an element C_{ij} may receive the value @, meaning that the connection from X_i to X_j is then disabled. A disabled connection cannot be enabled later.

We will denote by $+_i$ the set of all elements in the ith column and ith row of the communication matrix C. The significance of this is that, while in principle the whole of the matrix C forms part of the memory of each X-machine X_i, in practice X_i is only permitted to access (read from or write into) the elements of $+_i$.

Note. For the sake of simplicity, in the above description we suppose that all messages passed from any X-machine to any other one will 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 message. The receiver could use this flag in order to correctly decode the message.

In each X-machine X_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 x (if it is a processing state), or as $\underline{x}$ (if it is a communicating state). The final states are processing states; there is no function emerging from them.

Let $\underline{x}$ be a communicating state of the X-machine X_i and let $f_1, \ldots, f_k \in \Phi_i$ be the functions emerging from it, where any function f_s, $s \in \{1, \ldots, k\}$, is a *communicating function*, which may have one of the following forms and meanings:

1. a memory value *val* is assigned to C_{ij}, for some $j \neq i$:
   if *condition*$_s$ **then** $C_{ij} \leftarrow val$
   where *condition*$_s$ depends on the current value of m_i and $+_i$;

2. a value is moved from C_{ji} to some variable v of m_i, for some $j \neq i$:
   if *condition*$_s$ **then** $v \leftarrow C_{ji}$, $C_{ji} \leftarrow \lambda$
   where *condition*$_s$ depends on the current value of m_i and that of $+_i$;
3. under some condition, some elements of $+_i$ are modified:
   if *condition*$_s$ **then** *modify* $+_i$,
   where *condition*$_s$ involves elements in the domain of f_s and the modifications consist of changing the values of some elements of $+_i$ into one of the special symbols in SS.

Note. For communicating functions (emerging from a communicating state), the elements of $+_i$ may be both observed and changed; the local memory m_i may only be observed, with one exception, which is when a value of an element of $+_i$ is assigned to some variable v of m_i.

If more than one of the communicating 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 its configuration).

Let now x be a processing state, which is not a final one, of the X-machine X_i and let $f_1, \ldots, f_k \in \Phi_i$ be the processing functions emerging from it. Then any function f_s depends only on the local memory and on the local input tape and is meant to (partially) change the current value of m_i and possibly add some information to the output tape O. None of these functions alter their local output tapes, so that these are not used at all by the system.

As with the individual X-machines, if more than one of the processing 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. In this case the system will not have performed its computation properly, and so the content of the output tape is not significant.

When an X-machine X_i moves to a final state, all elements in $+_i$ have to change their values into @. Thus, in a system which is well behaved, a transition by a component machine to a final state will usually be associated with a communicating function of the third form defined above.

A CSXMS starts with all X-machines in their initial states, $C = C^0$ and $m_i = m_i^0$ for all $i \in \{1, \ldots, n\}$. Thus, the *initial configuration* of the system is $z^0 = (z_1^0, \ldots, z_n^0, C^0)$, where $z_i^0 = (m_i^0, q_i^0, s_i^0, \epsilon)$ with $q_i^0 \in I_i$.

The component X-machines then act simultaneously, which means that at any given time some machines may be in specific configurations while others are in the course of one configuration to another. The system will stop successfully when all X-machines reach final states, at which point all values in C should be @.

Definition 2.8. A *configuration* of a *CSXMS* is a *final* one if $z_i = (m_i, q_i, s_i, g_i)$ for all $i = 1, \ldots, n$, with $s_i = \epsilon$ and $q_i \in T_i$.

We can think about a change of configuration of the system, denoted $z \models z'$, as follows: let t be the time when the system reached the configuration z and t' the closest following moment of time at which a component terminates the execution of a function; then z' is the configuration of the system at time t'. Of course, it is possible that several components terminate the execution of a function at the same time t'. A change of configuration:

$$z = (z_1, \ldots, z_n, C) \models z' = (z_1', \ldots, z_n', C') \tag{1}$$

with $z_i = (m_i, q_i, s_i, g_i)$, $z_i' = (m_i', q_i', s_i', g_i')$, $s_i = \sigma_i s_i'$, $\sigma_i \in \Sigma_i^\epsilon$, $g_i' = g_i \gamma_i$, $\gamma_i \in \Gamma_i^\epsilon$ for any i, may be described as follows. For i taking the values $1, 2, \ldots, n$ in this order, there are two possibilities:

- either $z_i = z_i'$; or
- there exists a function $f \in \Phi_i$ emerging from q_i and reaching q_i', $q_i' \in F_i(q_i, f)$ and there exists a $C' \in \mathscr{CM}_n$, such that $f(m_i, C, \sigma_i) = (\gamma_i, m_i', C')$.

Note. For a component machine X_i where $z_i = z_i'$, this does not necessarily mean that this component has done nothing, but rather that it has not entirely completed executing a function. These partial actions do not influence the completed ones since the memories m_i are local to the components X_i and, according to the assumption above, the access to each element C_{ij} of the current matrix C is done under mutual exclusion.

Let $\overset{\star}{\models}$ be the reflexive and transitive closure of $\models$. Then the output computed by a CSXMS, if it terminates properly, can be defined as follows.

```
select
    $alt_1$
or $alt_2$
    ......
or $alt_k$
[or terminate]
end;
```

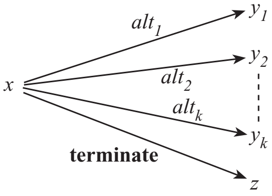

Fig. 2. The **select** construct for communicating states.

Definition 2.9. The *output of a CSXMS corresponding to an input sequence*. For any $s = (s_1, \ldots, s_n) \in \Sigma_1^\star \times \ldots \times \Sigma_n^\star$, the output corresponding to these input sequences, computed by the system S_n, is defined as: $S_n(s) = \{g = (g_1, \ldots, g_n) \in \Gamma_1^\star \times \ldots \times \Gamma_n^\star \mid \exists$ an initial configuration z^0 and a final one z, where $z^0 \overset{\star}{\models} z$, with $z = (z_1, \ldots, z_n, C)$, $C \in \mathscr{CM}_n$ and $z_i = (m_i, q_i, \epsilon, g_i)$ for all $i = 1, \ldots, n\}$.

3. Communicating X-Machine Systems Using Channels

The mechanism introduced above assures only a low level of synchronisation. In this section we will introduce channels as a higher level of synchronisation. The mechanism resembles that found in Occam [Inm84] and the formalism CSP [Hoa85, BuD93]. The *CSXM* 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 synchronised. 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.

Our aim is to implement communication using channels between the components of a *CSXMS*. It will be shown that the assumption in the previous section (about mutual exclusion when accessing the elements of C) is no longer needed, since this kind of communication assures mutual exclusion.

For this aim, we will introduce in each communicating state of each X-machine X_i the classical **select** construct with guarded alternatives and **terminate** clause, as presented in Fig. 2. The alternatives alt_s, $s \in \{1, \ldots, k\}$, should have the following forms:

1. [**when** $cond_k$ =>] $j \,!\, val$
2. [**when** $cond_k$ =>] $j \,?\, v$

with the following meanings:

1. if $cond_k$ is fulfilled, then val has to be sent to the X-machine X_j (via C_{ij});
2. if $cond_k$ is fulfilled, then v of M_i has to receive a value from the X-machine X_j (via C_{ji}).

The alternatives are *macrofunctions* described below; val is a memory value, $cond_s$ depends only on the local memory M_i and the local input tape, and v is a variable of M_i. As usual, the square brackets show that the information they include is optional.

The **terminate** clause acts as follows: if the other alternatives in the **select** construct are false and will be false forever, then the X-machine stops. In other words, if present, the **terminate** clause applies when all X-machines X_j to/from which X_i tries to send/receive messages had stopped. Executing **terminate** implies moving into a final state.

In the following, the form of functions emerging from a communicating state can be only as in Fig. 2. In order to implement the **select** construct in the communicating states, we will proceed as follows:

1. New strings are added to SS: $SS = \{\lambda, @, \$, \mathrm{OK}\} \cup \bigcup_{j=1}^{n} \{(j, \mathrm{S}), (j, \mathrm{R}), (j, \overline{\mathrm{S}}), (j, \overline{\mathrm{R}})\}$; the meaning of these new elements will be described below.
2. An additional X-machine named X_{n+1}, also called *Server*, is introduced. It acts simultaneously with $X_1, X_2, \ldots, X_n$. Of course, now the communication matrix is of order $(n+1) \times (n+1)$. No actual messages

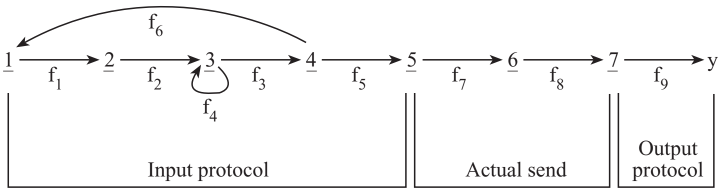

Fig. 3. The *send* macrofunction: [**when** *cond* =>] $j\,!\,val$.

(memory values) are sent to *Server* or received by *Server*, i.e. the value of the elements of $+_{n+1}$ are always in SS.

3. Let $s = n + 1$ and let d be the sth column of C; so $d_i = C_{is}$, $i \in \{1,...,n + 1\}$. During execution, the following values may be assigned to d_i:
   - @ (when we want X_i to stop); then @ has to be assigned to all elements in $+_i$ and a final state has to be reached;
   - (j, S), meaning that X_i asks *Server* for authorisation for sending a message to X_j;
   - (j, R), meaning that X_i asks *Server* for authorisation for receiving a message from X_j;
   - OK, if *Server* authorises a send/receive operation;
   - $(j, \overline{\mathrm{S}})$, if *Server* rejects an attempt of X_i to send a message to X_j, due to the fact that X_j is not (yet) ready to receive a message from X_i;
   - $(j, \overline{\mathrm{R}})$, if *Server* rejects an attempt of X_i to receive a message from X_j, due to the fact that X_j is not (yet) ready to send a message to X_i.

In this way, a *CSXMS* (*using channels*) is a system: $S_{n+1} = ((X_i)_{i=1,...,n+1}, \mathscr{CM}_{n+1}, C^0)$ where $X_{n+1} = Server$.

Each X-machine X_i will use $d_i = C_{is}$, with $i \neq n + 1$, only:

- in the input and output protocols associated to the send and receive operations;
- when moving to a final state; in this case, d_i has to receive the value @. As mentioned before, afterwards X_i has to assign @ to $+_i$ (in order to disable the connections between X_i and the other X-machines) and stop.

The local memory $L = M_{n+1}$ of *Server* stores the set of those values $i \in \{1, \dots n\}$ for which $i \in L$ iff $d_i \neq$ @. Initially it contains all i with $d_i = \lambda$. While L is not empty, an item of L is chosen. When L becomes empty, *Server* stops. For selecting an item from L we can do a random choice, we can organise L as a list etc.

When *Server* considers the value i, it acts as follows, where the symbol '—' stands for no action:

```
case $d_i$ of
    @                :   delete $i$ from $L$;
    $d_i = (j, S)$   :   if $d_j = (i, R)$ then $d_i \leftarrow$ OK; $d_j \leftarrow$ OK
                                      else if $d_j = (i, \overline{R})$
                                              then —
                                              else $d_i \leftarrow (j, \overline{S})$
    $d_i = (j, R)$   :   if $d_j = (i, S)$ then $d_i \leftarrow$ OK; $d_j \leftarrow$ OK
                                      else if $d_j = (i, \overline{S})$
                                              then —
                                              else $d_i \leftarrow (j, \overline{R})$
    else             :   —
end
```

Figures 3 and 4 describe the *send* and *receive* alternatives. Each of them is divided into three parts: the input protocol, the actual send/receive part and the output protocol. Note that all the intermediate states in these protocols are communicating states, since for each $i \in \{1,...,n\}$, d_i is an element of $+_i$.

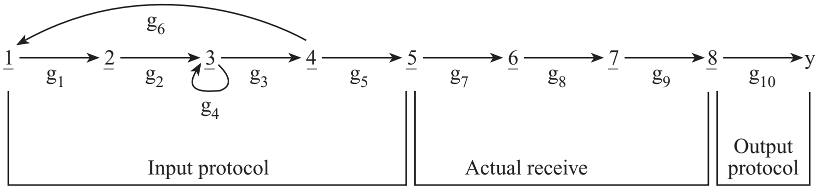

Fig. 4. The *receive* macrofunction: [**when** *cond* =>] $j\,?v$.

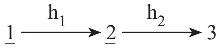

Fig. 5. The **terminate** macrofunction.

f_1: **if** $cond \,\&\, C_{ij} \neq @$
then —

f_2: $d_i \leftarrow (j, \mathrm{S})$

f_3: **if** $d_i \neq (j, \mathrm{S})$
then —

f_4: **if** $d_i = (j, \mathrm{S})$
then —

f_5: **if** $d_i = \mathrm{OK}$
then —

f_6: **if** $d_i \neq \mathrm{OK}$
then —

f_7: $C_{ij} \leftarrow val$

f_8: **if** $C_{ji} = \$$
then $C_{ji} \leftarrow \lambda$

f_9: $d_i \leftarrow \lambda$

g_1: **if** $cond \,\&\, C_{ji} \neq @$
then —

g_2: $d_i \leftarrow (j, \mathrm{R})$

g_3: **if** $d_i \neq (j, \mathrm{R})$
then —

g_4: **if** $d_i = (j, \mathrm{R})$
then —

g_5: **if** $d_i = \mathrm{OK}$
then —

g_6: **if** $d_i \neq \mathrm{OK}$
then —

g_7: **if** $C_{ji} \neq \lambda$
$v \leftarrow C_{ji}; C_{ji} \leftarrow \lambda$

g_8: $C_{ij} \leftarrow \$$

g_9: **if** $C_{ij} = \lambda$ **or** $C_{ij} = @$
then —

g_{10}: $d_i \leftarrow \lambda$

Figure 5 describes the **terminate** alternative, where 3 is a final state.

h_1: **if** for all j appearing in the other alternatives, $d_j = @$
then —

h_2: $d_i \leftarrow @; +_i \leftarrow @$

Note 3.1. When working with d_i, the following assertions are true:

d1. initially d_i is @ or λ;

d2. after receiving the value @ (when X_i stops), d_i can no longer be modified;

d3. d_i may be modified only by:

  d3.1. the X-machine X_i, in the input protocols (in order to ask for permission to send or receive a message) and in the output protocols (where it receives the value λ);

  d3.2. *Server*, when it receives a request from X_i for a communication with some X_j. Let us suppose that X_i decides to send a message (the case when X_i decides to receive a message is treated in a similar way). X_i is in state $\underline{3}$ and $d_i = (j, \mathrm{S})$. Three cases may occur:

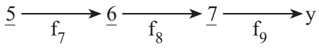

Fig. 6. Actual send and output protocol.

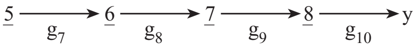

Fig. 7. Actual receive and output protocol.

- $d_j = (i, \mathrm{R})$: *Server* authorises this communication, it assigns OK to both d_i and d_j, and both X-machines may start their actual send/receive parts by reaching state $\underline{5}$;
- $d_j = (i, \overline{\mathrm{R}})$, i.e. X_j had previously tried to receive a message from X_i but was refused by *Server*, and so returns to its state $\underline{1}$: X_i will remain in state $\underline{3}$ until d_j gets a new value (through the action of X_j); this new value will be obviously different from $(i, \overline{\mathrm{R}})$ so one of the other cases will now apply, and X_i will actually leave state $\underline{3}$;
- $d_j \neq (i, \overline{\mathrm{R}})$ and $d_j \neq (i, \mathrm{R})$: X_i will return to state $\underline{1}$ with $d_i = (j, \overline{\mathrm{S}})$.

d3.3. *Server*, when it receives from some X-machine X_j a request for communication with X_i and authorizes it as in d3.2; then it assigns OK to d_i;

d4. when X_i reaches state $\underline{4}$ in order to try to send or receive a message from/to X_j, $d_i \in \{\mathrm{OK}, (j, \overline{\mathrm{S}}), (j, \overline{\mathrm{R}})\}$;
d5. when entering the actual send/receive part, $d_i = \mathrm{OK}$ and its value is not changed until reaching the output protocol;
d6. after executing the output protocol, $d_i = \lambda$.

Note 3.2. When working with C_{ij}, $1 \leqslant i, j \leqslant n$, the following assertions are true:

c1. initially C_{ij} is @ or λ;
c2. after receiving the value @ (when X_i or X_j assigns this value to it, due to the fact that one of the X-machines X_i or X_j stops), C_{ij} can no longer take another value;
c3. when not in an actual send/receive, $C_{ij} \in \{@, \lambda\}$ and C_{ij} can be modified only if X_i or X_j assigns to it the value @;
c4. when entering an actual send/receive for a communication between X_i and X_j, $C_{ij} = \lambda$, as well as C_{ji};
c5. when entering the output protocols of the send/receive actions, $C_{ij} = C_{ji} = \lambda$;
c6. when the execution of the system terminates normally, all elements of C are @.

Theorem 3.1. The implementation of **select** constructs in the communicating states is correct.

Proof. We will divide the proof into two parts.

Correctness of the message passing:

We are interested in the correctness of sending a message from X_i to X_j, $1 \leqslant i, j \leqslant n$. Let us suppose that *Server* has authorised this transmission by assigning to d_i and d_j the value OK. From c4 it follows that $C_{ij} = C_{ji} = \lambda$. Now X_i has to perform the actions in Fig. 6:

f_7: $C_{ij} \leftarrow val$

f_8: **if** $C_{ji} = \$$
  then $C_{ji} \leftarrow \lambda$

f_9: $d_i \leftarrow \lambda$

while X_j has to perform the actions in Fig. 7:

g_7: **if** $C_{ij} \neq \lambda$
  then $v \leftarrow C_{ij}\ ; C_{ij} \leftarrow \lambda$

g_8: $C_{ji} \leftarrow \$$

g_9: **if** $C_{ji} = \lambda$ **or** $C_{ji} = @$
  then –

g_{10}: $d_j \leftarrow \lambda$

At the beginning $C_{ij} = \lambda$, and so the conditions in g_7 and f_8 mean that the order of execution is compulsory: f_7, g_7, g_8, f_8. This interleaving shows that the order in which the assignments: $d_i \leftarrow$ OK and $d_j \leftarrow$ OK are done by the X-machine *Server* is not relevant. At the same time, it is clear that the message passing $v \leftarrow val$ is achieved. Both machines will now have entered their output protocols.

Three cases may occur:

1. X_i and X_j terminate their output protocols simultaneously. In this case, no problem arises, as the actions are disjoint.
2. X_i executes its output protocol first. Then $d_i = \lambda$, $d_j =$ OK, $C_{ij} = C_{ji} = \lambda$. If an X-machine (including X_i) tries to communicate with X_j, it will fail (that is, the *Server* will refuse the request), since $d_j =$ OK. It is important to note that now only X_i may modify C_{ji}, namely by assigning to it the value @. This is possible if X_i acts 'very quickly', assigns successively λ and @ to d_i and afterwards assigns @ to C_{ji}. But the condition '**or** $C_{ji} =$ @' in X_j prevents X_j to be blocked in state $\underline{7}$.
3. X_j executes its output protocol first. Then $d_j = \lambda$, $d_i =$ OK, $C_{ij} \in \{\lambda, @\}$ and $C_{ji} = \lambda$. If an X-machine (including X_j) tries to communicate with X_i, it will fail, since $d_i =$ OK.

Correct handling of the common matrix C by the client components X_i, $i \neq n+1$:

We already proved that during the actual message passing the handling of the matrix C is done correctly. We have still to consider the way in which the X-machines perform the actions prior to stopping (reaching a final state). Let us suppose that X_i tries to perform the actions: $d_i \leftarrow @$; $+_i \leftarrow @$. The following situations can occur:

- All X-machines X_j communicating with X_i have already stopped. In this situation the change of $+_i$ will not affect the other X-machines.
- There is at least one process X_j which expresses its intention to communicate with X_i (i.e. X_j is in the state $\underline{1}$) and X_i has not yet assigned @ to C_{ij} or C_{ji}. Then X_j will reach state $\underline{2}$, it will assign (i, R) or (i, S) to d_j, and will reach state $\underline{3}$. X_j will remain in state $\underline{3}$ until *Server* considers j. Since $d_i \in \{@, \lambda\}$, d_j will be set to $(i, \overline{\mathrm{R}})$ or $(i, \overline{\mathrm{S}})$ and X_j will again reach state $\underline{1}$. When C_{ij} and C_{ji} are or become both @, X_j will not try again to communicate with X_i.
- There is at least one process X_j, which did not complete its action of receiving the message from X_i. As we mentioned that the sequence f_7, g_7, g_8, f_8 is compulsory, it follows that X_j can only be in state $\underline{7}$ (in Fig. 4). If X_i has not already assigned @ to C_{ji}, then $C_{ji} = \lambda$ (from the proof of the transmission's correctness) so X_j will complete the receiving action. If X_i has assigned @ to C_{ji}, X_j will reach state $\underline{8}$ (according to g_9) and then will also complete the receiving action.

Note 3.3. From Note 3.2, it follows that when $d_i = (j, \mathrm{S})$ and $d_j = (i, \overline{\mathrm{R}})$ or $d_i = (j, \mathrm{R})$ and $d_j = (i, \overline{\mathrm{S}})$, then X_i will remain in state $\underline{3}$; in this way, when two X-machines try to communicate, livelock is avoided.

Note 3.4. It is possible for X_i and X_j to try both to send and receive a message from the other one. The direction in which the transmission will actually be done will then depend on both the item chosen by *Server* from its local memory L and the alternatives chosen in each of the two X-machines. □

4. Writing on the Output Tape under Mutual Exclusion

In Section 2, we supposed that the writing operations on the output tape are serialised. Now we can show how this may actually be achieved.

An additional X-machine X_{n+2} with the alias *Output* is added to the system. The system now has the form: $S_{n+2} = ((X_i)_{i=1,\dots,n+2}, \mathcal{CM}_{n+2}, C^0)$ and X_{n+2} is the only X-machine which alters its output tape (see Definition 2.6).

Writing to the output tape is achieved by sending a message to the X-machine *Output*. Now C is a square matrix of order $(n+2) \times (n+2)$. The local memory M_{n+2} of *Output* is a variable x. The output tape of *Output* is initially void. $I_{n+2} = \{\underline{1}\}$ and $T_{n+2} = \{3\}$. The X-machine *Output* repeatedly tries to receive an item of data from any of the X-machines $X_1, X_2, \dots, X_n$ and to add it to its tape. When all these X-machines have stopped, *Output* stops too (via the **terminate** clause). The state transition diagram for *Output* is shown in Fig. 8.

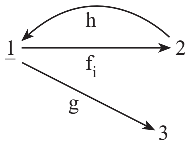

Fig. 8. The state transition diagram for Output.

f_i: $i\,?\,x$ for all $i = 1, 2, \ldots, n$

g: **terminate** h: add x to the output tape

5. Examples

Example 5.1. *The Producer–Consumer problem with bounded queue.*

A producer produces items and places them into a buffer. The consumer takes items from the buffer and consumes them. Let *max* be the size of the buffer. 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 there, i.e. the buffer is a queue;
- reading from an empty buffer and writing in a full buffer must be avoided.

We will suppose that these items are lowercase letters. The producer stops after sending the first letter z, and the consumer stops after receiving the first z. The output tape has to contain the characters produced, as well as the characters consumed; the last ones will appear in the uppercase form.

According to the previous section, we will model the problem with a CSXMS with five components. The initial form C^0 of the communication matrix C is:

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

X_5 is the alias for the X-machine *Output* (the X-machine that achieves writing on the output tape under mutual exclusion), while X_4 is the alias for *Server*.

X_1 corresponds to the producer. M_1 contains a variable *ch*. The input tape L_1 contains the items that the producer intends to place in the queue; $z \in L_1$. We have $I_1 = \{1\}$ and $T_1 = \{4\}$. The state transition diagram for X_1 appears in Fig. 9(a).

f_1: $ch \leftarrow \text{first}(L_1)$
$L_1 \leftarrow \text{tail}(L_1)$

f_2: $2\,!\,ch$

f_3: **if** $ch = z$ **then** $d_1 \leftarrow @$

f_4: **if** $ch \neq z$ **then** $-$

f_5: $5\,!\,ch$

X_3 models the activities of the consumer. $m_3^0 = \emptyset$, $I_3 = \{\underline{1}\}$ and $T_3 = \{4\}$. The state transition diagram is shown in Fig. 9(c).

h_1: $2\,?\,ch$

h_2: $5\,!\,\text{uppercase}(ch)$

h_3: **if** $ch = z$ **then** $d_3 \leftarrow @$

h_4: **if** $ch \neq z$ **then** $-$

The X-machine X_2 implements the activities concerning the buffer. The local memory M_2 contains a variable

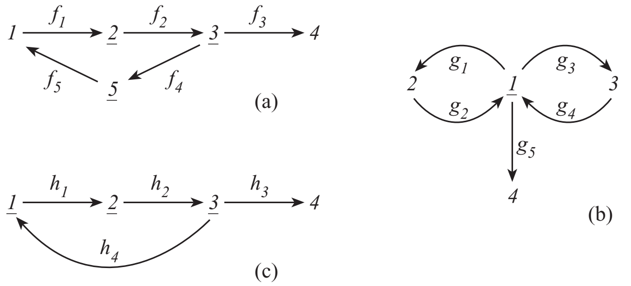

Fig. 9. The Producer–Consumer problem with bounded queue. The state transition diagrams for: (a) X_1; (b) X_2; (c) X_3.

ch and a queue *Q*, the maximum size *max* of *Q* and the current size *len* of *Q*. Initially the queue is empty, so that $len = 0$. We have $I_2 = \{\underline{1}\}$ and $T_2 = \{4\}$. The state transition diagram appears in Fig. 9(b).

g_1: **when** $len < max$ => $1\,?ch$ g_2: $ch \Rightarrow Q$; $len \leftarrow len + 1$

g_3: **when** $len > 0$ => $3\,!\,\mathrm{first}(Q)$ g_4: $Q \leftarrow \mathrm{tail}(Q)$; $len \leftarrow len - 1$

g_5: **terminate**

where '$\Rightarrow$' is the operator used for adding an item to the queue.

Example 5.2. *Finding the first n prime numbers.*

We will introduce a CSXM system with $n + 4$ components. The X-machines $X_0, X_1, \ldots, X_{n+1}$ form a pipeline structure. The X-machine X_0 pumps the numbers $2, 3, \ldots$ to X_1. For $i = 1, \ldots, n$, the X-machine X_i does the following: the first number it receives from X_{i-1} is stored as a witness value and added to the 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 X_{i+1} (for further checking), otherwise it is discarded. Obviously, the witness values of $X_1, \ldots, X_n$ are the first *n* prime numbers. The X-machine X_{n+1} receives numbers from X_n and discards them. A mechanism for proper termination of the system has to be introduced. For this purpose the role of X_{n+1} is changed: after it receives a value, it stops. In this way, by introducing the **terminate** clause in the other X-machines, they will stop in the order: $X_n, X_{n-1}, \ldots, X_0$. *Server* and *Output* are the aliases for X_{n+2} and X_{n+3}.

The initial form C^0 of the communication matrix *C* is:

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

For the X-machine X_0, M_0 contains only a variable *i* initialised with 2. $I_0 = \{\underline{1}\}$ and $T_0 = \{3\}$. The state transition diagram appears in Fig. 10(a), where:

f_1: $1\,!\,i$ f_2: $i \leftarrow i + 1$ f_3: **terminate**

For each $i = 1, \ldots, n$, the internal memory M_i of the X-machine X_i contains two variables *x* (the witness value) and *y*. $I_i = \{\underline{1}\}$ and $T_i = \{6\}$. The state transition diagram is shown in Fig. 10(b), where:

g_1: $i - 1\,?x$ g_2: $n + 3\,!\,x$ g_3: $i - 1\,?y$

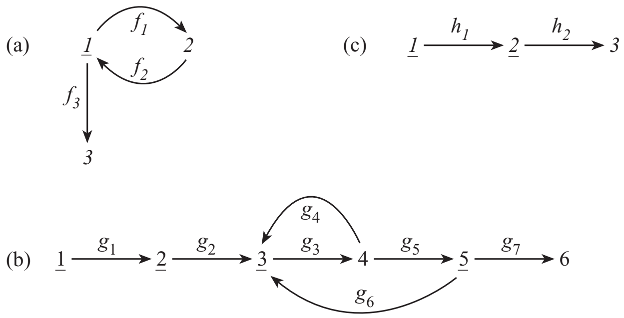

Fig. 10. Finding the first n prime numbers. The state transition diagrams for: (a) X_0; (b) X_i, $i = 1, \dots, n$; (c) X_{n+1}.

g_4: **if** $y \bmod x = 0$ **then** $-$ $\qquad$ g_5: **if** $y \bmod x \neq 0$ **then** $-$

g_6: $i + 1\, !\, y$ $\qquad$ g_7: **terminate**

The internal memory of the X-machine P_{n+1} contains a single variable x. $I_{n+1} = \{\underline{1}\}$ and $T_{n+1} = \{3\}$. The state transition diagram appears in Fig. 10(c), where:

h_1: $n\, ?\, x$ $\qquad$ h_2: $d_{n+1} \leftarrow @$

6. From CSXM Systems to Concurrent Programs

In this section, the functions emerging from a communicating state may have only the forms given in the previous sections. The way we modelled channel operations has the remarkable property that it allows an automatic scheme for writing concurrent programs in an Occam, Pascal-FC or Ada-like style [BuD93, Geo97].

For each X-machine we will write a process as follows. The local variables correspond to the local memory of the X-machine. For each non-final state x, a sequence of statements, labelled with x, will be produced:

1. If $\underline{x}$ is a communicating state, a **select** statement labelled with x will be provided. Each function emerging from x corresponds to an alternative:

```
x : select
        [when condition_1 =>]
         channel_operation_1; goto $y_1$
      ...
    or  [when condition_k =>]
         channel_operation_k; goto $y_k$
    [or terminate]
    end
```

   where the **terminate** alternative is present only if there is an (unique) function emerging from x and labelled with **terminate**.

2. If x is a processing state, the following sequence of statements will be used:

```
x: b:=false;
   repeat
     i:=random(k)+1 { k functions are emerging from x;
                       random(k) returns one of the values $0, \dots, k-1$}
     if $f_i$ may be applied then b:=true
   until b;
   apply $f_i$;
   choose randomly $y_i \in F(s, f_i)$; goto $y_i$
```

For the prime number generator described above in Example 5.2, the processes corresponding to the $n+4$ X-machines of the system are:

```
process $X_0$;
    var i:integer;
begin
   i:=2;
   1: select
          send i to $P_1$; goto 2
       or terminate
       end;
   2: i:=i+1; goto 1
end;

process $X_{n+1}$;
   var x:integer;
begin
   1: select
         receive x from $X_n$; goto 2
       end;
   2: select
          terminate
       end
end;
```

For the X-machines $X_1, \ldots, X_n$ the following process type may be written:

```
process type XType(i:integer);
   var x,y:integer; b:boolean;
begin
   1: select
         receive x from $X_{i-1}$; goto 2
      end;
   2: select
         send x to $X_{n+3}$; goto 3
      end;
   3: select
         receive y from $X_{i-1}$; goto 4
      end;
   4: b:=false;
       repeat
          i:=random(2)+1;
          if i=1 then b:=y mod x = 0
                 else b:=y mod x <> 0;
       until b;
       if i=1 then goto 3;
              else goto 5;
   5: select
         send y to $X_{i+1}$; goto 3
      or terminate
      end
end;
```

Of course, the X-machines X_{n+2} (alias *Server*) and X_{n+3} (alias *Output*) are also included in the system and are acting simultaneously with the ones considered above. We will describe only the process associated to X_{n+3}:

```
process X_{n+3};
   var x,len:integer;
      tape:array[1..100] of integer;
begin
   len:=0; { len is the initial length of the tape }
   1: select
         1 ? x; goto 2;
      or 2 ? x; goto 2;
      ...
      or n ? x; goto 2;
      or terminate
   end;
   2: len:=len+1; tape[len]:=x; goto 1
end;
```

7. Conclusions

The communicating X-machines models studied in [BCG99] and [GeV00] only provided a low level of synchronisation. Moreover, they worked under the assumption that the X-machines act under mutual exclusion which could not be guaranteed for many situations.

The $CSXMS$ proved to be a natural way for implementing communication between the components through channels. This paper deals mainly with the implementation of this mechanism. A new X-machine, the *Server*, is added to the system. Any communication between two X-machines is performed only after the *Server* allows it. The main operations *send*, *receive* and the *terminate* clause are implemented as macrofunctions, using the lower-level facilities of the basic $CSXMS$ model. The construct *select* with guarded alternatives is also modelled.

Controlling the communications within this system requires the *Server* and the macrofunctions to interact through the vector d which forms part of the communication matrix. These interactions do not require any new basic features to be added to the system and so the results obtained in [BCG99] for the computational power of $CSXMS$ are still applicable. Using this approach, the correctness of this implementation is proved in Theorem 3.1. This model of the system also assumes a unique output tape where the writing operations are serialised, and Section 4 describes exactly how this serialisation is obtained.

Finally, for any $CSXMS$, Section 6 describes the mechanism for automatic generation of the corresponding concurrent program, written in a Pascal-FC or Ada-like style. Of course, comparisons with these languages, where the implementation models do not need any equivalent of our *Server* process, raise the issue of whether a distributed solution could be developed here that did not require such a process. This issue requires further investigation, and here we simply observe that these languages use syntactic constraints to restrict the direction of communication through any given chanel, whereas our model cannot assume such restrictions, and so must cater for the possibility of any machine trying to use any of its channels for either sending or receiving.

Further work that can be developed directly from the results described here concerns two aspects. One is the use of this model for testing purposes, as compared with the less structured forms of $CSXMS$ that have been studied previously. The other is the implementation of alternative mechanisms that can be used for message passing, such as remote procedure invocation for example.

References

[BCG99] Bălănescu, T., Cowling, A. J., Georgescu, H., Gheorghe, M., Holcombe, M. and Vertan, C.: Communicating stream X-machines are no more than X-machines: *Journal of Universal Computer Science*, 5(9):494–507, 1999.

[BGG98a] Bălănescu, T., Georgescu, H. and Gheorghe, M.: Guarded additive valence grammars as models for synchronization problems. In *Annals of Bucharest University, Mathematics-Informatics series*, Vol. 47(1), 19–26, 1998.

[BGG98b] Bălănescu, T., Georgescu, H. and Gheorghe, M.: On counting derivation in grammar systems. *Romanian Journal of Information Science and Technology*, 1(1):23–42, 1998.

[BGG98c] Bălănescu, T., Georgescu, H. and Gheorghe, M.: Grammatical models for some process synchronizers. In *Proceedings of the MFCS'98 Satellite Workshop on Grammar Systems*, Silesian University, Brno, pp. 117–137, 1998.

[BGG01] Bălănescu, T., Georgescu, H. and Gheorghe M.: Stream X-machines with underlying distributed grammars. *Informatica* (submitted).
[BWW96] Barnard, J., Whitworth, J. and Woodward, M.: Communicating X-machines. *Journal of Information and Software Tehnology*, 38(6):401–407, 1996.
[BuD93] Burns, A. and Davies, G.: *Concurrent Programming*. Addison-Wesley, Reading, MA, 1993.
[Cho78] Chow, T. S.: Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering* 4(3):178–187, 1978.
[Eil74] Eilenberg, S.: *Automata, Languages and Machines*. Academic Press, New York, 1974.
[Geo97] Georgescu, H.: *Concurrent Programming*. Editura Tehnică, Bucharest, (in Romanian), 1997.
[GeV00] Georgescu, H. and Vertan, C.: A new approach to communicating X-machines systems. *Journal of Universal Computer Science*, 6(5):490–502, 2000.
[Hoa85] Hoare, C. A. R.: *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, NJ, 1985.
[Hol88] Holcombe, M.: X-machines as a basis for dynamic system specification. *Software Engineering Journal*, 3:69–76, 1988.
[HoI98] Holcombe, M. and Ipate, F.: *Correct Systems: Building a Business Process Solution*. Springer, Berlin, 1998.
[Inm84] INMOS Limited: *Occam Programming Manual*. Prentice-Hall, Englewood Cliffs, NJ, 1984.
[IpH96] Ipate, F. and Holcombe, M.: Another look at computability. *Informatica*, 20:359–372, 1996.
[IpH97] Ipate, F. and Holcombe, M.: An integration testing method that is proved to find all faults. *International Journal of Computer Mathematics*, 69:159–178, 1997.
[IpH01] Ipate, F. and Holcombe, M.: Generating test sequences from non-deterministic X-machines. *Formal Aspects of Computing*, 12(6):443–458, 2000.

Received December 1999
Accepted January 2001