STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **XLVI**, Number 1, 2001

AN EFFICIENT WAY TO MODEL P SYSTEMS BY X-MACHINE SYSTEMS

HORIA GEORGESCU

Abstract. Starting from the powerful concept of stream X-machine, the author proposed in some previous papers an original way to integrate more stream X-machines with ε-transitions into a system. The communication between the components is assured by means of a communication matrix, used as a common memory. It was proved that by introducing an additional component, it is possible to achieve communication in a structured way, namely by channels, with **select** constructs appearing in each communicating state. This model was used for proving that P systems may be modelled by communicating stream X-machine systems. Even if the proof is a constructive one, it is not satisfactory due to its low level of concurrency. In this paper a much more efficient construction, from the concurrency's point of view, is presented.

Keywords: *P systems, Communicating X-machine systems, concurrent processes, communication using channels.*

1. Introduction

The concept of *X-machines* was introduced by Eilenberg in [4] and used by Holcombe [7] as a possible specification language. Afterwards, a lot of research has been done in this field.

The new features which differentiate an X-machine from a finite-state one are: a set of processing functions Φ, an input and an output tape and a set X which characterizes the internal memory of the machine. The transitions between the states are performed according to these functions. The X-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. A new item may be added to the output tape after a transition takes place. In this way both the system data and the control structure are modelled, while allowing them to be separated.

2000 *Mathematics Subject Classification.* 68N19, 94A40.

1998 *CR Categories and Descriptors.* B.4.4 [**Hardware**]: Input/Output and Data Communication – *Performance Analysis and Design Aids*; D.1.3 [**Software**]: Programming Techniques – *Concurrent Programming*.

The research performed during the last years concerned mainly their generative power and their possible use for testing. Very little attention has been paid to the possible communication between these machines and consequently to their use for specification of concurrent processes.

In [2] cooperating distributed grammar systems are used for modelling their concurrent behaviour.

In [5] another approach was proposed, using a communication matrix for the communication between the X-machines. Using this model, in [1] it was proved that communicating stream X-machines systems are equivalent (from the generative power point of view) with a single X-machine.

A different approach from [5] is proposed in [3], where more powerful tools are added. The new mode allows the use of channels as the mechanism for passing messages between the X-machines. It also allows implementation of specific constructs for channels as **select** with an optional **terminate** clause. The main idea was to introduce a new X-machine, called *Server*, for controlling the communication between the components of the system; the last column (the one corresponding to the X-machine *Server*) of the communication matrix is used too. Following this model, an automatic scheme for writing concurrent programs (in Pascal-FC or Ada like style) was proposed.

Any non-trivial biological system is a hierarchical construct, made up of several *organs* which are well defined and delimited from their neighbouring organs. Each organ evolves internally, but also cooperates with neighbour organs in order to keep alive the system as a whole; cooperation consists in a flow of materials, energy and information, necessary for the functioning of the system.

A membrane structure is composed of *regions* delimited by *membranes*. A region is a space enclosed by membranes. neighbour regions communicate through the membranes separating them. The space outside the skin membrane is called the *outer region*.

P systems have been studied mainly from the point of view of their computational power and it was shown that their generative power is that of the Turing machines. The main results can be found in [8] and [9].

P systems were used also for solving NP-complete problems in polinomial time. For this purpose, one approach is to allow the number of membranes to grow dinamically (see [11]), while another approach is to count only the changes of configuration (see [10]). Unfortunately, in the first approach the number of membranes grows exponentially, while in the second approach the local time complexity is exponential too.

In [6] it was shown that P systems may be modelled by communicating stream X-machine systems.

In Section 2 and Section 3, the basic results concerning communicating stream X-machine systems (CSXMS) and P systems are reviewed. In section 4 it is presented a new construction for modelling P systems by CSXMS; this construction assures a high degree of concurrency.

2. Communicating Stream X-machine Systems

Definition 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:*
$$f : M \times \Sigma^\varepsilon \to \Gamma^\varepsilon \times M;$$
- 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 and an output tape. F must be a total function.

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 $(m^0, q^0, s^0, \varepsilon)$, where $q^0 \in I$ and $s^0 \in \Sigma^\star$ is the input sequence.

A *change of configuration*, denoted by $\vdash$: $(m, q, s, g) \vdash (m', q', s', g')$ is possible if:

- $s = \sigma s'$, $\sigma \in \Sigma^\varepsilon$;
- there is a function $f \in \Phi$ with $F(q, f) \neq \emptyset$ and $q' \in F(q, f)$ (in which case we say that *f may be applied, f emerges from q and reaches q'*), so that $f(m, \sigma) = (\gamma, m')$ and $g' = g\gamma$, $\gamma \in \Gamma^\varepsilon$.

The output corresponding to an input sequence $s \in \Sigma^\star$, is defined as:

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

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

Definition 2. *A Communicating Stream X-machine System (CSXMS for short) is a system*

$$S_n = ((X_i)_{i=1,\ldots,n}, \mathcal{CM}_n, C^0),$$

where:

- $X_i = (\Sigma_i, \Gamma_i, Q_i, M_i \times \mathcal{CM}_n, \Phi_i, F_i, I_i, T_i, m_i^0)$ *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.*
- *$\mathcal{CM}_n$ is the set of all matrices of order $n \times n$ with elements in $\widetilde{M}$. This set defines the possible values of the global memory of the system, which is used for communication between the component X-machines; therefore it is referred to as the communication matrix;*
- *C^0 is the initial communication matrix;*
- *for any i and $f \in \Phi_i$, $f : M_i \times \mathcal{CM}_n \times \Sigma_i^\varepsilon \to \Gamma_i^\varepsilon \times M_i \times \mathcal{CM}_n$.*

Let $\Gamma = \Gamma_1 \cup \ldots \cup \Gamma_n$. A common output tape O is used by all components. It is initially void and afterwards it contains sequences $g \in \Gamma^\star$.

We mention that each X-machine X_i can access (read from or write to) only $+_i$, where $+_i$ denotes the set of all elements in the ith line and ith column of the communication matrix. An element C_{ij} may receive the value @, meaning that the connection from X_i to X_j is disabled. A disabled connection can not be enabled later.

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*. The final states are processing states; there is no function emerging from them.

The functions emerging from a processing state depend only on the local memory and on the local input tape and are meant to (partially) change the local memory and possibly add some information to the output tape O.

The functions emerging from a communicating state depend on the local memory and on $+_i$ and are meant to move a value from the internal memory to the communication matrix and viceversa, as well to assign a special value to the communication matrix.

When an X-machine X_i moves to a final state, all elements in $+_i$ have to change their values into @.

A *configuration* of a CSXMS system S_n has the form: $z = (z_1, \ldots, z_n, C)$, where:

- $z_i = (m_i, q_i, s_i, g_i)$, $i = 1, \ldots, n$;
- m_i is the current value of the memory M_i of X_i;
- q_i is the current state of X_i;
- $s_i \in \Sigma_i^\star$ is the current input sequence of X_i;
- $g_i \in \Gamma_i^\star$ is the current output sequence of X_i;
- C is the current value of the communication matrix of the system.

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

We can think about a change of configuration $z \models z'$ as follows: let t be the time when the system reached the configuration z and t' the closest following moment

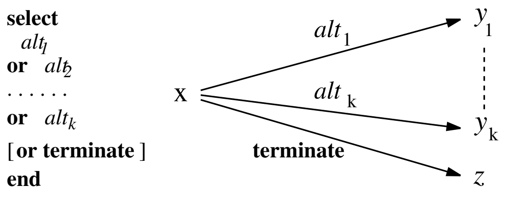

FIGURE 1. The **select** construct for communicating states

of time at which a component terminates the execution of a function; then z' is the configuration of the system at time t'.

A change of configuration:

$$z = (z_1, \ldots, z_n, C) \models z' = (z'_1, \ldots, z'_n, C') \tag{1}$$

with $z_i = (m_i, q_i, s_i, g_i)$, $z'_i = (m'_i, q'_i, s'_i, g'_i)$, $s_i = \sigma_i s'_i$, $\sigma_i \in \Sigma_i^\varepsilon$, $g'_i = g_i \gamma_i$, $\gamma_i \in \Gamma_i^\varepsilon$ for any i, may be described as follows. Let $C_0 = C$. 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 $q'_i \in F_i(q_i, f)$ and $C_i \in \mathcal{CM}_n$ so that $f(m_i, C_{i-1}, \sigma_i) = (\gamma_i, m'_i, C_i)$.

The X-machines act simultaneously. The system stops successfully when all X-machines reach final states (i.e. all values in C are @).

Let $\overset{\star}{\models}$ be the reflexive and transitive closure of $\models$. Then the *output computed by a CSXMS S_n corresponding to an input sequence s* can be defined as follows: $X(s) = \{g = (g_1, \ldots, g_n) \in \Gamma_1^\star \times \ldots \times \Gamma_n^\star \,|\exists z^0$ an initial configuration and z a final one, $z^0 \overset{\star}{\models} z$, with $z = (z_1, \ldots, z_n, C)$, $C \in \mathcal{CM}_n$ and $z_i = (m_i, q_i, \varepsilon, g_i)$ for any $i = 1, \ldots, n\}$.

The mechanism introduced above assures only a low level of synchronization. Therefore channels were intoduced as a higher level of synchronisation.

For this aim, in each communicating state of each X-machine X_i the classical **select** construct with guarded alternatives and **terminate** clause was introduced, as presented in Fig.1. 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});

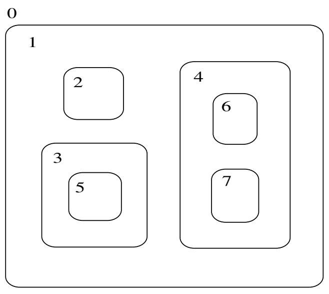

FIGURE 2. A membrane structure

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*; *val* is a memory value, $cond_s$ depends only on the local memory M_i and on 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 have stopped. Executing **terminate** implies moving to a final state.

In the following, the form of functions emerging from a communicating state can be only as in Fig. 1.

In [3] an implemetation of the **select** constructs was proposed and it was proved that this implementation is a correct one.

The above results are important due to the well known power of communication using channels. A first result appears in [3]: the authors present an automatic scheme for generating a concurrent program, written in an Pascal-FC or Ada like style, starting from an arbitrary CSXMS that uses in communicating states only **select** constructs. Another result appears in Section 4.

3. P SYSTEMS - MEMBRANE COMPUTING

Let us consider the membrane structure in Fig. 2, where the outer region has the label 0. Due to its intrinsic hierarchical type, a membrane structure may be represented in a paranthesized form. For the example in Fig. 2, the corresponding representation is: $[_1[_2]_2[_3[_5]_5]_3[_4[_6]_6[_7]_7]_4]_1$.

Membrane structures may be represented also as a special kind of trees, which we will call *P-trees*. A P-tree associated to a membrane structure is defined as follows:

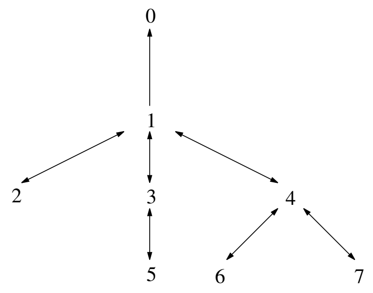

FIGURE 3. The tree associated to the membrane structure in Fig. 2

- each of the $n+1$ nodes in the tree corresponds to a region in the membrane structure; the nodes are labeled with $0, 1, ..., n$, where 0 is always the label of the outer region, and 1 is the label of the region just inside the skin membrane;
- for each node $i \in \{1, ..., n\}$, edges diverge towards its father and its sons, corresponding to the membranes which separate the respective region from the neighbour regions;
- no edge diverges from node 0.

The edges show the way in which information may be sent from one node (region) to another one.

The P-tree T associated to the membrane structure in Fig. 2 is shown in Fig. 3. To each node (region in the membrane structure) i we associate a finite languages M_i over an alphabet V. The elements of these languages are called *objects*. Since the sets of objects are changing while the system evolves, we denote by M_i^0 their initial content and by M_i their current content: $(M_0, ..., M_n)$ is the *current configuration* of the system.

Generally, a membrane structure is *dynamic*: a region may be *divided* into several regions or may be *dissolved*. In this paper we will consider only *static structures*, in which their forms are never changed, i.e. no region can be divided or dissolved.

Let R_i be the set of *evolution* (*developmental*, *replicated rewriting*) *rules* for each region $i = 1, 2, ..., n$. They have the form:
$a \to (\alpha_1, t_1) \parallel \ldots \parallel (\alpha_k, t_k)$
for some k, where $a \in V$ and $\alpha_1, ..., \alpha_k \in V^*$, while t_j, $j = 1, ..., k$ indicate the *targets* of the rule and can be only i, the father of i or one of its sons.

Such a rule can be applied to an object in M_i if the object has the form $o = \beta a \gamma$; if the rule is applied, o is erased and the new objects $\beta \alpha_j \gamma$ are created and added to M_{t_j} respectively.

A *computation* starts from the initial configuration $(M_0^0, M_1^0, ...M_n^0)$: these sets are initially associated to the nodes of the tree T.

At each step of the computation, the rules are applied to the objects in parallel; for each object one of the rules (if any) that can be applied to it is chosen randomly and the resulting objects are sent to their specified targets. All objects which do not evolve are passed unchanged to the next step.

A computation is *complete* if it halts: no rule can be used in the current configuration. In this case, the *output of the system* for this computation is M_0 (the language associated to the outer region).

We can now summarize the above disscussion concerning the (static) membrane structure as follows:

Definition 3. *A P system (membrane structure) is a construct*

$$\Pi = (V, \mu, (M_0^0, M_1^0, ..., M_n^0), R)$$

where:

- *μ is the membrane structure. Let us consider that it is represented as a P-tree with the nodes $0, 1, ..., n$, where $n+1$ is the number of the nodes in the tree. The root of the tree is 0 and its son is 1;*
- *V is an alphabet;*
- *$M_0^0, M_1^0, ..., M_n^0$ are the sets of objects initially associated to the nodes $0, 1, ..., n$;*
- *R is the finite set of evolution (replicated rewriting) rules.*

A change of configuration $(M_0, M_1, ..., M_n) \mapsto (M'_0, M'_1, ...M'_n)$ is performed by applying rules to the objects in $M_0, M_1, ..., M_n$ in parallel, as described above, and by sending the resulting objects to the specified targets.

The output of the system is the union L of all the languages M_0 in final configurations.

Remark 1. *Many variants of the above definitions may be considered. Some of them are presented below:*

- a P system (membrane structure) may be dynamic, not only static. In dynamic systems, the membranes may be divided, dissolved or thickened (the communication through them is inhibited);
- priorities may be attached to the evolution rules;
- the membranes may have electric charges (+, - or 0); in this case the evolution rules involve these electrical charges, too;
- the evolution rules may have more specific forms;
- the ouput can be related to an arbitrary node, not necessarily to the root of the tree. Moreover, the output may be considered to be $L \cap O^*$, where O is a given output alphabet;
- for each node i, a specific set of evolution rules may be considered;

- the sets M_i may be replaced by multisets, i.e. an object may appear more than once in M_i.

4. Implementing P systems using $CSXMS$

In [6] it was shown that a communicating X-machine system can be associated to any P system, which simulates the behaviour of the P system:

Theorem 1. *Any P system may be simulated by a $CSXMS$.*

The proof was a constructive one. Starting from a given P system, the corresponding CSXMS was built as follows. An X-machine P_i was associated to each node $i \in \{0, \dots n\}$. An additional X-machine P_{n+1} was considered too. In order to use communication through channels, the X-machine $Server$ mentioned in Section 2 was added to the system.

The idea of this construct was quite simple. The process (X-machine) P_{n+1} receives first from each of $P_1, ..., P_n$ the new created objects that are to be sent to its neighbours together with their targets, as well as the number of the new generated objects (including those which remain local) and afterwards sends to $P_0, P_1, ..., P_n$ the objects for which they are targets. These two steps are executed repeatedly until the information which P_{n+1} receives at the first step is void.

The above construction has a simple form and solves the problem. However it is not satisfactory in real implementations, due to its very low degree of parallelism: P_{n+1} waits for *all* of the components $P_1, ..., P_n$ to send their information and only afterwards executes the second step. Therefore we will present a second and much more efficient way to associate a CSXMS to a P system.

We will omit some cumbersome details, as those concerning the form of the local memories of the component of the system of X-machines. The stress will be put on the actions performed by the components and on the communication between them.

The components of the system are $P_0, P_1, ..., P_n$ and $Server$. A channel is associated to each edge of the P-tree. For each P_i let f_i be the father of node i and S_i the set of the sons of this node (of course, if i is a leaf, then $S_i = \emptyset$); P_i is linked through channels in both directions to P_{f_i} and P_s, for all s in S_i.

The basic idea of this new construct is that the nodes have to receive the new objects sent to them in a postorder manner, i.e. an inner membrane always receives information before its enveloping membrane does it.

For this purpose each node, after generating new objects, performs the following actions:

- sends to each of its sons the appropriate information
- receives information from its sons
- receives information from its father

- sends to its father the appropriate information, together with the number of new objects generated in the subtree for which it is the root

The new objects process P_0 receives at each step are sent to the output tape. As mentioned above, P_0 receives also the total number of the new generated objects. Only if this number is zero (no new objects were generated in the nodes of the P-tree), it starts the halting procedure.

The halting procedure is done in a preorder manner, using (of course) the **select** construct with a **terminate** clause. Process P_0 merely stops; in this way, the channel linking him to it unique son is disabled. When the process associate to any other node tries to receive information from its father and this father has already stoped, it will stop too; this is ensured by including in the **select** implementing this receive operation, the **terminate** clause. In this way, all channels having this node as sender are disabled.

The component P_0 sends to P_1 the number 0 and receives from it the new generated objects for which it is the target, as well as the number k of all new generated objects when passing from a configuration to the next one. If $k = 0$ then P_0 halts.

```
$nr \leftarrow -1$
repeat
   if $nr = 0$
   then select
            terminate
         end
   else -
   $nr \leftarrow 0$
   select
     1 ! 0
   end
   select
     1 ? $J, k$
   end
   $nr \leftarrow k$
until false
```

The processes $P_1, ..., P_n$ perform the actions described above:

```
repeat
   $nr \leftarrow 0$
   generate the new objects, update the current set $M_i$ and collect in $J_{f_i}$
     and $\{J_s | s \in S_i\}$ the information to be send to the neighbours
   for all $s \in S_i$
     select
       $s$ ! $J_s$
```

```
        **end**
      **endfor**
      **for** all $s \in S$
        **select**
          $s$ ? $J, k$
        **end**
        $M_i \leftarrow M_i \cup J$; $nr \leftarrow nr + k$
      **endfor**
      **select**
        $f_i$ ? $J, k$
      **or terminate**
      **end**
      $M_i \leftarrow M_i \cup J$; $nr \leftarrow nr + k$
      **select**
        $f_i$ ! $J_{f_i}$
      **end**
    **until false**
```

The number of the components in the system is $n + 2$. The number of channels used for communication is $4n + 2$: $2n$ channels linking the nodes in the P-tree and $2(n + 1)$ channels linking these nodes to $Server$.

The communication matrix may be replaced by a matrix with 4 lines and $n + 1$ columns.

The main enhancement of this new construct consists in the fact that the degree of concurrency is much higher then in the former construct and in this way corresponds to the model originally designed for P systems.

The halting of the whole CSXM system is ensured (as mentioned) in preorder, so that some nodes may still perform some actions until they are "announced" by their father that they have to halt.

5. Conclusions

In this paper we proposed a new construct for modelling P systems by CXSMS. The implementation uses communication through channels between components, as described in [3]. The CSXM associated to a P system has a high degree of concurrency, which can be exploited when a multiprocessor device is available. The new construct can be rather easily implemented in programming languages like Java, so that the evolution of a membrane computing system may also be implemented in such a language.

Further work includes the study of the different variants of P systems, as described in the third section of this paper. A Java implementation of P systems, using the tools presented in this paper, is in course of development.

References

[1] T. Bălănescu, A. J. Cowling, H. Georgescu, M. Gheorghe, M. Holcombe, C. Vertan: Communicating stream X-machines are no more than X-machines: *J.UCS*, Vol. 5, Nr. 9, September 1999, 494–507.

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

[3] A. J. Cowling, H. Georgescu, C. Vertan : A structured way to use channels for communication in X-machine systems, *FACS*, 12 (2000), 485–500.

[4] S. Eilenberg : *Automata, languages and machines*, Academic Press, 1974.

[5] H. Georgescu, C. Vertan: A new approach to communicating X-machines systems, *J.UCS*, 2000, Vol. 6, issue 5, 490–502.

[6] H. Georgescu : Modelling P systems by communicating X-machine systems. *Annals of Bucharest University, Mathematics-Informatics Series*, L (1), 3–12, 2001.

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

[8] Gh. Păun: Computing with membranes. An Introduction, *Bulletin of EATCS*, 67 (2), 139–152, 1999.

[9] Gh. Păun: Computing with membranes, *J. of Computer and System Sciences*, 61, 1 (2000), 108–143.

[10] S. N. Krishna, R. Rama: P systems with replicated rewriting, *J. Automata, Languages, and Combinatorics*, (to appear).

[11] Gh. Păun: P Systems with active membranes: Attacking NP-complete problems, *J. Automata, Languages and Combinatorics* 6, 1 (2001).

Academy of Economic Studies, Bucharest, Romania
E-mail address: hg@phobos.cs.unibuc.ro, g_horia@hotmail.com