Nat Comput (2009) 8:817–832
DOI 10.1007/s11047-008-9103-y

Transforming communicating X-machines into P systems

Petros Kefalas · Ioanna Stamatopoulou · Ilias Sakellariou · George Eleftherakis

Published online: 1 October 2008

Abstract Tissue P systems (tPS) represent a class of P systems in which cells are arranged in a graph rather than a hierarchical structure. On the other hand, communicating X-machines (XMs) are state-based machines, extended with a memory structure and transition functions instead of simple inputs, which communicate via message passing. One could use communicating XMs to create models built out of components in a rather intuitive way. There are investigations showing how various classes of P systems can be modelled as communicating XMs. In this paper, we define a set of principles to transform communicating XMs into tPS. We describe the rules that govern such transformations, present an example to demonstrate the feasibility of this approach and discuss ways to extend it to more general models, such as population P systems, which involve dynamic structures.

Keywords P systems · X-machines · Nature-inspired computation · Multi-agent systems for natural phenomena

1 Introduction

In the last years, a nature-inspired computational paradigm, called P systems Păun (2000), abstracting from the structure and functionality of the living cell has been intensively and

P. Kefalas (✉) · I. Stamatopoulou · G. Eleftherakis
Department of Computer Science, CITY College, 13 Tsimiski street, 54624 Thessaloniki, Greece
e-mail: kefalas@city.academic.gr

I. Stamatopoulou
e-mail: istamatopoulou@seerc.org

G. Eleftherakis
e-mail: eleftherakis@city.academic.gr

I. Sakellariou
Department of Applied Informatics, University of Macedonia, 156 Egnatia street, 54006 Thessaloniki, Greece
e-mail: iliass@uom.gr

extensively studied and numerous variants have been considered. They have been investigated for their computational power and complexity aspects Păun (2002), used to solve hard problems, model various biological systems and provide solutions to questions in different areas. Occasionally, some attempts have been made to use P systems towards modelling swarm-based multi-agent systems (Stamatopoulou et al. 2005a), in order to take advantage of the structure and reconfiguration features of P systems, such as cell death, cell division, reconfiguration of structure etc. The main problem which appears in such modelling activity is that the model resulting for the object interaction within a cell is not always easy to develop. This drawback may be overcome by using state-based models that provide the necessary "intuitiveness" to describe the behaviour of the system components or agents. For instance, communicating X-machines (XMs) have been used as a suitable paradigm of modelling agent based specifications (Kefalas et al. 2003c).

A natural consequence of the above complementary features exhibited by these models, is to either try to combine both formalisms (Stamatopoulou et al. 2007a, 2007b) or to transform one formalism to another. The current trend in P system community research shows more interest in connecting this model with other computational approaches—Petri nets (Klein and Koutny 2007), process algebra (Ciobanu and Aman 2007), cellular automata (Corne and Frisco 2007) etc. In the past, relationships between some classes of P systems and communicating XMs have been investigated. Especially transformations of P systems into communicating XMs have been particularly considered (Kefalas et al. 2003a). Most of these studies have been focusing in translations between these models in order to make use of various strengths offered by different formalisms—model checking for process algebra, invariants for Petri nets, or testing methods for XMs.

This paper presents some principles for transforming communicating XMs into Tissue P systems (tPS), a special class of P systems. Section 2 provides the basic background on XM modelling and communicating XMs accompanied by an example. The definition of tPS and the computational model associated with it are presented in Sect. 3. In Sect. 4, we demonstrate how a transformation from one model to another is feasible and apply the guidelines to the particular example introduced. We then discuss in detail the rationale of using communicating XMs as a starting point as well as how the resulting tPS model could be enhanced further to take advantage of the dynamic features of another class of P systems, namely population P systems (PPS). We conclude by discussing the ideas behind the transformation and the issues that need further consideration.

2 State-based modelling with X-machines

2.1 X-machines

An XM is a state-based computational model introduced by Eilenberg (Eilen-berg 1974). It is widely accepted as a suitable model to formally specify the components of a system. Stream X-machine (sXM) models, in particular, were found to be well-suited for describing reactive systems (Holcombe and Ipate 1998). Since then, valuable findings regarding the use of this model as a formal notation that contributes toward the specification, verification and testing of software systems, have been reported (Kefalas et al. 2003b; Eleftherakis 2003; Holcombe and Ipate 1998). A sXM model consists of a number of states and also has a memory, which accommodates mathematically defined data structures. The transitions between states are labelled by functions. More formally, a *deterministic sXM* is a tuple (Σ, Γ, Q, M, Φ, F, A, q_0, m_0) where:

- Σ and Γ are the *input* and *output* alphabets respectively;
- Q is the finite set of *states*;
- M is the (possibly) infinite set called *memory*;
- Φ is a set of partial functions φ, called *processing functions*, that map an input and a memory state to an output and a possibly different memory state, $\varphi{:}\Sigma \times M \rightarrow \Gamma \times M$;
- F is the next state partial function, F: $Q \times \Phi \rightarrow Q$, which given a state and a function from the set Φ determines the next state. F is often described as a *state transition function*;
- A, a subset of Q, is the set of *final states*;
- q_0 and m_0 are the *initial state* and *initial memory*, respectively.

In the example used in this paper $A = Q$ and for this reason A will not be used.

2.2 Example of a sXM: Japanese bees and hornets

The example that we chose to present is a representative case study of modelling natural phenomena, which exhibit highly dynamic behaviour, as multi-agent systems. Hornets are the major enemy of bees and can slaughter them in thousands in less than an hour. Japanese bees have the ability to survive a hornet attack in their hive by employing a unique attack strategy: they surround the hornet and by moving their bodies increase the temperature of the environment so that it is intolerable by the hornet. It is interesting that bees can survive up to 49°C while hornets cannot stand temperature that exceeds 47°C. With such an attack strategy, the attacking hornet is literally roasted by the bees.

The overall model consists of two sXM models, one for the Japanese bee and one for the hornet. The states in which a bee can be are:

$$Q = \{work\_in\_hive, approaching\_hornet, attacking, killed\}$$

A bee can perceive an input stating the actual stimulus (including temperature) and the position it is coming from in order to trigger functions in Φ, so:

$$\begin{aligned}\Sigma = &\{(Percept, Pos) | Percept \in \{freePos, lethalBite, hornet, hornetInHive,\\ &attackingBee, deadHornet\} \cup Temperature\}\end{aligned}$$

where $Temperature \in \mathbb{R}+$ and $Pos \in \mathbb{N}_0 \times \mathbb{N}_0$. A bee can output messages depending on the function in Φ triggered, for instance:

$$\begin{aligned}\Gamma = &\{working, saw\_hornet, learned\_about\_hornet, ready\_to\_attack, in\_formation,\\ &dead\_bee, saw\_dead\_hornet, learned\_about\_dead\_hornet\} \cup Temperature\end{aligned}$$

The memory holds the current bee position, whether a hornet is in the hive or not, and the possible position of the hornet, i.e.

$$M = \{(BeePos, Alert, HornetPos) | Alert \in \{noAlert, hiveInDanger\}$$

where $BeePos, HornetPos \in \mathbb{N}_0 \times \mathbb{N}_0\}$.

An instance of a bee may have an initial memory m_0 = ((10,5),*noAlert*,(*null*,*null*)) and an initial state q_0 = *work_in_hive*. The set Φ of the bee sXM model consists of a number of functions, as for example:

$$\begin{aligned}&perceiveDanger((hornetInHive, HornetPos), (Pos, noAlert, null)) =\\&\quad(learned\_about\_hornet, (Pos, hiveInDanger, HornetPos));\\&produceHeat((Temperature, Pos), (CurrentPos, hiveInDanger, HornetPos)) =\\&\quad\text{if } Temperature < 49 \text{ then}\\&(Temperature + 0.1, (CurrentPos, hiveInDanger, HornetPos));\\&attackedByHornet((lethalBite, Pos), (CurrentPos, hiveInDanger, HornetPos)) =\\&\quad(dead\_bee, (CurrentPos, noAlert, HornetPos)).\end{aligned}$$

Similarly, the states in which a hornet can be are:

$$Q = \{moving\_around, killing\_bees, dead\_by\_heat\}$$

The input and output sets are:

$$\begin{aligned}\Sigma &= \{(Percept, Pos) | Percept \in \{freePos, bee\} \cup Temperature\}\\\Gamma &= \{searching, killed\_bee, dead\_hornet\}\end{aligned}$$

respectively. The memory holds the current hornet position, i.e.

$$M = \{HornetPos | HornetPos \in \mathbb{N}_0 \times \mathbb{N}_0\}$$

An instance of a hornet may have m_0 = (20,18) and $q_0 = moving\_around$. The set Φ of the hornet sXM model consists of a number of functions, as for example:

$$\begin{aligned}&killBee((bee, Pos), CurrentPos) = (killed\_bee, Pos);\\&killed((Temperature, Pos), CurrentPos) =\\&\text{if } Temperature \geq 47 \text{ then } (dead\_hornet, CurrentPos).\end{aligned}$$

The state transition diagrams of both the bee and the hornet are shown in Fig. 1.

2.3 Communicating X-machines

In addition to having stand-alone sXM models, communication is feasible by exchanging messages between components' processing functions. The structure of a communicating

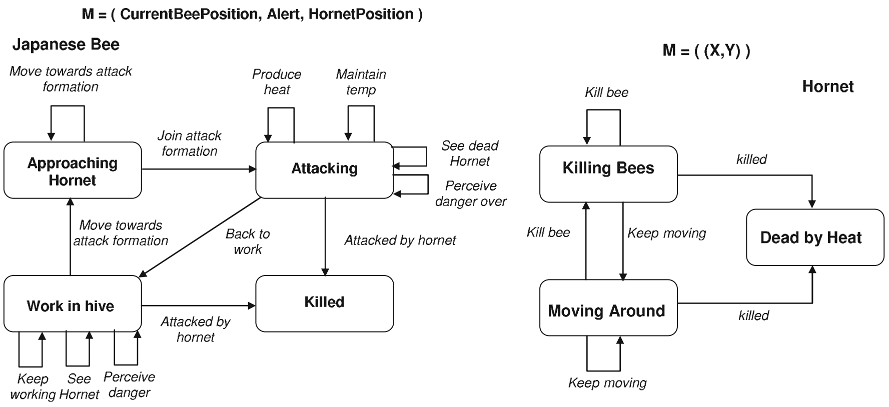


Fig. 1 The transition diagrams *F* of the sXM models for a bee and a hornet

X-machine system (CXS) is defined as the graph whose nodes are the components (machines) and edges are the communication channels among them. A CXS is a tuple:

$$\mathcal{Z} = ((C_i)_{i=1,\ldots,n},\ T)$$

where:

- C_i is the *i*th *communicating X-machine component*, and
- T is a set of *transformation functions* $T_{q_i,\varphi_i,q_j,\varphi_j}$; each such function relates to two components C_i and C_j,

$$T_{q_i,\varphi_i,q_j,\varphi_j} : \Sigma_i \times \Gamma_i \times M_i \rightarrow \Sigma_j,$$

where q_i, φ_i, Σ_i, Γ_i, M_i are from C_i and q_j, φ_j, Σ_j are from C_j, and transforms an input from Σ_i, an output from Γ_i and a memory from M_i produced by φ_i which is triggered in q_i, into an input from Σ_j that will be processed by φ_j starting in q_j.

A *communicating X-machine component* (CXM) can be derived by incorporating into a sXM information about how it is to communicate with other CXMs that participate in the system. In order to define the communication interface of a sXM, two things have to be stated: (a) which of its functions receive their inputs from which machines, and (b) which of its functions send their outputs to other machines.

Graphically on the state transition diagram we denote the acceptance of input from another component by a solid circle along with the name C_i of the CXM that sends it. Similarly, a solid diamond with the name C_k denotes that output is sent to the CXM C_k. An abstract example of the communication between two CXMs is depicted in Fig. 2 and their formal definition can be found in (Stamatopoulou et al. 2007a).

A CXS starts in an initial configuration with each CXM C_i, in its initial state $q_{0,i}$, with initial memory value $m_{0,i}$ and having an initial multiset of input values from Σ_i. We will consider both that the execution time of every processing function is the same or that they have different execution times. In the first case we can define a *maximal parallelism mode*, where a maximal number of components executes one function at the same time. In the second case we can define an *arbitrary parallelism* implying that at each step only an arbitrary number of components execute one function. In any of the two modes, a computation consists of an arbitrary number of computation steps starting from an initial configuration and ending when every component has reached a final state or has consumed all its inputs. The result, the multiset of output symbols, is read from a distinguished component, C_{i_o}. For a CXS $\mathcal{Z}$ the multiset computed using maximal and arbitrary parallelism is denoted by $MS_m(\mathcal{Z})$ and $MS_a(\mathcal{Z})$, respectively.

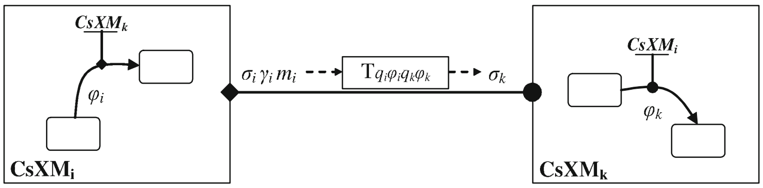


Fig. 2 Abstract example of the communication between two communicating X-machines

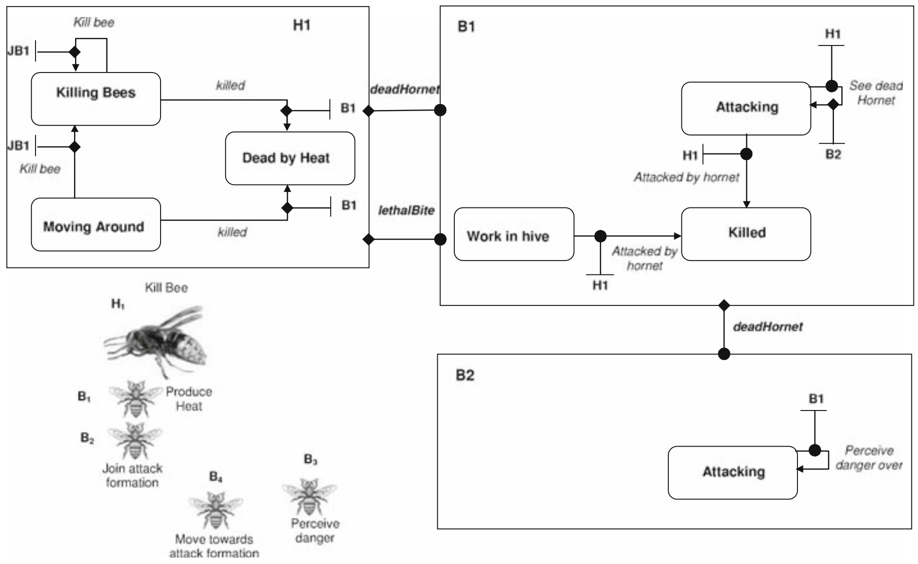


Fig. 3 A CXS consisting of two bees B1 and B2 attacking a hornet $H1$ (not including the communication between $B3$ and $B4$)

2.4 Example of CXS: two bees attacking a hornet

Consider the instance of the hive as shown at the bottom left of Fig. 3. There is a hornet $H1$, bee $B1$ is next to the hornet, $B2$ is next to $B1$ in the attack formation, $B3$ is approaching the attack formation passing by $B4$ which is informed about the existence of a hornet in the hive. Annotations show the sXM function triggered at the current state of the world. The five instances $H1$ and $B1$, $B2$, $B3$, $B4$ may form a CXS, part of which is illustrated in Fig. 3. We omitted the functions and states that are not relevant to the communication for reasons of clarity in exposition. When a hornet bites a bee the function *killBee* of $H1$ sends a message to $B1$, through a transformation functions like:

$$T_{moving\_around,\, killBee,\, attacking,\, attackedByHornet}(killed\_bee, (Pos)) = (lethalBite, Pos)$$

Function *attackedByHornet* in $B1$ accepts the messages sent by *killBee* as inputs. The rest of the functions not annotated with receive or send obtain their input from their associated multiset of inputs and send their output to the corresponding multiset. Similarly, when a hornet is dead, functions of $B1$ send messages to $B2$, through transformation functions like:

$$T_{attacking,\, see\_dead\_hornet,\, attacking,\, perceive\_danger\_over}$$
$$(saw\_dead\_hornet, (BeePos, Alert, HornetPos)) = (deadHornet, HornetPos)$$

3 Tissue P systems

The model of *Tissue P system* (tPS for short) (Păun 2002) has been introduced as a generalisation of the P system model with the aim of extending the cell-like approach exhibited by the usual model to the context of tissues and other multicellular organisms. A tPS is a network of membranes that work in parallel and two arbitrary membranes communicate only when a connection exists between them. Formally a tPS is a construct $\mathscr{P} = (V, V_t, g, M_1, \ldots, M_n, R_1, \ldots, R_n, i_0)$ where:

- V is a finite alphabet of symbols called *objects*;
- V_t is a subset of V containing *terminal objects*;
- g is a subset of $\{1,2,\ldots,n\} \times \{1,2,\ldots,n\}$, specifying the *links* between membranes;
- M_i, for each $1 \leq i \leq n$, is a multiset over V defining the *initial content* of the ith membrane;
- R_i, for each $1 \leq i \leq n$, is a finite set of *rules* associated with membrane i, dealing with communication, object transformation, etc.;
- i_0 defines the membrane where the *result* is obtained.

The rules of each R_i, $1 \leq i \leq n$, set are either *rewriting rules*, $x \rightarrow y$, where x, y are multisets over V, or *communicating rules*, $x \rightarrow y(z_1)_{j_1} \ldots (z_r)_{j_r}$, where x, y, z_k, $1 \leq k \leq r$, are multisets and z_k is sent to membrane j_k.

A computation in a tPS consists of a sequence of computation steps; it starts with the initial multisets and stops when no rule is applicable. The result is represented by the multiset computed in i_0 and is denoted by $M(\mathscr{P})$.

4 Transforming communicating X-machines into P systems

The question we investigate is under what circumstances generic guidelines or principles for transforming communicating XMs to some classes of P systems exist. We are dealing with two different methods that possess different characteristics. CXMs provide a straightforward and rather intuitive way for dealing with a component's behaviour and, in general a CXS computation relies on different execution time slots for its components' functions. P systems' rules specifying the behaviour of the individual membranes are simple rewriting mechanisms, which are not always intuitive in modelling in general, but are very appropriate for specifying certain classes of systems, like chemical interactions, signalling pathways, etc. Finally, a P system computation follows the maximal parallelism execution mode. In this approach we consider a specific class of P systems, namely tPS.

The rationale behind such transformation is to automatically or semi-automatically produce tPS models. This will have the advantage of using existing CXS models whose components have been thoroughly verified and tested. The resulting tPS model will have cells, objects, various rules (transformation and communication rules as it will be seen latter on). The model can then be simulated using in-house produced tools and enriched with other aspects like cell differentiation, death, birth and bond-making rules (see Sect. 5) and transformed into other more suitable classes of P systems, like PPS (Bernardini and Gheorghe 2004). The transformation process will be addressed in stages and special sections will be allocated in this respect.

4.1 CXM components and tPS membranes

Every CXM component, C_i, will be associated with a membrane, i, with objects, transformation and communication rules. We will refer to these membranes in the next sections.

4.2 Objects in membranes

We consider that all the objects that are used inside the tPS membranes are of the form (*tag*: *value*), where *tag* is a descriptor that gives the type of the object and is introduced to help identifying various objects in further steps, whereas *value* defines an element of this type. The following objects are defined:

- the states in Q generate objects of the form $(state : q)$, where $q \in Q$;
- the memory values in M yield objects of the form $(memory : m)$, where $m \in M$;
- the inputs from Σ produce objects of the form $(input : \sigma)$, where $\sigma \in \Sigma$;
- objects of the form $(output : \gamma)$, where $\gamma \in \Gamma$, correspond to the outputs of the CXM component.

4.3 Initial multisets

For every CXM, C_i, an initial multiset of input symbols, $\sigma_1 ... \sigma_p$, is available; thus the corresponding membrane, i, will consists of the following initial multiset:

$$(state : q_{0,i})(memory : m_{0,i})(input : \sigma_1)\ldots(input : \sigma_p)$$

where $q_{0,i}$, $m_{0,i}$ are the initial state and initial memory value of C_i.

There is one issue that arises by the fact that we derive multisets for the input objects of the tPS, as this is how they are defined, considering that sXMs are defined to accept their input from a stream instead. Note that the computation of a sXM is deterministic in contrast to that of a tPS. Indeed, by considering multisets of input objects, the transformation results in a tPS with many possible computations (depending on the order in which the input symbols are consumed) one of which will be equivalent to the computation of the original sXM model. This issue is further discussed in Sect. 5.4.

4.4 Transformation rules for processing functions

For every CXM, C_i and for every function $\varphi_i : \Sigma_i \times M_i \rightarrow \Gamma_i \times M_i$, which is not involved in any communication, such that $\varphi_i(\sigma_i, m_i^1) = (\gamma_i, m_i^2)$, where $m_i^1, m_i^2 \in M_i$, $\sigma_i \in \Sigma_i$, $\gamma_i \in \Gamma_i$, and for every $q_i^1, q_i^2 \in Q_i$ such that $F_i(q_i^1, \varphi_i) = q_i^2$, a rule is created in membrane i:

$$(state : q_i^1)(memory : m_i^1)(input : \sigma_i) \rightarrow (state : q_i^2)(memory : m_i^2)(output : \Gamma_i)$$

4.5 Transformation and communication rules for processing functions

For every CXM, C_i, and

- every function $\varphi_i{:}\Sigma_i \times M_i \rightarrow \Gamma_i \times M_i$ which is involved in a communication with $\varphi_j{:}\Sigma_j \times M_j \rightarrow \Gamma_j \times M_j$ *of CXM* C_j, such that
  - $\varphi_i(\sigma_i, m_i^1) = (\gamma_i, m_i^2)$, where $m_i^1, m_i^2 \in M_i$, $\sigma_i \in \Sigma_i$, $\gamma_i \in \Gamma_i$
  - $\varphi_j(\sigma_j, m_j^1) = (\gamma_j, m_j^2)$, where $m_j^1, m_j^2 \in M_j$, $\sigma_j \in \Sigma_j$, $\gamma_j \in \Gamma_j$
- every $q_i^1, q_i^2 \in Q_i$ such that $F_i(q_i^1, \varphi_i) = q_i^2$
- every $q_j^1, q_j^2 \in Q_j$ such that $F_j(q_j^1, \varphi_j) = q_j^2$
- a transformation $T_{qi,\ \varphi i,\ qj,\ \varphi j}(\gamma_i, m^2{}_i) = \sigma_j$

a rule which rewrites ($state$:q_i^1)($memory$:$m_i{}^1$)($input$:σ_i) by ($state$:$q_i{}^2$)($memory$:$m_i{}^2$) ($output$:γ_i) in membrane i and sends ($input$:σ_j) to membrane j, is constructed:

$$(state : q_i^1)(memory : m_i^1)(input : \sigma_i) \rightarrow$$
$$(state : q_i^2)(memory : m_i^2)(output : \Gamma_i)(input : \sigma_j)_j$$

4.6 Main Result

Given the constructions described in Sects. 4.1–4.5, we can now formulate the following result:

Theorem 1 *For any communicating X-machine system, $\mathscr{Z}$, having all the memory components as finite sets, there are tissue P systems, $\mathscr{P}_1, \mathscr{P}_2$, such that $MS_m(\mathscr{Z}) = M(\mathscr{P}_1)$, $MS_a(\mathscr{Z}) = M(\mathscr{P}_2)$.*

Proof Let the CXS

$$\mathscr{Z} = ((C_i)_{i=1,\ldots,n}, T)$$

with the sXM components

$$C_i = (Upsigma_i, \Gamma_i, Q_i, M_i, \Phi_i, F_i, A_i, q_{0,i}, m_{0,i}), 1 \leq i \leq n,$$

and $MS_m(\mathscr{Z})$ and $MS_a(\mathscr{Z})$, the multisets computed using maximal parallel mode and arbitrary parallel mode, respectively, in component i_0. The following tPSs are considered:

$$\mathscr{P}_j = (V_j, V_{t,j}, g, M_{1,j}, \ldots, M_{n,j}, R_{1,j}, \ldots, R_{n,j}, i_0), 1 \leq j \leq 2.$$

Each $\mathscr{P}_j, j = 1, 2$, has n membranes that are built according to the construction in Sect. 4.1. The set g is the same for both tPSs and a tuple (i, k) belongs to g if and only if there is a message sent from a processing function in C_i to a function in C_k. The sets V_1, V_2 contain the union of all the objects mentioned in Sect. 4.2 for all the CXM components and $V_{t,1}$,$V_{t,2}$ contain the objects obtained from output symbols, ($output$:out),$out \in \Gamma_i$.

The multisets $M_{i,1}$, $M_{i,2}$, $1 \leq i \leq n$, contain the objects introduced in Sect. 4.3 and obtained from the input symbols of C_i, initial state and initial memory value.

The sets $R_{i,1}$, $R_{i,2}$, $1 \leq i \leq n$, contain the rules defined in Sects. 4.4 and 4.5. If q_i used in these rules, represents a final state then a rule similar to that in Section 4.4 or 4.5 is added to both sets, but the object ($state$:q_i) is removed from the right hand side.

From the above construction it follows that if maximal parallelism is used then when a final state, q_f, is reached by the CXS components, then $\mathscr{P}_1$ will not allow any rule to be further applied, as objects ($state$:q_f) are removed from the cells, and consequently $\mathscr{Z}$ is simulated by $\mathscr{P}_1$. In order to allow $\mathscr{P}_2$ to simulate $\mathscr{Z}$ using an arbitrary parallel mode, rules $(state : q) \rightarrow (state : q), q \in Q$, will be inserted in every cell. Consequently, $MS_a(\mathscr{Z}) = M(\mathscr{P}_2)$.

4.7 Example transformation

The above example of a CXS consisting of a hornet $H1$ and two attacking bees $B1$ and $B2$ can be transformed according to the above principles as follows; there will be three cells, namely C_{H1} with the initial multiset w_{H1},C_{B1} and C_{B2} with the initial multisets w_{B1} and w_{B2} respectively. The objects which appear during computation in cell C_{H1} will be:

- ($state$: q), $q \in \{moving\_around, killing\_bees, dead\_by\_heat\}$,
- $(memory : Pos), Pos \in \mathbb{N}_0 \times \mathbb{N}_0$,
- ($input$: inp), $inp \in \{(Percept,Pos) \mid Percept \in \{freePos, bee\} \cup Temperature\}$,
- ($output$: out), $out \in \{searching, killed\_bee, dead\_hornet\}$.

Initially the objects w_{H1} are:

$$(state : moving\_around)(memory : (20, 18))(input : inp_1)\ldots(input : inp_t)$$

corresponding to the initial state, initial memory values and initial input to this machine.

The transformation rules for non-communicating functions are indicatively as follows:

$$\begin{aligned}&(state : moving\_around)(memory : Pos)(input : (freePos, NewPos)) \rightarrow\\&(state : moving\_around)(memory : NewPos)(output : searching)\end{aligned}$$

as a rule associated with *keep_moving* function.

The objects which appear during computation in cells C_{B1} and C_{B2} will be:

- ($state$: q), where $q \in \{work\_in\_hive, approaching\ \_hornet, attacking, killed\}$;
- ($memory$: ($BeePos$,$Alert$,$HornetPos$))
- ($input$: inp), where $inp \in \{(Percept, Pos)|Percept \in \{freePos, lethalBite, hornet, hornet InHive, attackingBee, deadHornet\} \cup Temperature, Temperature \in \mathbb{R}+$ and $Pos \in \mathbb{N}_0 \times \mathbb{N}_0\}$
- ($output$:out), where $out \in \{working, saw\ \_hornet, learned\_about\_hornet, ready\_to\_attack, in\_formation, dead\_bee, saw\_dead\_hornet, learned\_about\_dead\_hornet\} \cup Temperature$.

Initially the objects w_{B1} and w_{B2} are:

$$\begin{aligned}&(state : attacking)(memory : ((20, 17), hive\_in\_danger, (20, 18)))\\&\quad(input : inp_1)\ldots(input : inp_t), \text{and}\\&(state : attacking)(memory : ((20, 16), hive\_in\_danger, (20, 18)))\\&\quad(input : inp_1)\ldots(input : inp_t), \text{respectively.}\end{aligned}$$

A transformation rule associated with the *produceHeat* non-communicating function of a bee is:

$$\begin{aligned}&(state : attacking)(memory : (CurrentPos, hiveInDanger, HornetPos))\\&\quad(input : (Temperature, Pos)) \rightarrow\\&(state : attacking)(memory : (CurrentPos, hiveInDanger, HornetPos))\\&\quad(output : Temperature + 0.1),\\&\text{if } Temperature < 49\end{aligned}$$

As far as communication is concerned, in the cells C_{H1} and C_{B1} there will be some transformation rules that correspond to the communicating functions. For example in C_{H1}:

$$\begin{aligned}&((state : killing\_bees)(memory : Pos)(input : (bee, BeePos)) \rightarrow\\&(state : killing\_bees)(memory : BeePos)(input : (lethalBite, Pos))(output : killed\_bee))_{CB1}\end{aligned}$$

associated with *kill_bee* function in $H1$ machine.

5 Discussion

5.1 Modelling with sXM and tPS

So far, a set of guidelines have been presented to transform a CXS model to a tPS model and a result showing the equivalent behaviour of the two mechanisms has been proved. CXS provide the necessary modelling message passing means and computation that demonstrate the feasibility of scaling up models. However, as in tPS models, they do suffer from a major drawback: the organisational structure of the composed system is predefined and remains static throughout the computation. Although for some systems this is a virtue, for some others, such as multi-agent systems modelling natural phenomena, reorganisation (change in the network of communication between agents and change in the number of agents that participate in the system) is an important feature that should be addressed in a model.

The current result is established for P systems using rewriting and communication rules, but other types of P systems may be considered where different communication rules, like symport/antiport, may be used. Of particular interest to modelling highly dynamic systems is a type of P systems, namely PPS, in which there exist rules for cell death, cell division and differentiation as well as bond-making rules. A brief comparison between CXS, tPS and PPS is shown in Table 1. Their complementarity has led to the current investigation of transformation of CXS to tPS and eventually to PPS.

5.2 Verification of sXM and CXS models

Another important issue which outlines the rationale behind transformation from CXS to P systems is that modellers rarely use P systems for modelling the behaviour and communication between agents. The main reason for that would be the lack of expressive power, as sXM serve this need in a far better way. One of the challenges that emerge in modelling biological systems is to develop models that behave correctly. In this respect, sXM are coupled with techniques for formal verification and testing, reassuring correctness of implementation with respect to their models (Holcombe and Ipate 1998; Eleftherakis 2003). The model needs to meet the requirements and satisfy any necessary properties which are part of its design objectives. If a formal approach, like sXM, is used then all

Table 1 Comparison of features of CXS, tPS and PPS with respect to modelling

Modelling feature	CXS	tPS	PPS
Agent internal state representation	√		
Complex data structures for knowledge, messages, stimuli etc.	√		
Direct communication / Message exchange	√	√	
Non-deterministic communication		√	√
Dynamic addition and removal of agent instances			√
Dynamic communications network restructuring			√
Maximal parallelism	√	√	√
Arbitrary parallelism	√	√	√
Formal verification of individual components	√		
Test cases generation for individual components	√		
Tool support	√		√

static and dynamic analysis techniques as well as tools can be exploited in the context of biological models, thus improving the confidence of the correctness of the final model.

Model Checking is a formal verification technique which is based on the exhaustive exploration of a given state space trying to determine whether a given property is satisfied by a system. It has been very successful over the last years verifying hardware and software systems (Dwyer et al. 2007). 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. In sXM, checking whether a property p is valid in some states of the sXM means determining whether there are some states in which some memory values satisfy the property p. An extended logic, *Xm*CTL, can verify models expressed as sXM against the requirements, since it can prove that certain properties, which implicitly reside on the memory of sXM, are true (Eleftherakis and Kefalas 2008). The temporal operators used in *Xm*CTL are the usual operators of CTL with the addition of two new memory quantifiers, namely $\mathbf{M_x}$ and $\mathbf{m_x}$:

- $\mathbf{M_x}$ (for all memory instances) requires that a property holds at all possible memory instances of a sXM state.
- $\mathbf{M_x}$ (there exists a memory instance) requires that a property holds at some memory instances of a sXM state.

For example the property *"a bee might eventually be killed"* can be expressed as an *Xm*CTL formula as $\mathbf{EFM_x}(xstate = killed)$. Another example of an important property might be *"a bee should always be working until danger is detected"*. This is expressed as $\mathbf{AM_x}(\text{xstate} = \text{work\_in\_hive})\ \mathbf{Um_x}(Alert = hiveInDanger)$.

However, as the complexity increases applying formal verification to complex biological models is not possible in some cases. Furthermore, while formal verification may be possible it requires too much time and effort which makes it completely impractical. An alternative would be to utilise formal modelling and verification in the early stages of the development coupled with informal verification steps provided through simulation (animation) to discover any undiscovered flaws in later stages. The simulation is needed in order to informally verify complex models with dynamic communication which, with current techniques, cannot be formally verified. However at the same time the animation of the model is a step which provides immediate feedback to the development team and facilitates effective communication of the formal experts and the people (biologists) with no formal background. All these features make this alternative a practical approach.

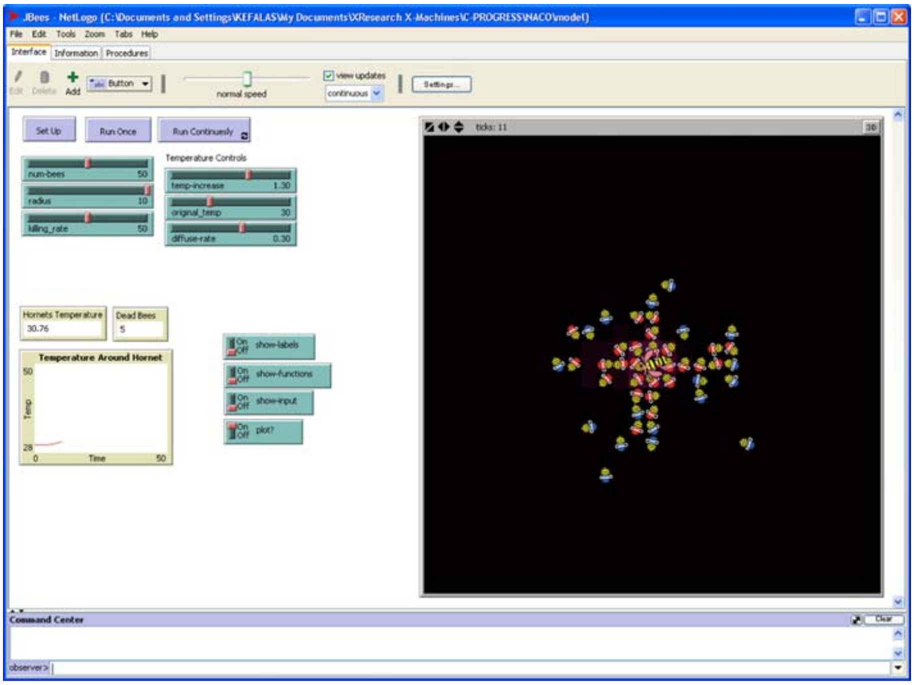
In most of the biological systems a visual animation of the model offers an intuitive way of studying the model. NetLogo (Wilensky 1999) is a modelling environment targeted for simulation of multi-agent systems that consist of a large number of agents. NetLogo offers a simple functional language, in which behaviours of agents can be encoded, and a programming environment that allows the easy creation of a GUI for a simulation supporting a great number of parameters. The environment is a suitable tool for rapid prototyping. By executing simulation scenarios and comparing the outcome with the expected behaviour of the system we are able to get immediate feedback indicating the "correctness" of the model. Taking this into account further iterations may be needed to improve the model. Our experience so far shows that there is a way to directly map CXS models to NetLogo code and we are currently investigating how this can be done semi-automatically.

In a previous work (Stamatopoulou et al. 2008; Jackson et al. 2005) we have demonstrated how a communicating X-machine specification, written in the X-machine Description Language (XMDL) (Kapeti and Kefalas 2000) can be directly ported in NetLogo and how dynamic reconfiguration rules can be implemented on top of that. The

NetLogo XMDL interpreter allows to transform effortlessly the XMDL specification to Netlogo programming structures, i.e. functions and procedures, and provide a rather accurate simulation of the specified system, under study.

The Japanese bee colony under attack by a hornet has produced a simulation shown in Fig. 4. Figure 5 shows three phases of the simulation depicting (a) bees moving towards an attack formation, (b) bees forming an attack formation, and (c) bees in full attack formation increasing the temperature and in consequence killing the hornet.

The simulation showed that the original model produced in sXM suffered from two shortcomings: (a) an extra transition (function *attackedByHornet*) was required from state *approaching_hornet* to state *killed*, and (b) the functions of the bee *produceHeat* and *maintainTemperature* should allow the bee to move to a new position closer to the hornet.


JBees - NetLogo {C:\Documents and Settings\KEFALAS\My Documents\XResearch X-Machines\C-PROGRESS\NACO\model}
File Edit Tools Zoom Tabs Help
Interface Information Procedures
Edit Delete Add Button
normal speed
view updates
continuous
Settings...
Set Up
Run Once
Run Continuously
ticks: 11
Temperature Controls
Temperature Around Hornet
50
Temp
28
0
Time
50
On Off
Command Center
Clear
observer>

num-bees 50
radius 10
killing_rate 50
temp-increase 1.30
original_temp 30
diffuse-rate 0.30
Hornets Temperature 30.76
Dead Bees 5
show-labels
show-functions
show-input
plot?

Fig. 4 A simulation developed in NetLogo based on CXS models for the bee colony

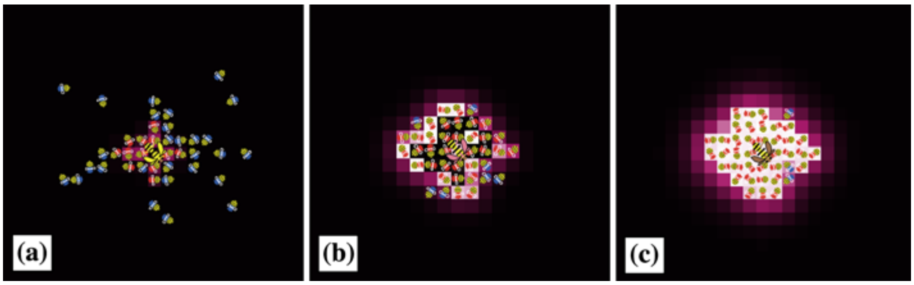


Fig. 5 Three phases of the simulation

These shortcomings could only become apparent through simulation and not through model checking.

5.3 Enhancing the model

In the core of this paper, we dealt only with communicating models that represent a (static) instance of a system, as the one depicted in Fig. 3. However, the model may be further enhanced with features that deal with a potential dynamic structure of the system, by considering PPS. For example, the following simple scenarios may be considered:

– if a bee is bitten by a hornet it should be removed from the PPS model,
– if a hornet is dead by heat it should also be removed from the PPS model,
– if another hornet arrives in the hive, a new cell should be generated,
– while bees are moving they should be able to communicate with other bees in the immediate neighbourhood, etc.

All the above issues can be dealt with by features of PPS, such as cell death, bond-making rules, cell division etc. In the first case, a cell death rule in C_{H1}, such as $((state : dead\_by\_heat)) \rightarrow \dagger$ will model it. In the second case, the corresponding death rule is $((state : killed)) \rightarrow \dagger$.

A bond-making rule such as:

$$(C_{B_i}, (memory : (PosBee_i, \_, \_)); (memory : (PosBee_j, \_, \_)), C_{B_j})$$
$$\text{if } next(PosBee_i, PosBee_j)$$

will produce a bond between two neighbouring bees.

5.4 Representing the input stream

In Sect. 4.3 it has been identified that there is an issue to consider when representing the input stream of sXM as a multiset of input objects inside the corresponding tPS membrane. Besides the approach followed above, other ways to resolve the issue also exist.

One would be to have an object of the form $(inputs : \sigma :: \varsigma)$ inside each tPS membrane where $\sigma :: \varsigma$ is a sequence of sXM input symbols corresponding to the sXM stream. If this is the case, we must consider that an additional tPS rewriting rule $(inputs : \sigma :: \varsigma) \rightarrow (input : \sigma)(inputs : \varsigma)$ is required and applied in every computation cycle to extract the next input (the first in the sequence) to be consumed.

A second option is to consider there is an additional membrane for generating the input symbols so that they are communicated one at a time to the corresponding receiver membranes. Note, at this point, that this is similar to the approach taken in practice for the implementation of the tool that transforms XMDL to PPSDL models (see next section) whereby during the animation of the model the inputs are entered one by one for each of the cells at each computation cycle.

5.5 Tools for transformation

There exist several tools that facilitate modelling with sXM, CXS as well as the testing and verification of models (Kapeti and Kefalas 2000; Thomson and Holcombe 2005). Also, there is a PPS modelling language accompanied with an animator which enables modelling directly to PPS (Stamatopoulou et al. 2005b). Bearing in mind the principles presented in this paper, an automatic transformation tool from CXS into PPS is developed (Kefalas et al. 2008). The benefit from such transformation of any CXS model to an equivalent PPS model is that we take advantage of existing CXS models that have been verified in order to enhance them with features that refer to the dynamic reconfiguration of their structure. Once the CXS model is compiled into a PPS model, one can add more PPS rules that deal with division, differentiation and death of cells as described above. The resulting model can be successfully animated, thus simulating the computation that takes place.

6 Conclusions

We presented a set of principles that guide the transformation of communicating X-machine models into different classes of P systems. One of the motives behind this attempt lies in the fact that the resulting P system model can be further enriched with various features like, removing or instantiating components, restructuring the links between them, etc. Thus, taking CXS models, which consists of sXM that can be individually verified, and transforming them into tPS and furthermore to population P system models, we could use more rules to extend the model with dynamic features. Practically, this means that not only we surpass the shortcomings of P systems in modelling agent behaviours, but we also feel quite confident (depending on a formal proof that the transformation is correct) that the resulting PPS model meets at least some quality requirements.

Acknowledgements The authors would like to thank Prof. Mike Holcombe for his inspiring ideas on X-machines. His support and contribution encouraged our research motivation for more than a decade.

References

Bernardini F, Gheorghe M (2004) Population P systems. J Univers Comput Sci 10(5):509–539

Ciobanu G, Aman B (2007) On the relationship between membranes and ambients. BioSystems 91(3): 515–530

Corne D, Frisco P (2007) Dynamics of HIV infection studied with cellular automata and conformon-P systems. BioSystems 91(3):531–544

Dwyer MB, Hatcliff J, Robby R, Pasareanu CS, Visser W (2007) Formal software analysis emerging trends in software model checking. In: Future of software engineering (FOSE'07), Washington, DC. IEEE Computer Society, pp 120–136

Eilenberg S (1974) Automata, languages and machines. Academic Press, London

Eleftherakis G (2003) Formal verification of X-machine models: towards formal development of computer-based systems. PhD thesis, Department of Computer Science, University of Sheffield

Eleftherakis G, Kefalas P (2008) Formal verification of generalised state machines. In: 12th Panhellenic conference in informatics (PCI'08), Samos Island, Greece, 28–30 August

Holcombe M, Ipate F (1998) Correct systems: building a business process solution. Springer, London

Jackson D, Holcombe M, Stamatopoulou I, Sakellariou I, Eleftherakis G, Kefalas P, Gheorghe M (2005) Modelling self-organisation in ant colonies. In: Special session on systems self-assembly at the 7th international conference on artificial evolution (EA'05), Lille, France, 26–28 October

Kapeti E, Kefalas P (2000) A design language and tool for X-machines specification. In: Fotiadis DI, Spyropoulos SD (eds) Advances in informatics. World Scientific Publishing Company, Singapore, pp 134–145

Kefalas P, Eleftherakis G, Holcombe M, Gheorghe M (2003a) Simulation and verification of P systems through communicating X-machines. BioSystems 70(2):135–148

Kefalas P, Eleftherakis G, Kehris E (2003b) Communicating X-machines: a practical approach for formal and modular specification of large systems. J Inf Softw Technol 45(5):269–280

Kefalas P, Holcombe M, Eleftherakis G, Gheorghe M (2003c) A formal method for the development of agent-based systems. In: Plekhanova V (ed) Intelligent agent software engineering. Idea Publishing Group Co., Miami, FL, pp 68–98

Kefalas P, Stamatopoulou I, Eleftherakis G, Gheorghe M (2008) Transforming state-based models to P systems models in practice. In: Membrane computing: 9th international workshop, Edinburgh, UK, pp 247–264

Klein J, Koutny M (2007) Synchrony and asynchrony in membrane systems. In: Hoogeboom HJ, Paun G, Rozenberg G, Salomaa A (eds) Membrane computing: 7th international workshop, Leiden, Netherlands. Lecture notes in computer science, vol 4361. Springer, pp 66–85

Păun G (2000) Computing with membranes. J Comput Syst Sci 61(1):108–143, also circulated as a TUCS report since 1998.

Păun G (2002) Membrane computing: an introduction. Springer, Berlin

Stamatopoulou I, Gheorghe M, Kefalas P (2005a) Modelling dynamic configuration of biology-inspired multi-agent systems with communicating X-machines and population P systems. In: Mauri G, Păun G, Pérez-Jiménez MJ, Rozenberg G, Salomaa A (eds) Membrane computing: 5th international workshop, Milan, Italy. Lecture notes in computer science, vol 3365. Springer, pp 389–401

Stamatopoulou I, Kefalas P, Eleftherakis G, Gheorghe M (2005b) A modelling language and tool for population p Systems. In: Proceedings of the 10th Panhellenic conference in informatics (PCI'05), Volas, Greece, pp 142–152

Stamatopoulou I, Kefalas P, Gheorghe M (2007a) OPERAS for space: Formal modelling of autonomous spacecrafts. In: Papatheodorou T, Christodoulakis D, Karanikolas N (eds) Current trends in informatics, Proceedings of the 11th Panhellenic conference in Informatics (PCI'07), Patras, Greece, vol B, pp 69–78

Stamatopoulou I, Kefalas P, Gheorghe M (2007b) OPERAS$_{CC}$: an instance of a formal framework for MAS modelling based on population P systems. In: Eleftherakis G, Kefalas P, Paun G (eds) Proceedings of the 8th workshop on membrane computing (WMC'07), Thessaloniki, South-East European Research Centre, pp 551–566

Stamatopoulou I, Sakellariou I, Kefalas P, Eleftherakis G (2008) OPERAS for social insects: formal modelling and prototype simulation. In: Gheorghe M, Ipate F (eds) Special issue of Romanian Journal of Information Science and Technology (ROMJIST) on Natural Computing—from biology to computer science and back to applications, vol 11, No. 3, pp 267–280

Thomson C, Holcombe M (2005) Using a formal method to model software design in XP projects. In: Eleftherakis G (ed) Proceedings of the 2nd south-east European workshop on formal methods, Ohrid, 18–19 November, South-East European Research Centre, pp 74–88

Wilensky U (1999) Netlogo. http://ccl.northwestern.edu/netlogo. Center for connected learning and computer-based modelling. Northwestern University, Evanston, IL