

# Simulation and Verification of P Systems through Communicating X-machines

P.Kefalas<sup>1</sup>, G.Eleftherakis<sup>1</sup>, M.Holcombe<sup>2</sup> and M.Gheorghe<sup>2</sup>

<sup>1</sup> Dept. of Computer Science, CITY College,  
13 Tsimiski Street, Thessaloniki 546 24, Greece  
Tel. +30310-275575, Fax. +30310-287564  
{kefalas, eleftherakis}@city.academic.gr

<sup>2</sup> Dept. of Computer Science, University of Sheffield,  
Regent Court, 211 Portobello Street, Sheffield S1 4DP, UK  
{m.holcombe, m.gheorghe}@dcs.shef.ac.uk

## Abstract

The aim of this paper is to prove the suitability of a parallel distributed computational model, communicating X-machines, to simulate in a natural way a well established model of molecular computation, P systems, and to present some further benefits of the approach allowing us to check for some formal properties. A set of rules to transform any P system with symbol-objects into a communicating X-machine model is presented and a variation of temporal logic for X-machines is briefly discussed, which facilitates model checking of desired properties of the system. Finally, the benefits resulting from the transformation are discussed.

## 1. Introduction

In the last years attempts have been made in order to devise computational models in the form of generative devices like P systems (Paun, 1998, 2002), inspired by bio-chemical mechanisms occurring in alive cells, splicing systems (Paun et al., 1998), expressing transformations defined upon DNA strands (more information on these subjects may be found at <http://psystems.disco.unimib.it/> and <http://www.wi.leidenuniv.nl/~pier/dna.html>, respectively). New computational paradigms of modelling concurrent systems' behaviour such as Gamma (Banatre and LeMetayer, 1990) or Cham (Berry and Boudol, 1992), inspired by chemical reactions developing in parallel, were introduced in the field of concurrent programming. Experiments have been made in order to show how DNA strands may be used as a massive parallel computer to solving well-known hard problems (Adleman, 1994). All these models, paradigms or experiments rely on some bio-chemical facts in approaching new ways of computing either at some abstract level or in a more practical manner. There are, on the other hand, computational models like random grammars, Boolean networks (Kauffman, 1993), developed to express bio-chemical reactions occurring at the cell level, or X-machines, utilized to model metabolic pathways (Holcombe, 2001), or the behaviour of bee colonies (Gheorghe et al., 2001). The important role of generative models has been emphasised in the context of analysis of the emergence of functional adaptive systems (Kauffman, 1993).

The need for developing complementary models (Clark and Paton, 1998) as well as for the use of general universal models, such as Turing machines (Chaitin, 2002), in order to specify various complex aspects related to micro or macro bio-structures led to approaches based on hybrid models (Duan et al., 1995). More complex models have been obtained by aggregating various computational paradigms – P systems and X-machines for instance –, and yielding mechanisms that benefit from both components: they exhibit an internal hierarchical structure, defined in accordance to P system's membrane organization, and a distributed set of evolution rules triggered in certain states acting like in an X-machine context (Balanescu et al., 2002).

Some variants of X-machines may become suitable for molecular computing models due to their completeness property, capability of naturally simulating different computationally complete models – Turing machines (Eilenberg, 1974), two-stack automata (Ipate and Holcombe, 1996), P systems with replicated rewriting (Aguado et al., 2001) –, suitability to define dynamic systems reacting to input stimuli, flexibility in expressing hierarchical or distributed systems, and ability in capturing hybrid specifications.

In this paper we focus on a particular type of X-machines, called communicating X-machines (Kefalas et al., 2001), introduced in order to simulate P systems with symbol-objects. At least three benefits of this approach may be emphasised:

- it is a natural environment to simulate P systems in,
- it is enough flexible such as when a new component is introduced into the play the others need not to be restructured – this feature might become useful when simulating membrane division –, and
- it is much more efficient than single X-machines initially used to simulate P systems (Aguado et al., 2001).

Communicating X-machines have been used as a suitable paradigm to modelling agent based specification (Kefalas 2002, Kefalas et al., 2003a) where model checking features have been considered to validate in a formal framework properties of the specified system. Some benefits of this new dimension of the

model are to be discussed in the context of the communicating X-machine systems used to simulate P systems with symbol-objects. The paper is structured into four main parts. First, basic definitions of P systems, X-machines and communicating X-machines are given. A section on simulating a P system with symbol-objects follows, discussing the transformation rules required in building a communicating X-machine system that simulates the behaviour of such a P system. Some benefits of modelling a P system with symbol-objects as an X-machine are discussed in the context of the X-machine model checking framework. Finally, some conclusions and further work are discussed.

## 2. Basic Definitions

A **P system with symbol-objects** is a computing device in which multisets of objects as well as evolution rules are placed in compartments, called **regions**, defined by a membrane structure. Formally, a P system is a construct (Paun, 1998):

$$\Pi = (V, T, \mu, w_1, \dots, w_m, (R_1, \rho_1), \dots, (R_m, \rho_m)),$$

where:

- $V$  is an alphabet - its elements are called **objects**;
- $T \subseteq V$  is the **output alphabet**;
- $\mu$  is a **membrane structure** consisting of  $m$  membranes, with the membranes labelled by the elements of a given set  $H$  of  $m$  labels,  $H = \{1, 2, \dots, m\}$ ;  $m$  is called the **degree** of  $\Pi$ ;
- $w_i$ ,  $1 \leq i \leq m$ , are strings which represent **multisets** over  $V$  **associated with the regions**  $1, 2, \dots, m$  of  $\mu$ ;
- $R_i$ ,  $1 \leq i \leq m$ , are finite sets of **evolution rules** over  $V$  - each  $R_i$  is associated with the region  $i$  of  $\mu$ ;  $\rho_i$  is a **partial order** relation over  $R_i$ , called a priority relation (on the rules of  $R_i$ ).

An evolution rule is of the form  $u \rightarrow v$ , where:

- $u$  is a string over  $V$  and
- $v = v'$  or  $v = v'\delta$ , where  $v'$  is a string over  $\{\alpha, \alpha_{out}, \alpha_{inj} \mid \alpha \in V, 1 \leq j \leq m\}$  and  $\delta$  is a special symbol not in  $V$ .

The membrane structure  $\mu$  is a hierarchical arrangement of membranes. Usually, the hierarchical structure is represented as  ${}_1[ \dots [ \dots [ \dots [ \dots ]_j \dots ]_i \dots ]_1$ , where  $1 < i, j \leq m$ . Each region may contain objects which form multisets over  $V$  and a set of evolution rules associated with it. The rules determine the action to be performed and possibly the region in which the effects of the action will appear. For example, a rule of the form  $ab \rightarrow a_{in3}d_{out}$  denotes that in the presence of objects  $a$  and  $b$  and an inner region with its membrane labelled 3, the rule is triggered with the following effect:

- objects  $a$  and  $b$  are removed from the current region,
- object  $a$  is placed in the same region as the one in which this rule resides,
- object  $c$  is placed in the inner region 3 as defined by the membrane structure, and
- object  $d$  is placed in the outer region as defined by the membrane structure.

The computation is carried out in a synchronous way (**macro-steps**) for all regions, i.e. all regions trigger as many rules as possible within the same cycle but the new objects which are generated during a computation macro-step are not utilized by any of the evolution rules within the same cycle. In one macro-step, the rules associated to each region are applied in a parallel non-deterministic manner (several **micro-steps**) to all objects residing in that region that match these rules. The result of a macro-step is a transition of the system to a new configuration, i.e. some objects may be removed from some regions, some others may be placed in as a result of rules that were triggered, some membranes may be dissolved, etc. The sequence of transitions is called a computation. A computation starting from the initial configuration,  $w_i$ ,  $1 \leq i \leq m$ , is successful if and only if no rule is applicable in any of the regions. The result of a computation is the number of objects sent to the environment during the computation.

An **X-machine** is a general computational model (Eilenberg, 1974) that resembles a Finite State Machine (FSM) but with two significant differences:

- there is an underlying data set attached to the machine, and
- the transitions are not labelled with simple inputs but with functions that operate on inputs and underlying data set values.

These differences allow the X-machines to be more expressive and flexible than the FSMs. Other machine models like *pushdown automata* or *Turing machines* are too low level and hence of little use for real system specification. X-machines employ a diagrammatic approach of modelling the control by extending the expressive power of the FSM. They are capable of modelling both the data and the control of a system. Data is held in memory, which is attached to the X-machine. Transitions between states are performed through the application of functions, which are written in a formal notation and model the processing of the data. An extremely useful in practice is the class of so called **stream X-machines**, defined by the restrictions on the underlying data set, involving input symbols, memory values and output symbols. Functions receive input symbols and memory values, and produce output while modifying the memory values (figure 1). The machine, depending on the current state of control and the current values of the memory, consumes an input

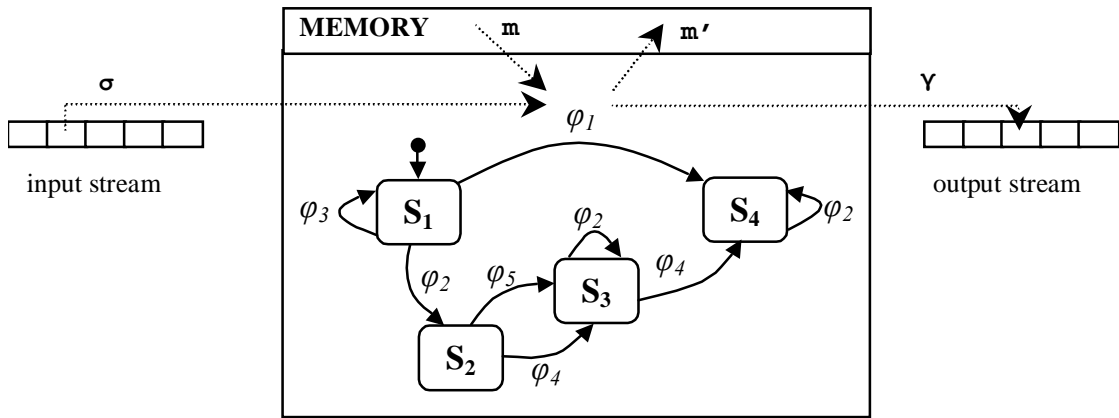
symbol from the input stream and determines the next state, the new memory value and the output symbol, which will be part of the output stream. The formal definition of a stream X-machine is (Ipaté and Holcombe, 1996):

$$M = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0),$$

where:

- $\Sigma, \Gamma$  are the **input** and **output finite alphabets** respectively;
- $Q$  is the finite set of **states**;
- $M$  is the (possibly) infinite set of **memory values**;
- $\Phi$  is the **type** of the machine  $M$ , a finite set of partial functions  $\varphi$  that map an input and a memory value to an output and a new memory value,  $\varphi: \Sigma^* \times M \rightarrow \Gamma^* \times M$ ;
- $F: Q \times \Phi \rightarrow P(Q)$  is the **next state** partial function that given a state and a function from the type  $\Phi$ , denotes a set of next states;  $P(Q)$  is the powerset of  $Q$  – the set of all subsets of  $Q$  –. When the set contains more than one state then the machine will decide where to move in by picking up non-deterministically one state from this set.  $F$  is often described as a transition state diagram;
- $q_0$  and  $m_0$  are the **initial state** and **initial memory value**, respectively.

Starting from the initial state  $q_0$  with the initial memory  $m_0$ , an input symbol  $\sigma \in \Sigma$  triggers a function  $\varphi \in \Phi$  which in turn causes a transition to a new state  $q \in Q$  (from the set  $F(q_0, \varphi)$ ) and a new memory value  $m \in M$ . The sequence of transitions caused by the stream of input symbols is called a computation. The computation halts when all input symbols are consumed. The result of a computation is the sequence of outputs produced by the sequence of transitions.



**Fig. 1.** An abstract example of an X-machine;  $\varphi_i$ : functions operating on inputs and memory,  $S_i$ : states. The general form of functions is:  $\varphi(\sigma, m) = (\gamma, m')$ .

A **communicating X-machine** model consists of several X-machines, which are able to exchange messages. These are normally viewed as inputs to some functions of an X-machine model, which in turn may affect the memory structure. A communicating X-machine model can be generally defined as a tuple:

$$((M_i)_{i=1..n}, CR),$$

where

- $M_i$  is the  $i^{\text{th}}$  **X-machine component** that participates in the system, and
- $CR$  is a **communication** relation between the  $n$  X-machine components.

There are several approaches in order to formally define a communicating X-machine. Some of them deviate from the original definition of the X-machine  $M_i$ , which has the effect of not being able to reuse existing models (Balanescu et al., 1999; Cowling et al., 2000; Barnard, 1998). Also, in these approaches  $CR$  is defined in different ways, with the effect of achieving either synchronous or asynchronous communication.

In the current work, the relation  $CR$  is defined as a relation  $CR \subseteq M \times M$ , where  $M = \{M_i \mid 1 \leq i \leq n\}$ , which determines the communication channels that exist between the X-machines of the system. A tuple  $(M_i, M_k) \in CR$  denotes that X-machine  $M_i$  can write a message through a communication channel to a corresponding input stream of X-machine  $M_k$ . A communicating X-machine component is therefore defined as (Kefalas et al., 2002):

$$M_i = (\Sigma_{M_i}, \Gamma_{M_i}, Q_{M_i}, M_{M_i}, \Phi_{M_i}, F_{M_i}, q_{0M_i}, m_{0M_i}),$$

that matches the original definition of a stream X-machine. The only difference lies in  $\Phi_{M_i}$ , which contains four types of functions  $\varphi_{M_i}$ :

- functions that read from standard input stream and write to standard output stream:

$$\varphi_{M_i}(\sigma, m) = (\gamma, m') \text{ where } \sigma \in \Sigma_{M_i}, \gamma \in \Gamma_{M_i}, m, m' \in M_{M_i},$$

- functions that read from a communication input stream a message that is sent by another X-machine  $M_j$  and write to standard output stream:  
 $\varphi_{M_i}((\sigma)_{M_j}, m) = ((\gamma)_{M_i}, m')$  where  $\sigma \in \Sigma_{M_i}$ ,  $\gamma \in \Gamma_{M_i}$ ,  $m, m' \in M_{M_i}$  and  $(M_j, M_i) \in CR$ ,
- functions that read from standard input stream and write to a communication output stream a message that is sent to another X-machine  $M_k$ :  
 $\varphi_{M_i}(\sigma, m) = ((\gamma)_{M_k}, m')$  where  $\sigma \in \Sigma_{M_i}$ ,  $\gamma \in \Gamma_{M_i}$ ,  $m, m' \in M_{M_i}$  and  $(M_i, M_k) \in CR$ ,
- functions that read from a communication input stream a message that is sent by another X-machine  $M_j$  and write to a communication output stream a message that is sent to another X-machine  $M_k$ :  
 $\varphi_{M_i}((\sigma)_{M_j}, m) = ((\gamma)_{M_k}, m')$  where  $\sigma \in \Sigma_{M_i}$ ,  $\gamma \in \Gamma_{M_i}$ ,  $m, m' \in M_{M_i}$  and  $(M_j, M_i) \in CR$  and  $(M_i, M_k) \in CR$ .

The notation  $(\sigma)_{M_j}$  denotes an incoming input from X-machine  $M_j$  while  $(\gamma)_{M_k}$  denotes an outgoing output to X-machine  $M_k$ . Instead of communicating with only one X-machine  $M_k$ ,  $M_i$  may communicate with  $M_{k1}, \dots, M_{kp}$ , sending them  $\sigma_1, \dots, \sigma_p$ , respectively; in this case all  $(M_i, M_{kj})$ ,  $1 \leq j \leq p$ , must belong to  $CR$  and  $(\gamma)_{M_k}$  will be replaced by  $(\gamma_1)_{M_{k1}} \& \dots \& (\gamma_p)_{M_{kp}}$ .

Graphically, if a *solid circle* appears on a transition diagram, this function accepts input from the communicating stream instead of the standard input stream. It is read as “reads from”. If a *solid diamond* appears on a transition function, this function may write to a communicating input stream of another X-machine. It is read as “writes to” (figure 2).

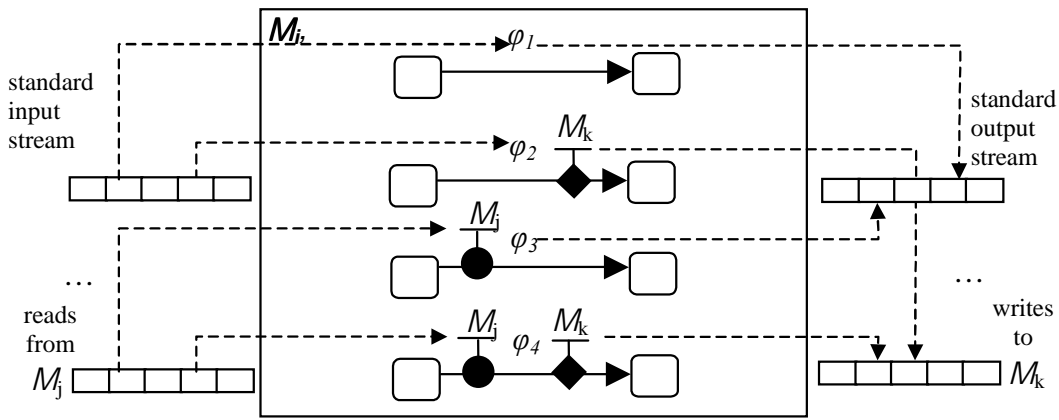


Fig. 2. An abstract example of a communicating X-machine component.

### 3. Modelling P Systems with Communicating X-machines

In this section, we will describe a number of transformation rules, which, if applied to a P system, can model this P system as a communicating X-machine system. The rules are general and can facilitate automatic transformation, thus exploiting important features of X-machine modelling, such as model checking. Let  $\Pi = (V, T, \mu, w_1, \dots, w_m, (R_1, \rho_1), \dots, (R_m, \rho_m))$  be a P system, where  $m$  is the degree of  $\Pi$ .

#### Transformation Rule 1: X-machine components of the communicating system

Each region of  $\Pi$  is modelled as an X-machine component  $M_i$ , therefore we start with the communicating X-machine system  $((M_1, M_2, \dots, M_m), CR)$ .  $CR$  is defined below.

#### Transformation Rule 2: Communication of components

The structure  $\mu$  of  $\Pi$  as well as the evolution rules  $R_i$  determine the communication channels between the X-machine components, i.e. the relation  $CR$ . For every string  $v$  over  $\{\alpha, \alpha_{out}, \alpha_{inj} \mid \alpha \in V, 1 \leq j \leq m\}$ , a tuple  $(M_i, M_j) \in CR$  if one of the following cases apply:

- Each rule instance of the form  $u \rightarrow v_1 \alpha_{out} v_2$  in  $R_i$ , implies a communication channel between  $M_i$  and  $M_j$ , if a pattern structure like  $j[\dots i[\dots]j \dots]j$  appears in  $\mu$ . That will allow X-machine  $M_i$  to send object  $\alpha$  as messages to X-machine  $M_j$ .
- Each rule instance of the form  $u \rightarrow v_1 \alpha_{inj} v_2$  in  $R_i$ , implies a communication channel between  $M_i$  and  $M_j$ , if a pattern structure like  $i[\dots j[\dots]j \dots]i$  appears in  $\mu$ . That will allow X-machine  $M_i$  to send object  $\alpha$  as messages to X-machine  $M_j$ .
- Each rule instance of the form  $u \rightarrow v \delta$  in  $R_i$ , implies a communication channel between  $M_i$  and  $M_j$ , if a pattern structure like  $j[\dots i[\dots]j \dots]j$  appears in  $\mu$ . That will allow X-machine  $M_i$  to send all its objects as messages to X-machine  $M_j$  when the membrane  $i$  is dissolved.

### Transformation Rule 3: Synchronization of components

As it stated in (Paun,1998), membrane systems are synchronous, in the sense that a global clock is assumed, i.e. the same clock holds for all regions of the system. The clock produces time units for every macro-step. Each macro-step consists of several micro-steps, i.e. a non-deterministic sequence of evolving rule applications. Once micro-steps for all membranes are completed a next macro-step should begin.

The above can be modelled by an X-machine, which acts as a clock as well as a scheduler. The clock X-machine component will send a message (*tick*) to all other X-machines, which can then start their micro-steps. This will require synchronization, which can be achieved if all X-machines notify the clock about their status. Only when the X-machines have completed their micro-steps, the clock should send a new *tick*.

The clock X-machine is defined as follows:

- There is only one state, i.e.  $Q_{clock} = \{scheduling\}$ .
- This is also the initial state,  $q_{0\ clock} = scheduling$ .
- The memory holds an m-tuple, i.e. the number of X-machines, representing the state of computation in each of the regions. Therefore,  $M_{clock} = \{(status\_x1, \dots, status\_xm) \mid status\_xi \in \{ready, ready', evolving, dissolved, dissolved'\}, 1 \leq i \leq m\}$ .
- Initially, one assumes that all X-machines are ready for a macro-step:  $m_{0\ clock} = (ready', \dots, ready')$ .
- The input set elements will either trigger the generation of a new *tick* or the changing of status of each X-machine that is represented in the memory, i.e.  $\Sigma_{clock} = \{null, cycle\_completed, cycle\_completed\_and\_dissolved, end\_communication\}$ , where *null* identifies the empty message.
- The output set elements are either ticks of the clock when all X-machines are ready for the next macro-step or messages indicating that the machine associated with membrane *i* has completed a macro-cycle, i.e.  $\Gamma_{clock} = \{tick\} \cup \{completed_i, completed\_macro\_step_i \mid 1 \leq i \leq m\}$ .
- Since there is only one state, the next state transition function contains elements that indicate a transition to the same state for all functions, i.e.  $F_{clock}(scheduling, \varphi_{i\ clock}) = \{scheduling\}$ , where  $\varphi_{i\ clock} \in \Phi_{clock}$ .

- The type  $\Phi_{clock}$  contains:

$finished_i((cycle\_completed)_i, (s_1, \dots, s_i, \dots, s_m)) = (completed_i, (s_1, \dots, ready, \dots, s_m)), 1 \leq i \leq m,$

$finished_i((cycle\_completed\_and\_dissolved)_i, (s_1, \dots, s_i, \dots, s_m)) =$

$(completed_i, (s_1, \dots, dissolved, \dots, s_m)), 1 \leq i \leq m,$

$finished_i((end\_communication)_i, (s_1, \dots, s_i, \dots, s_m)) = (completed\_macro\_step_i, (s_1, \dots, s'_i, \dots, s_m)), 1 \leq i \leq m,$

$end\_macro\_step(null, (s_1, \dots, s_m)) = ((end)_{M_1} \& \dots \& (end)_{M_m}, (s_1, \dots, s_m)), s_i$  is either *ready* or *dissolved*,

$generate\_time\_unit(null, (s_1, \dots, s_m)) = ((tick)_{M_1} \& \dots \& (tick)_{M_m}, (c_1, \dots, c_m))$

if  $s_1 \neq evolving$  and ... and  $s_m \neq evolving$

then either

$c_i = dissolved$  when  $s_i = dissolved'$  or  $s_i = dissolved$ , or

$c_i = evolving$  when  $s_i = ready'$ ;  $1 \leq i \leq m.$

Once either *cycle\_completed* or *cycle\_completed\_and\_dissolved* is sent by  $M_i$ ,  $finished_i$  is triggered and the corresponding memory status becomes either *ready* or *dissolved*, respectively. When all machines have been informed that a macro-step is over then an *end\_comunication\_i* message is sent to clock machine by every  $M_i$  that triggers  $finished_i$  which in turn changes the current memory status (either *ready* or *dissolved*) into its prime value. Once all active (not dissolved) machines  $M_i$  have sent either *cycle\_completed* or *cycle\_completed\_and\_dissolved* to the clock machine then all the memory components of the clock machine are either *ready* or *dissolved* and *end\_macro\_step* is triggered and an *end* message is sent to all  $M_i$ . The function *generate\_time\_unit* restarts a new macro-step once all the memory values of the clock machine are either *ready'* or *dissolved, dissolved'*. The status of the memory component will remain *dissolved* if the membrane referred by is dissolved or will change to *evolving* otherwise.

The overall communicating X-machine system finally becomes:

$$((M_1, M_2, \dots, M_m, clock), CR \cup \{(clock, M_i), (M_i, clock) \mid 1 \leq i \leq m\})$$

where  $(M_1, M_2, \dots, M_m)$  are defined by Transformation Rule 1 and  $CR$  is defined by Transformation Rule 2.

### Transformation Rule 4: States of each component

Each X-machine  $M_i$  has a state set

$Q_{M_i} = \{ready\_for\_macro\_step, evolving, end\_evolving\_rules\}$ , if  $u \rightarrow v \delta \notin R_i$ ,

or

$Q_{M_i} = \{ready\_for\_macro\_step, evolving, end\_evolving\_rules, dissolved\}$ , if  $u \rightarrow v \delta \in R_i$ .

The initial state of any X-machine  $M_i$  is  $q_{0M_i} = ready\_for\_macro\_step$ .

### Transformation Rule 5: Memory

The memory of each X-machine  $M_i$  is a tuple containing two components that are multisets over  $V$  and a flag indicating whether the machine is still active or corresponds to a membrane that has been previously dissolved:  $M_{M_i} = \{(z_1, z_2, f) \mid z_1, z_2 \in V^*, f \in \{active, dissolved\}\}$ . The first element of the tuple is the multiset of current objects and the second element of the tuple is the multiset of new objects that appear in the region  $i$  during a macro-step. The latter behaves as a temporary buffer, so that at the end of each macro-step all objects of this temporary buffer appear to the region as current objects. Otherwise, if there was only one multiset then the multiset of new objects could trigger a rule within the current macro-step, something which is not allowed to happen.

The initial memory of an X-machine  $M_i$  should contain the string  $w_i$  associated with membrane  $i$  while the temporary buffer of objects is empty and the machine is active, i.e.  $m_{0M_i} = (w_i, \varepsilon, active)$ , where  $\varepsilon$  denotes the empty multiset.

### Transformation Rule 6: Input

The input set contains elements that can trigger the functions, which we define below, as equivalent to evolution rules. The input set elements of an X-machine  $M_i$  are:

- *tick* and *end* that are received as messages through the communication channel from clock X-machine,
- *null* that is the empty message that can trigger any transition at any time, and
- multisets that represent objects which are passed as messages through the communication channel from another X-machine component to  $M_i$ .

Therefore,  $\Sigma_{M_i} = \{null, tick, end\} \cup V$ .

### Transformation Rule 7: Output

Accordingly, the output set of an X-machine  $M_i$  contains:

- multisets that represent objects which are passed as messages through the communication channel from  $M_i$  to another X-machine component,
- *cycle\_completed*, *cycle\_completed\_and\_dissolved*, *end\_communication* that are messages sent to the clock X-machine,
- *evolution\_started*, *ignoring\_tick*, *ignoring\_end*, *received\_objects* used by different functions.

The first type of output will be used when rules of the form  $u \rightarrow v\alpha_{out}$  or  $u \rightarrow v\delta$  are in  $R_i$  while the second type when a macro-step is completed. Therefore,  $\Gamma_{M_i} = V \cup \{cycle\_completed, cycle\_completed\_and\_dissolved, end\_communication, evolution\_started, ignoring\_tick, ignoring\_end, received\_objects\}$ .

### Transformation Rule 8: Type

For each X-machine  $M_i$ , the functions in  $\Phi_i$  are divided into four categories:

- Functions that establish communication with the clock X-machine;
- Functions that facilitate the receipt of incoming messages from other X-machines components;
- Functions that deal with dissolved membranes;
- Functions that model the evolution rules.

#### Transformation Rule 8.1: Macro-steps

There are three functions that facilitate communication with clock; one in order to initialise a macro-step and the others to determine the completion of a macro-step. The first, named *do\_micro\_steps*, accepts a *tick* as an input through the communication channel from clock but does not change the memory:

$do\_micro\_steps((tick)_{clock}, (curr\_objs, buffer, active)) = (evolution\_started, (curr\_objs, buffer, active))$ .

The second function sends a message *cycle\_completed* to the clock only when no evolution rule is applicable, i.e. when strings in the current objects do not trigger any rule:

$end\_evolution(null, (curr\_objs, buffer, f)) = ((cycle\_completed)_{clock}, (curr\_objs, buffer, f))$ ,  $f=active$  or  $f=dissolved$ ;

if  $u_{i1} \subset curr\_objs$  and ... and  $u_{in} \subset curr\_objs$ , where  $u_{ij} \rightarrow v$  in  $R_i$ ,  $n$  is the cardinality of  $R_i$  and " $\subset$ " denotes multiset inclusion.

The third function is triggered when an *end* comes from the clock through the communication channel, which means all X-machine components finished up their macro-step; *buffer* is emptied into the current strings, when  $M_i$  is active or is sent together with the current objects to  $M_k$  - if this is associated to the surrounding region -, when  $M_i$  indicates a machine that model a membrane being dissolved:

$end\_micro\_steps((end)_{clock}, (curr\_objs, buffer, active)) = ((end\_communication)_{clock}, (curr\_objs \uplus buffer, \varepsilon, active))$ ,

$end\_micro\_steps((end)_{clock}, (curr\_objs, buffer, dissolved)) = ((end\_communication)_{clock} \& (curr\_objs \uplus buffer)_{M_k}, (\varepsilon, \varepsilon, dissolved))$ ,

The function should transfer all objects from buffer into the memory part representing current objects in order to be ready for the next macro-step. The symbol  $\cup$  denotes the union of two multisets.

### Transformation Rule 8.2: Incoming objects

Due to transitions that model evolution rules that exist in other X-machines and involve objects to be sent to the X-machine  $M_i$  one should have functions able to process objects received by. This is achieved by a function that reads objects sent as messages through the communication channel from X-machine  $M_j$ :

$$\text{incoming}((\text{objects})_{M_j}, (\text{curr\_objs}, \text{buffer}, \text{active})) = (\text{received\_objects}, (\text{curr\_objs}, \text{buffer} \cup \text{objects}, \text{active})), (M_j, M_i) \in CR.$$

The incoming objects are placed in *buffer*.

### Transformation Rule 8.3: Dissolved membranes

As stated in Transformation Rule 1, each region  $i$  is modelled by an X-machine  $M_i$ . Although in the computation performed in a P system a membrane may dissolve, in the simulation of the P system with X-machines, an X-machine cannot cease to exist. That is the reason for including a *dissolved* state in the Transformation Rule 4. On the other hand, the communication structure of the communicating X-machine system is static as described in Transformation Rule 2. A way to overpass this lack of dynamic organisation is to allow messages to travel through dissolved membranes, if the membranes to be dissolved are not the innermost ones. This is feasible through a function, called *pass\_it\_over*, which having read a message from X-machine  $M_j$  passes the message through the communication channel to another X-machine  $M_k$ :

$$\text{pass\_it\_over}((\text{objects})_{M_j}, (\varepsilon, \varepsilon, \text{dissolved})) = ((\text{objects})_{M_k}, (\varepsilon, \varepsilon, \text{dissolved}))$$

where  $(M_j, M_i) \in CR$  and  $(M_i, M_k) \in CR$  are determined by the communication setup  $CR$  in Transformation Rule 2. In addition, a dissolved membrane should ignore any *tick* or *end* coming from the clock X-machines since it cannot initiate any more macro-steps or stop a macro-step:

$$\begin{aligned} \text{ignore\_tick}((\text{tick})_{\text{clock}}, (\varepsilon, \varepsilon, \text{dissolved})) &= (\text{ignoring\_tick}, (\varepsilon, \varepsilon, \text{dissolved})), \\ \text{ignore\_end}((\text{end})_{\text{clock}}, (\varepsilon, \varepsilon, \text{dissolved})) &= (\text{ignoring\_end}, (\varepsilon, \varepsilon, \text{dissolved})). \end{aligned}$$

### Transformation Rule 8.4: Evolution rules

Each evolution rule  $u \rightarrow v$  in  $R_i$  is modelled as a function in the corresponding machine  $M_i$ . The rule is triggered only if  $u$  appears in the multiset of current objects. The objects in  $u$  are removed from the multiset of current objects and objects occurring in  $v$  are added to the buffer. However, there are specific cases in the presence or absence of  $\alpha_{in}$ ,  $\alpha_{out}$  and  $\delta$ .

Assume that the membrane structure follows the pattern  $k[\dots[j_1[\dots[j_1 \dots j_p[\dots]_{j_p} \dots]_{j_1} \dots]_{j_1} \dots]_{j_1} \dots]_k$ , which means, according to Transformation Rule 2 of communication setup  $CR$ , that  $(M_i, M_{j_h}) \in CR$ ,  $1 \leq h \leq p$ , and  $(M_i, M_k) \in CR$ . The most general form of a rule is  $u \rightarrow x_1 \alpha_1 \dots x_r \alpha_r x_{r+1} \delta$ , where  $x_i \in V^*$ ,  $\alpha_i \in \{\alpha_{in}, \alpha_{out} \mid \alpha \in V, h \in \{j_1, \dots, j_p\}\}$ ,  $1 \leq i \leq r$ . In the notation below the symbol “-” denotes the difference of two multisets, while the symbol “ $\cup$ ” denotes the union of two multisets. There are four case:

- if  $t: u \rightarrow x_1 \alpha_1 \dots x_r \alpha_r x_{r+1}$  (no dissolve operator) then a function  $t$  is defined as follows:  
 $t(\text{null}, (\text{curr\_objs}, \text{buffer}, \text{active})) = ((\alpha_1)_{M_{j_1}} \& \dots \& (\alpha_r)_{M_{j_r}}, (\text{curr\_objs} - u, \text{buffer} \cup x_1 \cup \dots \cup x_{r+1}, \text{active}))$ , if  $\{i_1, \dots, i_r\} \subseteq \{j_1, \dots, j_p\}$ ,  $u \subseteq \text{curr\_objs}$ , and there is no rule  $t'' \in R_i$ ,  $t'' > t$  and its left hand side is in *curr\_objs*;
- if  $t: u \rightarrow x$ ,  $x \in V^*$  (no dissolve operator), then the function is:  
 $t(\text{null}, (\text{curr\_objs}, \text{buffer}, \text{active})) = (x, (\text{curr\_objs} - u, \text{buffer} \cup x, \text{active}))$ , if  $u \subseteq \text{curr\_objs}$ , and there is no rule  $t'' \in R_i$ ,  $t'' > t$  and its left hand side is in *curr\_objs*;
- if  $t: u \rightarrow x_1 \alpha_1 \dots x_r \alpha_r x_{r+1} \delta$  (dissolve operator present) then a function  $t$  is defined as follows:  
 $t(\text{null}, (\text{curr\_objs}, \text{buffer}, \text{active})) = ((\alpha_1)_{M_{j_1}} \& \dots \& (\alpha_r)_{M_{j_r}}, (\text{curr\_objs} - u, \text{buffer} \cup x_1 \cup \dots \cup x_{r+1}, \text{dissolved}))$ , if  $\{i_1, \dots, i_r\} \subseteq \{j_1, \dots, j_p\}$ ,  $u \subseteq \text{curr\_objs}$ , and there is no rule  $t'' \in R_i$ ,  $t'' > t$  and its left hand side is in *curr\_objs*;
- if  $t: u \rightarrow x \delta$ ,  $x \in V^*$  (dissolve operator present), then the function is:  
 $t(\text{null}, (\text{curr\_objs}, \text{buffer}, \text{active})) = (x, (\text{curr\_objs} - u, \text{buffer} \cup x, \text{dissolved}))$ , if  $u \subseteq \text{curr\_objs}$ , and there is no rule  $t'' \in R_i$ ,  $t'' > t$  and its left hand side is in *curr\_objs*.

### Transformation Rule 9: Next state partial functions

For every X-machine  $M_i$  the next state partial function  $F_{M_i}$  is the following:

$$\begin{aligned} F_{M_i}(\text{ready\_for\_macro\_step}, \text{do\_micro\_steps}) &= \{\text{evolving}\}, \\ F_{M_i}(\text{evolving}, \text{end\_evolution}) &= \{\text{end\_evolution\_rules}\}, \\ F_{M_i}(\text{evolving}, \text{incoming}) &= \{\text{evolving}\}, \\ F_{M_i}(\text{evolving}, t_j) &= \{\text{evolving}\}, \text{ where function } t_j \text{ models an evolution rule of } R_i, \\ F_{M_i}(\text{end\_evolution\_rules}, \text{incoming}) &= \{\text{end\_evolution\_rules}\}, \\ F_{M_i}(\text{end\_evolution\_rules}, \text{end\_micro\_steps}) &= \{\text{dissolved}, \text{ready\_for\_macro\_step}\}. \end{aligned}$$

When the membrane  $i$  is dissolved then in  $M_i$  the following transitions are defined:

$$F_{M_i}(\text{dissolved}, \text{pass\_it\_over}) = \{\text{dissolved}\},$$

$$F_{M_i}(\text{dissolved}, \text{ignore\_tick}) = \{\text{dissolved}\},$$

$$F_{M_i}(\text{dissolved}, \text{ignore\_end}) = \{\text{dissolved}\},$$

The behaviour of the system may be described as follows:

- It starts with:  $q_{0M_i} = \text{ready\_for\_macro\_step}$ ,  $1 \leq i \leq m$ ,  $q_{0 \text{ clock}} = \text{scheduling}$ ,  $m_{0M_i} = (w_i, \varepsilon, \text{active})$ ,  $1 \leq i \leq m$ , and  $m_{0 \text{ clock}} = (\text{ready}', \dots, \text{ready}')$ .
- The communicating X-machine system will start working when from the clock X-machine a *tick* message is sent to all  $M_i$ ,  $1 \leq i \leq m$ ; this is achieved by triggering *generate\_time\_unit* that will also change (*ready'*, ..., *ready'*) into (*evolving*, ..., *evolving*).
- Once a *tick* message has been sent to each  $M_i$ ,  $1 \leq i \leq m$ , then *do\_micro\_steps* is triggered and an *evolution\_started* output is provided and consequently every machine  $M_i$  will enter the state *evolving*.
- In *evolving* state the machine  $M_i$  may
  - receive objects from other machine components according to the communication protocol defined by *CR – incoming* function will update *buffer* keeping the same state *evolving* -, or
  - simulate evolution rules laying in region  $i$  of  $\Pi$  as defined by Transformation Rule 8.4 –  $t$  functions will update both *curr\_objs* and *buffer*, entering the same state *evolving* and keeping the third component of the memory to *active* value when  $\delta$  is not present in the corresponding evolution rule of  $R_i$  or change it into *dissolved* otherwise -.
- When none of  $t$ 's may be applied then *end\_evolution* is triggered and either a *cycle\_completed* or a *cycle\_completed\_and\_dissolved* message is sent to clock X-machine.
- When all  $M_i$  X-machines finish up their work cycle, then clock X-machine will wake up and *end\_macro\_step* is triggered (all memory status values are either *ready* or *dissolved*) and an *end* message is sent to every machine  $M_i$ .
- X-machines  $M_i$  receive *end* message and *end\_micro\_steps* functions are triggered; these functions update their memory values and *end\_communication* messages are sent back to clock X-machine; if some  $M_i$  is in state *dissolved* then *end* message is ignored.
- The clock X-machine in turn reacts to *end\_communication* messages by triggering *finish<sub>i</sub>* and changing the corresponding memory status from  $s_i$  into  $s_i'$ .
- When all the component memory values of clock machine are either *ready'* or *dissolved*, *dissolved'*, then the function *generate\_time\_unit* is triggered and these values are transformed into *evolving* or *dissolved*, respectively; a *tick* message is sent to all  $M_i$ ; a new macro-step in active components may restart.

It may be observed that no stopping conditions occur in the communicating X-machine system built in this way. We avoided deliberately introducing this construction in order to simplify the approach. Such a feature may be easily accommodated into the system by changing *end\_micro\_steps* to signal when no evolution rule has been applied, and adding a new condition to *generate\_time\_unit* in order to stop the whole simulation process when no evolution rule occurred.

The above presented approach has two interesting aspects: on the one hand it shows that the behaviour of a P system with symbol-objects may be simulated as a communicating X-machine system and, on the other hand, by its algorithmic structure this construction may be used to implement a P system with symbol-objects in a language suitable to specify any X-machine definition. In a language such as XMDL (Kefalas et al, 2003a) we may animate an X-machine specification and get (partial) descriptions of the behaviour of the P systems modelled as communicating X-machine systems. The construction provided for getting a communicating X-machine system from a P system with symbol-objects leads to an automatic way of devising two parts of the XMDL description of the system:

- a fixed part containing the whole definition of the clock X-machine and the states, memory values, partially the next state function definition and some of the processing functions (*do\_micro\_steps*, *incoming*, *end\_evolution*, *end\_micro\_steps*, *pass\_it\_over*, *ignore\_tick*, *ignore\_end*) of  $M_i$  machine components, and
- the bunch of functions associated with evolution rules that may be introduced according to each P system specification; the algorithm of defining them is given by transformation Rule 8.4.

The approach may be further enriched by building in top of the X-machine model a coherent framework allowing to check that a system specified in this way exhibits some formal properties, issue that is addressed in the next part.

### Example Transformation

Let us consider the definition of the P system shown in figure 3:

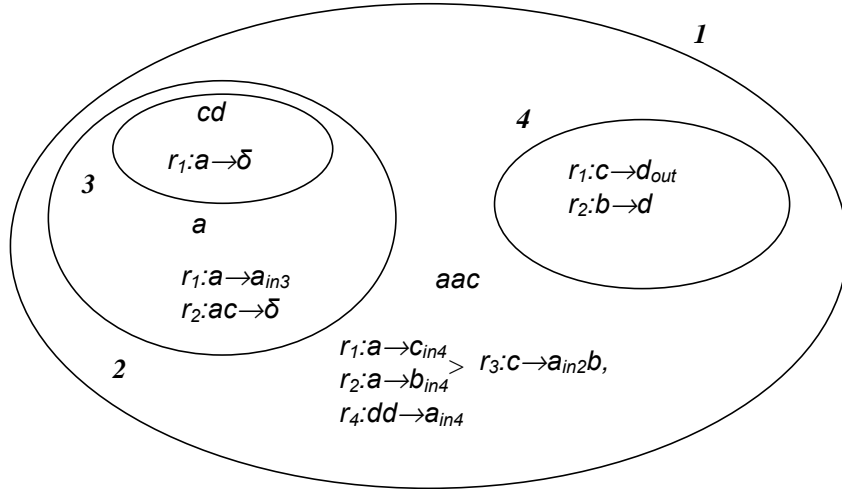
$$\Pi = (V, T, \mu, w_1, w_2, w_3, w_4, (R_1, \rho_1), (R_2, \rho_2), (R_3, \rho_3), (R_4, \rho_4))$$

$$V = \{a, b, c, d, e\}$$

$$T = \{e\}$$

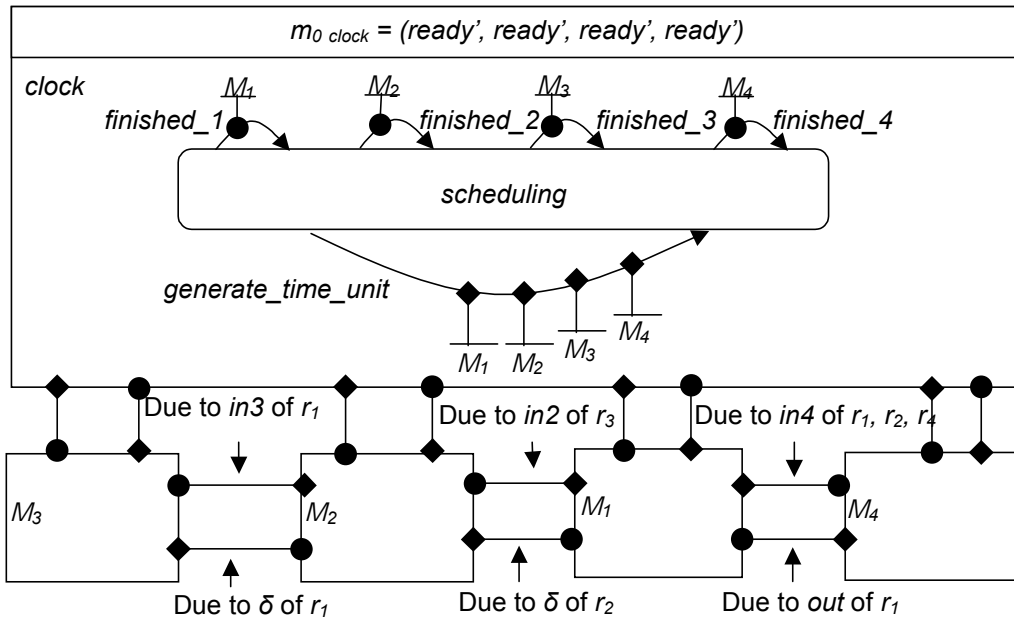
$$\mu = [1[2[3]3]_2[4]4]_1$$

$w_1 = aac, R_1 = \{r_1:a \rightarrow c_{in4}, r_2:a \rightarrow b_{in4}, r_3:c \rightarrow a_{in2}b, r_4:dd \rightarrow a_{in4}\}, \rho_1 = \{r_1 > r_3, r_2 > r_3\}$   
 $w_2 = a, R_2 = \{r_1:a \rightarrow a_{in3}, r_2:ac \rightarrow \delta\}, \rho_2 = \emptyset$   
 $w_3 = cd, R_3 = \{r_1:a \rightarrow \delta\}, \rho_3 = \emptyset$   
 $w_4 = \varepsilon, R_4 = \{r_1:c \rightarrow d_{out}, r_2:b \rightarrow d\}, \rho_4 = \emptyset$



**Fig. 3.** A P system consisting of four membranes.

According to the transformation rules 1,2 and 3 the communicating X-machine system consists of four X-machines  $M_1, M_2, M_3$  and  $M_4$ , which communicate between them as well as with clock machine as shown in figure 4. This figure also shows the model of the clock machine.



**Fig. 4.** The structure of the communicating X-machine that models a P system. The memory attached to the clock is the initial memory value  $m_{0\ clock}$ .

Transformation rules 4 to 9, define the four X-machines  $M_1, M_2, M_3$  and  $M_4$ . For example, figure 5 shows the next state function  $F_{M_2}$  of  $M_2$ , i.e. states, functions, and their communication with the rest of the machines. The memory shown is the initial memory  $m_{0M_2} = (a, \varepsilon, active)$ .

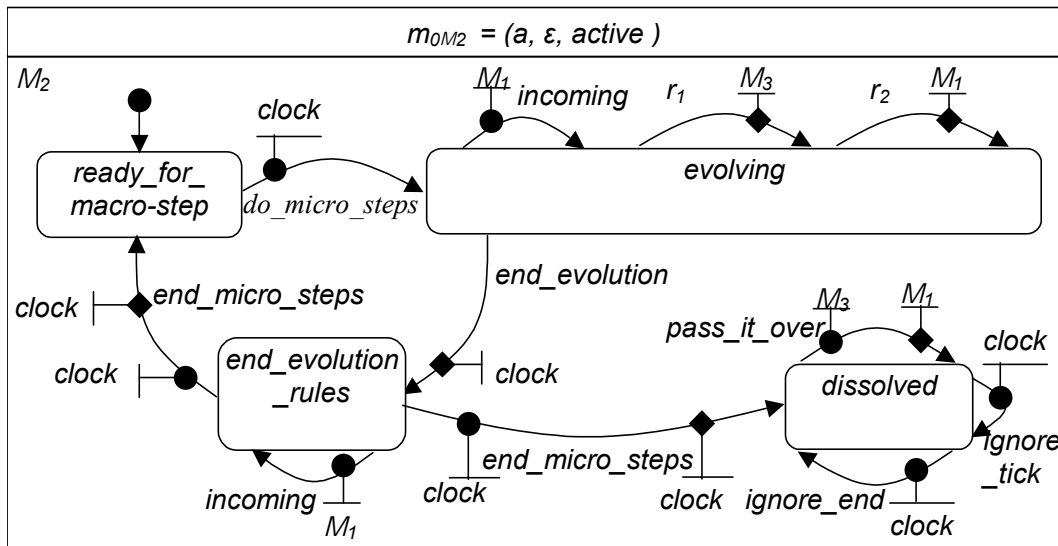


Fig. 5. The next state partial function of the communicating component X-machine  $M_2$ .

## 4. Model Checking

Having designed a model sufficient to simulate a P system, it would be desirable to verify whether this model in a given formal framework exhibits some specific behaviour, i.e. under all circumstances, some required properties are true in that model. Verification of such models can be achieved either using formal reasoning or by explicit model checking. Following the first approach, work reported in (Perez-Jimenez and Sancho-Caparrini, 2002) attempts to verify that a P system generates squares of natural numbers by using formal reasoning. Having formalised the syntax of the P system, the process of verification is based on the analysis of the content of every membrane in every computation that can be obtained in this P system (Sancho-Caparrini, 2002). On the other hand, *model checking* is a formal verification technique, which determines whether given properties of a system are satisfied by a model (Clarke et al., 1999). A model checker takes a model and a property as inputs and outputs either a claim that the property is true or a counterexample falsifying the property. The verification can be accomplished using an efficient breadth first search procedure, which views the transition system as a model for the logic and determines if the specifications are satisfied by that model. Also, model checking is formal as opposed to simulation, which may reveal inconsistencies and misconceptions in a model, but does not guarantee completeness of the model with respect to requirements.

In order to apply model checking to X-machines, temporal logic is extended with memory quantifier operators:

- $M_x$ , for all memory instances, and
- $m_x$ , there exist memory instances,

These two operators together with the two basic path (transition sequence) quantifiers:

- $A$ , meaning "for all paths", and
- $E$ , meaning "there exists a path",

and the five basic  $CTL^*$  operators (Emerson and Halpern, 1986):

- $X$  (next time) requires that a property holds in the next state,
- $F$  (eventually) requires a property to hold at some state on a path,
- $G$  (always) requires a property to hold at every state on a path,
- $U$  (until) requires a property  $p$  to hold in a path until another property  $q$  holds,
- $R$  (release) as a dual operator of  $U$ ,

can verify the model expressed as an X-machine against the requirements, since it can prove that certain properties, which implicitly reside in the memory of X-machine are true (Eleftherakis et al., 2001). The resulting logic is called  $XmCTL$ . In all the above, a path is a sequence of transitions between states of some model (denoted as  $state \rightarrow /function \rightarrow state' \dots$ ). The new syntax and semantics facilitate model checking of X-machines in two ways:

- expressiveness suited to the X-machine model, and
- effective reduction of the state space through selective refinement of the original X-machine model.

There are plenty of interesting properties that one would like to check in a communicating X-machine that models a P system. These are properties that deal each time with the model at hand.

Here are some properties with respect to the P-system example described in the previous section ( $mem_1, mem_2, mem_3$  indicate elements of the memory tuple):

- Property 1: There exists a path in which for all states and for all memory instances the object  $a$  is in the multiset of current objects:

$$EG M_x [a \subseteq mem_1]$$

For example in the case of  $evolving \rightarrow /r_1 \rightarrow evolving \rightarrow /r_2 \rightarrow evolving$  path,  $a$  is in  $mem_1$  in all the states.

- Property 2: There exists a path in which for all states and for at least one memory instance the object  $a$  is in the multiset of current objects:

$$EG m_x [a \subseteq mem_1]$$

For example in the case of  $evolving \rightarrow /r_1 \rightarrow evolving \rightarrow /r_2 \rightarrow evolving \rightarrow /r_3 \rightarrow evolving$  path,  $a$  is in  $mem_1$  in the first three states but not in the last one.

- Property 3: There exists a path in which the state of the X-machine eventually becomes *dissolved*:

$$E [M_x (q \neq dissolved) \cup M_x (q = dissolved)]$$

- Property 4: For all paths and for all states in that path, all memory instances satisfy that either the current objects multiset is not empty or the buffer is not empty or the state is *dissolved*:

$$AG M_x [mem_1 \neq \varepsilon \vee mem_2 \neq \varepsilon \vee q = dissolved]$$

In other words, the X-machine is either at *dissolved* state or in any other state that one of the memory elements (current objects or buffer) are not empty.

- Property 5: For all paths there exists a state where all memory instances satisfy that this state is *ready\_for\_macro-step* and the buffer is empty:

$$AF M_x [mem_2 = \varepsilon \wedge q = ready\_for\_macro\_step]$$

- Property 6: There exists a path and there exists a state in this path where for at least one memory instance the state is *evolving* and the buffer is empty and for every path starting from this state the next state is *evolving* or *end\_evolution\_rules*:

$$EF m_x [(mem_2 = \varepsilon \wedge q = evolving) \wedge AX M_x (q = evolving \vee q = end\_evolution\_rules)]$$

In other words, in an X-machine, eventually there will not be any applicable rules in the *evolving* state.

A constraint such as that expressed by Property 4 checks that a given component exhibits a consistent behaviour whereas Property 5 checks that an X-machine component  $M_i$  that is active should reach the state *ready\_for\_macro-step* when all the current objects have been transferred into the first component of the memory.

## 5. Conclusions

The power of an X-machine model, namely communicating X-machine systems (Kefalas et al., 2003b) is proven when simulating the behaviour of P systems with symbol-objects. Additionally by considering into play a model checking framework the communicating X-machine system exhibits some new capabilities regarding testing the system against some given properties. Compared with other similar approaches (Aguado et al., 2001) this model of communicating X-machines is more efficient than X-machine approach : if one having  $m$  regions and expecting at most  $p$  evolution rule occurrences to be applied in each region, then in the current model one macro step takes  $2p+2$  units ( $p$  to process all objects,  $p$  to get all incoming multisets and 2 to synchronize with clock machine by using *end\_evolution* and *end\_micro\_steps*) whereas in the case of simple X-machines (Aguado et al, 2001) it takes at least  $pm$  units necessary only to simulate every evolution rule in each region of the P system. On the other hand its complexity is similar with that of other types of communicating X-machine systems (Aguado et al., 2001), but with a potential ability of modelling cell division due to its flexible way of aggregating components into a communicating distributed system without altering their initial definition. In this case the definition of a communicating X-machine should be extended to deal with a kind of multisets of components rather than a set of components, because cell division relies on multiplying existing defined elements (Paun, 2001). The major problem in simulating a P system in the context of a communicating X-machine system consists in providing a mechanism to synchronize the components. In (Aguado et al., 2001) the communication matrix has been a very effective way to synchronize the parts of the system, whereas in the current context an additional component machine, called the 'clock X-machine component' (Transformation Rule 3) which sends specific tick messages marking the macro-steps of P system evolution is introduced. Once this message is received by the other component machines, they may (re)start working independently until a new tick message is received.

This approach leads to a natural question of whether or not is worth considering a more complex model that combines the features of both P systems (hierarchical organization and parallel evolution of components) and X-machines (state based model reacting to stimuli coming from environment) and provides a more coherent view of modelling alive organisms at a cellular level. An initial attempt has been made by considering 'molecular X-machines (Eilenberg machines)' (Balancescu et al., 2002), but much more work is expected in order to reveal its appropriateness for both computational aspects and modelling purposes.

## Acknowledgements

The authors would like to thank the referees for their very useful and constructive comments.

## References

- Adleman, L.M., 1994. Molecular computation of solutions to combinatorial problems. *Science* 226, pp.1021-1024.
- Aguado, J., Balanescu, T., Cowling, T., Gheorghe, M., Holcombe, M., Ipate, F., 2001. P systems with replicated rewriting and stream X-machines (Eilenberg machines). *Fundamenta Informaticae* 49, pp. 1-17.
- Balanescu, T., Cowling, A.J., Georgescu, H., Gheorghe, M., Holcombe, M., Vertan, C., 1999. Communicating stream X-machines systems are no more than X-machines. *J. of Universal Computer Sci.*, 5 (9), pp. 494-507.
- Balanescu, T., Gheorghe, M., Holcombe, M., Ipate, F., 2002. A variant of EP systems. Pre-proc. Workshop on Membrane Computing, Curtea de Arges, Romania, Publication No.1, MolCoNet project – IST – 2001 – 32008, pp. 57 – 64.
- Banatre, J-P., Le Metayer, D., 1990. The gamma model and its discipline of programming. *Sci. Comp. Programming* 15, pp. 55-77.
- Barnard, J., 1998. COMX: a design methodology using communicating X-machines. *Journal of Information and Software Technology*, 40, pp. 271-280.
- Berry, G., Boudol, G., 1992. The Chemical abstract machine. *Theor. Comput. Sci.* 96, pp. 217-248.
- Chaitin, G. J., 2002. Meta-mathematics and the foundations of mathematics. *Bulletin of the EATCS* 77, pp. 167-179.
- Clarke E.M., Grumberg O., Peled. D., 1999, *Model Checking*. MIT Press, Cambridge, Massachusetts
- Clarke, E.M., Emerson, E.A., Sistla, A.P., 1986. Automatic verification of finite state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8 (2), pp. 244–263.
- Clark, L., Paton, R., 1998. Towards computational models of chemotaxis in *Escherichia coli*. In Holcombe, M., Paton, R. (Eds), *Information Processing in Cells and Tissues*. Plenum Press, pp. 39-46.
- Cowling, A.J., Georgescu, H., Vertan, C., 2000. A structured way to use channels for communication in X-machines systems. *Formal Aspects of Computing*, 12, pp. 485-500.
- Duan, Z.H., Holcombe, M., Linkens, D.A., 1995. Modelling of a soaking pit furnace in hybrid machines. *Syst. Analysis, Modelling, Simulation* 18, pp. 153-157.
- Eilenberg, S., 1974. *Automata, languages and machines*, volume A. Academic Press.
- Eleftherakis, G., Kefalas, P., Sotiriadou, A., 2001. XmCTL: Extending temporal logic to facilitate formal verification of X-machines models. *Annales of the University of Bucharest, Mathematics-Informatics*, Year L, pp.79-95.
- Emerson, E.A., Halpern, J.Y., 1986. Sometimes and not never revisited: On branching time versus linear time. *Journal of the ACM*, 33, pp. 151-178.
- Gheorghe, M., Holcombe, M., Kefalas, P., 2001. Computational models of collective foraging. *BioSystems* 61, pp. 133-141.
- Holcombe, M., 2001. Computational models of cells and tissues: Machines, agents and fungal infection. *Briefings in Bioinformatics* 2, pp. 271-278.
- Ipate, F., Holcombe, M., 1996. Another look at computability. *Informatica*, 20, pp.359-372.
- Kauffman, S. A., 1993. *The origins of order. Self-organization and selection in evolution*. Oxford University Press.
- Kefalas P., 2002. Formal modelling of reactive agents as an aggregation of simple behaviours. *Lecture Notes in Artificial Intelligence* 2308 (I.P.Vlahavas and C.D.Spyropoulos eds.), Springer-Verlag, pp. 461-472.
- Kefalas, P., Holcombe, M., Eleftherakis, G., Gheorghe M., 2003a. A formal method for the development of agent based system. In: *Intelligent Agent Software Engineering* (V.Plekhavona ed.), Idea Group Publishing, pp. 68-98 (to appear).
- Kefalas, P., Eleftherakis, G., Kehris, E., 2003b. Communicating X-machines: from theory to practice. *Lecture Notes in Computer Science* (to appear).
- Paun, Gh., 1998. Computing with membranes. *J. Comput. Sys. Sci.* 61 (1) (2000), pp.108-143 (circulated since November 1998 as TUCS Report No 208, [www.tucs.fi](http://www.tucs.fi)).

- Paun, Gh., 2001. From cells to computers: computing with membranes (P systems). *BioSystems* 59, pp. 139-158.
- Paun, Gh., 2002. *Membrane Computing. An Introduction*. Springer, Berlin.
- Paun, Gh., Rozenberg, G., Salomaa, A., 1998. *DNA Computing. New Computing Paradigms*. Springer, Berlin.
- Perez-Jimenez, M.J., Sancho-Caparrini, F., 2002, Verifying a P system generating squares. *Romanian J. Inform. Sci. and Technology*, 5, 2-3, pp.295-305.
- Sancho-Caparrini, F., 2002. *Verifying of programs in unconventional computing models*. PhD Thesis, Sevilla University.