Journal of Automata, Languages and Combinatorics **19** (2014) 1–4, 239–250

FROM KERNEL P SYSTEMS TO X-MACHINES AND FLAME

Ionuţ Mihai Niculescu
Department of Mathematics and Computer Science, University of Piteşti
Str. Târgu din Vale 1, 110040, Piteşti, Romania
e-mail: ionutmihainiculescu@gmail.com

Marian Gheorghe
Department of Computer Science, University of Sheffield
Regent Court, Portobello Street, Sheffield S1 4DP, UK and
Department of Mathematics and Computer Science, University of Piteşti
Str. Târgu din Vale 1, 110040, Piteşti, Romania
e-mail: m.gheorghe@sheffield.ac.uk

Florentin Ipate and Alin Ştefănescu
Department of Computer Science, University of Bucharest
Str. Academiei 14, 010014, Bucharest, Romania and
Department of Mathematics and Computer Science, University of Piteşti
Str. Târgu din Vale 1, 110040, Piteşti, Romania
e-mail: florentin.ipate@ifsoft.ro and alin@stefanescu.eu

ABSTRACT

Kernel P systems have been introduced with the aim of unifying different variants of P systems. They provide a coherent framework for specifying and solving a broad spectrum of problems. The tools built so far have aimed to formally verify systems modelled as kernel P systems, but the simulators obtained are not very efficient. In this paper we show how to translate a class of kernel P systems into communicating stream X-machines, the model underlying the agent-based platform FLAME. This allows us to automatically translate kernel P systems into FLAME code, which is proved to be scalable and robust.

Keywords: P systems, X-machines, FLAME, translation

1. Introduction

Membrane computing studies computational models, called P systems, inspired by the functioning and structure of the living cell. Since their introduction by Gheorghe Păun in [15], P systems have have been intensely studied [16, 17]. In particular, many variants of P systems have been introduced and investigated in terms of computational power and their capability to solve computationally hard problems [17]. Furthermore,

in recent years, significant progress has been made in using various types or classes of P systems to model and simulate systems and problems from many different areas [3]. However, in many cases, the specifications developed required the ad-hoc addition of new features, not provided in the initial definition of the given P system class. While allowing more flexibility in modelling, this has led to a plethora of P system variants, with no coherent integrating view, and sometimes even confusion with regard to what variant or functioning strategy is actually used. The concept of *kernel P system* (*kP system*) [7, 8] has been introduced as a response to these problems. It integrates in a coherent and elegant manner many of the P system features most successfully used for modelling various applications and, thus, provides a framework for analyzing these models formally. The expressive power and efficiency of the newly introduced kP systems have been illustrated in a number of case studies involving NP-complete problems [9].

Originally introduced by Eilenberg [5], X-machines have been proposed by Holcombe as a specification language [10]. They combine a finite state machine-like control with data structures and operations, thus sharing the benefits of both these worlds. The most studied class of X-machines are those which, at each step, process an input symbol and produce in response an output symbol. These are called stream X-machines [11]. One of the main benefits of using a stream X-machine as a specification language is its associated testing method: under well-defined conditions, this guarantees that an implementation that passes the tests will conform to the specification. In order to specify distributed systems, several variants of communicating stream X-machines have been produced, most notably, the ones defined in [2] and [12]. Communicating stream X-machines are at the basis of FLAME [19] (Flexible Large-Scale Agent Modelling Environment), a platform for agent-based modelling on parallel architectures, successfully used in various applications ranging from biology to macroeconomics.

For various research purposes previous attempts have been made to either combine generative devices with X-machines - grammar systems and X-machines [6], membrane systems and X-machines [18] - or translate some basic variants of P systems into X-machines [1] or vice-versa [14].

In this paper we show how the behaviour of a class of kernel P systems using only rewriting and communication rules can be simulated by communicating stream X-machine systems. An implementation of such X-machine systems in FLAME will also be briefly discussed.

2. The Main Concepts and Definitions

The key concepts used in the paper, communicating stream X-machines and kernel P systems, are introduced in this section.

2.1. Communicating X-machines and FLAME

In this subsection we define the concepts of stream X-machine and communicating stream X-machine and discuss their relationship with FLAME [19]. The definitions

in this subsection are largely from [12]. We assume the reader is familiar with usual notations from formal languages and finite automata domain.

A stream X-machine is like a finite state machine in which the transitions are labelled by (partial) functions (called processing functions) instead of mere symbols. The machine has a memory (that represents the domain of the variables of the system to be modelled) and each processing function will read an input symbol, discard it and produce an output symbol while (possibly) changing the value of the memory.

Definition 1 A *Stream X-Machine* (*SXM* for short) is a tuple

$$Z = (\Sigma, \Gamma, Q, M, \Phi, F, I, T, m_0),$$

where:

- Σ and Γ are finite sets called the *input alphabet* and *output alphabet* respectively;
- Q is the finite set of *states*;
- M is a (possibly) infinite set called *memory*;
- Φ is the *type* of Z, a finite set of *function symbols*. A *basic processing function* $\phi : M \times \Sigma \longrightarrow \Gamma \times M$ is associated with each function symbol ϕ.
- F is the (partial) *next state function*, $F : Q \times \Phi \nrightarrow 2^Q$. As for finite automata, F is usually described by a *state-transition diagram*.
- I and T are the sets of *initial* and *terminal* states respectively, $I \subseteq Q, T \subseteq Q$;
- m_0 is the *initial memory* value, where $m_0 \in M$;
- all the above sets, i. e., Σ, Γ, Q, M, Φ, F, I, T, are non-empty.

It is sometimes helpful to think of an X-machine as a Finite Automaton (FA for short) with the arcs labelled by function symbols from the type Φ. The automaton $A_Z = (\Phi, Q, F, I, T)$ over the alphabet Φ is called *the associated FA* of Z. The automaton A_Z is *deterministic* if the machine has only one initial state and F maps each state/processing relation pair into at most one single state, i. e., $I = \{q_0\}$ and $F : Q \times \Phi \longrightarrow Q$.

Definition 2 We define a configuration of a SXM by (m, q, s, g), where $m \in M$, $q \in Q$, $s \in \Sigma^*$, $g \in \Gamma^*$. An initial configuration will have the form (m_0, q_0, s, ϵ), where m_0 is as in Definition 1, $q_0 \in I$ is an initial state, and ϵ is the empty word. A final configuration will have the form (m, q_f, ϵ, g), where $q_f \in T$ is a terminal state.

Definition 3 A change of *configuration*, denoted by $\vdash$, $(m, q, s, g) \vdash (m', q', s', g')$, is possible if $s = \sigma s'$ with $\sigma \in \Sigma$, $g' = g\gamma$ with $\gamma \in \Gamma$ and there exists $\phi \in \Phi$ such that $q' \in F(q, \phi)$ and $\phi(m, \sigma) = (\gamma, m')$. A change of configuration is called a *transition* of a SXM.

We denote by $\vdash^*$ the reflexive and transitive closure of $\vdash$.

A number of communicating SXMs variants have been defined in the literature. In what follows we will be presenting the communicating SXM model as defined in [12] since this is the closest to the model used in the implementation of FLAME [19] (there

are however, a few differences that will be discussed later). The model defined in [12] appears to be also the most natural of the existing models of communicating SXMs since each communicating SXM is a standard SXM as defined by Definition 1. In this model, each communicating SXM has only one (global) input stream of inputs and one (global) stream of outputs. Depending on the value of the output produced by a communicating SXM, this is placed in the global output stream or is processed by a SXM component. By contrast, the models defined in [2] and [13] separate the *standard* input and output streams, used in the communication with the environment, and *communicating* input and output streams, used in the communication between SXM components. Thus, the definitions of the processing functions are modified accordingly and no longer match the definition of a standard SXM as given in Definition 1. Additionally, in [2], the communication between SXM components is handled using a communication matrix, which is also used in the definition of the processing functions.

The following definitions are largely from [12].

Definition 4 A *Communicating Stream X-Machine System* (*CSXMS* for short) with n components is a tuple $S_n = ((Z_i)_{1 \leq i \leq n}, E)$, where:

- $Z_i = (\Sigma_i, \Gamma_i, Q_i, M_i, \Phi_i, F_i, I_i, T_i, m_{i,0})$ is the SXM with number $i, 1 \leq i \leq n$.
- $E = (e_{ij})_{1 \leq i,j \leq n}$ is a matrix of order $n \times n$ with $e_{ij} \in \{0, 1\}$ for $1 \leq i, j \leq n$, $i \neq j$ and $e_{ii} = 0$ for $1 \leq i \leq n$.

A CSXMS works as follows:

- Each individual *Communicating SXM* (*CSXM* for short) is a SXM plus an implicit input queue (i. e., of FIFO (first-in and first-out) structure) of infinite length; the CSXM only consumes the inputs from the queue.
- An input symbol σ received from the external environment (of FIFO structure) will go to the input queue of a CSXM, say Z_j, provided that it is contained in the input alphabet of Z_j. If more than one such Z_j exist, then σ will enter the input queue of one of these in a non-deterministic fashion.
- Each pair of CSXMs, say Z_i and Z_j, have two FIFO channels for communication; each channel is designed for one direction of communication. The communication channel from Z_i to Z_j is enabled if $e_{ij} = 1$ and disabled otherwise.
- An output symbol γ produced by a CSXM, say Z_i, will pass to the input queue of another CSXM, say Z_j, providing that the communication channel from Z_i to Z_j is enabled, i. e. $e_{ij} = 1$, and it is included in the input alphabet of Z_j, i. e. $\gamma \in \Sigma_j$. If these conditions are met by more than one such Z_j, then γ will enter the input queue of one of these in a non-deterministic fashion. If no such Z_j exists, then γ will go to the output environment (of FIFO structure).
- A CSXMS will receive from the external environment a sequence of inputs $s \in \Sigma^*$ and will send to the output environment a sequence of outputs $g \in \Gamma^*$, where $\Sigma = \Sigma_1 \cup \cdots \cup \Sigma_n$, $\Gamma = (\Gamma_1 \setminus In_1) \cup \cdots \cup (\Gamma_n \setminus In_n)$, with $In_i = \cup_{k \in K_i} \Sigma_k$, and $K_i = \{k \mid 1 \leq k \leq n, e_{ik} = 1\}$, for $1 \leq i \leq n$.

Definition 5 A *configuration* of a CSXMS S_n has the form $z = (z_1, \ldots, z_n, s, g)$, where:

- $z_i = (m_i, q_i, \alpha_i, \gamma_i), 1 \leq i \leq n$, where $m_i \in M_i$ is the current value of the memory of Z_i, $q_i \in Q_i$ is the current state of Z_i, $\alpha_i \in \Sigma_i^*$ is the current contents of the input queue and $\gamma_i \in \Gamma_i^*$ is the current contents of the output of Z_i;
- s is the current value of the input sequence;
- g is the current value of the output sequence.

Definition 6 An *initial configuration* has the form $z_0 = (z_{1,0}, z_{2,0}, \ldots, z_{n,0}, s, \epsilon)$, where $z_{i,0} = (m_{i,0}, q_{i,0}, \epsilon, \epsilon)$, with $q_{i,0} \in I_i$. A *final configuration* has the form $z_f = (z_{1,f}, z_{2,f}, \ldots, z_{n,f}, \epsilon, g)$, where $z_{i,f} = (m_i, q_{i,f}, \alpha_i, \gamma_i)$, with $q_{i,f} \in T_i$.

Passing from a configuration z to a new configuration z' supposes that at least one of the X-machines changes its configuration, i. e., a processing function is applied.

Definition 7 A change of configuration of a CSXMS S_n, denoted by $\models$,

$$z = (z_1, \ldots, z_n, s, g) \models z' = (z'_1, \ldots, z'_n, s', g'),$$

with $z_i = (m_i, q_i, \alpha_i, \gamma_i)$ and $z'_i = (m'_i, q'_i, \alpha'_i, \gamma'_i)$, is possible if one of the following is true for some i, $1 \leq i \leq n$:

1. $(m'_i, q'_i, \alpha'_i, \gamma'_i) = (m_i, q_i, \alpha_i\sigma, \epsilon)$, with $\sigma \in \Sigma_i$; $z'_k = z_k$ for $k \neq i$; $s = \sigma s'$, $g' = g$;
2. $(m_i, q_i, \sigma\alpha_i, \gamma_i) \vdash (m'_i, q'_i, \alpha'_i, \gamma)$ with $\sigma \in \Sigma_i$, $\gamma \in (\Gamma_i \setminus In_i)$; $z'_k = z_k$ for $k \neq i$; $s' = s$, $g' = g\gamma$;
3. $(m_i, q_i, \sigma\alpha_i, \gamma_i) \vdash (m'_i, q'_i, \alpha'_i, \gamma)$ with $\sigma \in \Sigma_i \cup \{\epsilon\}$, $\gamma \in (\Gamma_i \cap \Sigma_j) \cup \{\epsilon\}$ for some $j \neq i$ such that $e_{ij} = 1$; $(m'_j, q'_j, \alpha'_j, \gamma'_j) = (m_j, q_j, \alpha_j\gamma, \epsilon)$; $z'_k = z_k$ for $k \neq i$ and $k \neq j$; $s' = s$, $g' = g$;

A change of configuration is called a *transition* of a CSXMS.

The three types of transitions above are called: *input transitions* (1), *output transitions* (2) and *internal transitions* (3). These are denoted by $\models_{inp}$, $\models_{out}$, $\models_{int}$, respectively.

We denote by $\models^*$ the reflexive and transitive closure of $\models$.

The correspondence between the input sequence applied to the system and the output sequence produced gives rise to the relation computed by the system, as defined next.

Definition 8 The *relation computed* by a CSXMS, $f_{S_n} : \Sigma \longleftrightarrow \Gamma$ is defined by: $s\, f_{S_n}\, g$ if there exists $z_0 = (z_{1,0}, \ldots, z_{n,0}, s, \epsilon)$ and $z_f = (z_{1,f}, \ldots, z_{n,f}, \epsilon, g)$ an initial and final configuration, respectively, such that $z_0 \models^* z_f$ and there is no other configuration z such that $z_f \models z$. Such a sequence of transitions (i. e., $z_0 \models^* z_f$) will be called a *complete sequence of transitions*.

Note that the computation induced by f_{S_n} will always start with an input transition.

Remark 1 The implementation of FLAME use CSXMS such that: (i) the associated FA of each CSXM has no loops; and (ii) the CSXMSs receive no inputs from the environment, i. e., the inputs received are either empty inputs or outputs produced (in the previous computation step) by CSXM components of the system.

Remark 2 We say that a CSXMS runs in a *slow environment* if inputs can be sent from the environment to the system only in situations where the input queues of all CSXMs are empty. It can be observed that in FLAME the CSXMS runs in a slow environment.

2.2. Kernel P Systems

A *kernel P system* (kP system for short) is made of compartments placed in a graph-like structure. A compartment C_i has a type $t_i = (R_i, \rho_i)$, $t_i \in T$, where T represents the set of all types, describing the associated set of rules R_i and the execution strategy that the compartment may follow. Note that, unlike traditional P system models, in kP systems each compartment may have its own rule application strategy. The following definitions are largely from [8].

Definition 9 A *kP system* of degree n is a tuple $k\Pi = (A, \mu, C_1, \ldots, C_n, i_0)$, where

- A is a finite set of elements called *objects*;
- μ defines the *membrane structure*, which is a graph, (V, E), where V are vertices indicating components (compartments), and E edges, i. e., links between components;
- $C_i = (t_i, w_{i,0})$, $1 \leq i \leq n$, is a *compartment* of the system consisting of a *compartment type*, t_i, from T and an *initial multiset*, $w_{i,0}$ over A; the type $t_i = (R_i, \rho_i)$ consists of a *set of rules*, R_i, and an *execution strategy*, ρ_i;
- i_0 is the *output compartment* where the result is obtained.

Each rule r may have a **guard** g denoted as $r\ \{g\}$. The rule r is applicable to a multiset u when its left hand side is contained into u and g holds for u. The guards are constructed using multisets over A and relational and Boolean operators. For example, rule $r : ac \to c\ \{\geq a^3 \wedge \geq b^2 \vee \neg > c\}$ can be applied iff the current multiset, w, includes the left hand side of r, i. e., ac and the guard holds for w: it has at least 3 a's and 2 b's or no more than a c. A formal definition may be found in [8].

Definition 10 A rule associated with a compartment type l_i can have one of the following types:

(a) **rewriting and communication** rule: $x \to y\ \{g\}$,
where $x \in A^+$ and y has the form $y = (a_1, t_1) \ldots (a_h, t_h)$, $h \geq 0$, $a_j \in A$ and t_j indicates a compartment type from T – see Definition 9 – with instance compartments linked to the current compartment; t_j might indicate the type of the current compartment, i. e., t_{l_i} – in this case it is ignored; if a link does not exist (the two compartments are not in E) then the rule is not applied; if a target

t_j refers to a compartment type that has more than one instance connected to l_i, then one of them will be non-deterministically chosen; in the sequel the multiset consisting of the symbols that are associated with target t_{l_i} will be denoted by u and the rule will be written as $x \to u(a_{i_1}, t_{i_1}) \dots (a_{i_p}, t_{i_p})$ $\{g\}$, where $t_{i_j} \neq t_{l_i}$, $1 \leq j \leq p$ and $0 \leq p \leq h$;

(b) **structure changing rules**; these may be *membrane division*, *membrane dissolution*, *link creation* or *link creation* rules. Since structural changes are not considered in this paper, their formal definition is not given here. For details the reader is referred to [8].

Each compartment can be regarded as an instance of a particular *compartment type* and is therefore subject to its associated rules. Each compartment has also an associated execution strategy. An execution strategy can be defined as a sequence $\rho = \rho_1 \& \rho_2 \& \dots \& \rho_n$, $n \geq 1$, where ρ_i, $1 \leq i \leq n$, associated with the set of rules R_t denotes one of the following:

- r, a rule from the set R_t; if r is applicable, then it is executed, then advancing towards the next element, ρ_{i+1}; otherwise, the compartment's execution strategy, ρ, terminates here;
- $(r_1, \dots, r_m)$, with $r_i \in R_t, 1 \leq i \leq m$, symbolizes a non-deterministic choice within a set of rules; one and only one applicable rule will be executed if such a rule exists, otherwise ρ_i is simply skipped;
- $(r_1, \dots, r_m)^*$, with $r_i \in R_t, 1 \leq i \leq m$, indicates the arbitrary execution of a set of rules from R_t; the group can be executed zero or more times by non-deterministically chosing any of the applicable rules;
- $(r_1, \dots, r_m)^\top, r_i \in R_t, 1 \leq i \leq m$, represents the maximally parallel execution of a set of rules.

An execution step in a compartment using the execution strategy ρ applied to a multiset u and leading to v is denoted $u \Longrightarrow_\rho v$.

Definition 11 A *configuration* of a kP system, $k\Pi$, has the form $w = (w_1, \dots, w_n)$, where $w_i, 1 \leq i \leq n$, is a multiset over A belonging to compartment C_i. A change of configuration, denoted $w = (w_1, w_2, \dots, w_n) \Longrightarrow w' = (w'_1, w'_2, \dots, w'_n)$, holds when $w_i \Longrightarrow_{\rho_i} w'_i$, $1 \leq i \leq n$. A change of configuration is called a *computation step.* The reflexive and transitive closure of this relation is denoted $\Longrightarrow^*$.

As usual in many P systems we compute a multiset, w, starting from the initial configuration, $w_0 = (w_{1,0}, \dots, w_{n,0})$, of a kP system, $k\Pi$, by applying $\Longrightarrow^*$, i. e., $w_0 \Longrightarrow^* w$. Finally the number of elements of the multiset is considered and $N(k\Pi)$ denotes the set of these numbers.

3. Major Results

In the sequel, we show how the behaviour of kernel P systems using only rewriting and communication rules can be simulated by communicating stream X-machine systems. Before presenting the result we make some notations.

The compartments of the kP system are using *multisets* over A, whereas the CSXMS is dealing with *strings*. When w denotes a multiset over A then $s(w)$ is any of the strings obtained by concatenating the symbols occurring in w.

Theorem 1 *For any kP system, $k\Pi$, of degree n and using only rewriting and communication rules there is a communicating stream X-machine system, S_{n+1}, with $n+1$ components such that for any multiset w computed by $k\Pi$ there is a complete sequence of transitions in S_{n+1} leading to $s(w)$.*

Proof. Let us consider a kP system of degree n, as introduced by Definition 9, and using only rules of type (a) – see Definition 10, $k\Pi = (A, \mu, C_1, \ldots, C_n, i_0)$.

We build the following CSXMS – as introduced by Definition 4 – with $n+1$ components, $S_{n+1} = ((Z_{i,t_i})_{1\leq i\leq n}, Z_{n+1}, E')$. For each compartment, $C_i = (t_i, w_i)$, $1 \leq i \leq n$, a CSXM with number i and corresponding to type t_i, is considered, $Z_{i,t_i} = (\Sigma_i, \Gamma_i, Q_i, M_i, \Phi_i, F_i, I_i, T_i, m_{i,0})$, $1 \leq i \leq n$, and $Z_{n+1} = (\Sigma_{n+1}, \Gamma_{n+1}, Q_{n+1}, M_{n+1}, \Phi_{n+1}, F_{n+1}, I_{n+1}, T_{n+1}, m_{n+1,0})$ is the CSXM with number $n+1$. The CSXM $Z_{i,t_i}, 1 \leq i \leq n$, will simulate the behaviour of the compartment C_i and Z_{n+1} is built such that it helps synchronising the other n CSXMs. The matrix $E' = (e'_{i,j})_{1\leq i,j\leq n+1}$ is defined such, $e'_{i,j} = 1, 1 \leq i, j \leq n$, iff there is an edge between i and j in the membrane structure of $k\Pi$ and $e'_{i,n+1} = e'_{n+1,i} = 1, 1 \leq i \leq n$ (i. e., there are connections between any of the first n CSXMs and Z_{n+1}, and vice-versa).

The idea of the proof is the following: the relation computed by S_{n+1}, $f_{S_{n+1}}$, is defined by an initial step when Z_{n+1} reads the input σ_0, the only symbol of the input sequence, which goes into the input queue of this CSXM. This input will initialise each of the Z_i, $1 \leq i \leq n$, CSXMs by sending $[\sigma_0, i]$ to it instead, which in turn will lead to processing the strings corresponding to initial multisets, $s(w_{i,0})$. For each computation step in $k\Pi$ and every execution strategy ρ_i, $1 \leq i \leq n$, the CSXMS S_{n+1} will perform a number of transitions. Finally, when the kP system stops the computation, and the multiset w is obtained in C_{i_0}, then the CSXMS moves to a final state and the result is sent out as an output sequence, $s(w)$.

We show that if $w_0 = (w_{1,0}, \ldots, w_{n,0}) \Longrightarrow^* w = (w_1, \ldots, w_{i_0}, \ldots, w_n)$, i. e., the multiset w_{i_0} is computed by $k\Pi$, then there is an initial configuration, $z_0 = (z_{1,0}, \ldots, z_{n+1,0}, s, \epsilon)$, and a final configuration, $z_f = (z_{1,f}, \ldots, z_{n+1,f}, \epsilon, g)$, in the CSXMS, S_{n+1}, such that $z_0\, f_{S_{n+1}}\, z_f$ and $s = \sigma_0$, $g = s(w_{i_0})$.

Definition of the CSXMS, S_{n+1}.

The components of Z_{i,t_i}, $1 \leq i \leq n$, are

- $\Sigma_i = \{[a, t_i] \mid a \in A\} \cup \{[\sigma_0, i], [\sigma'_0, i]\}$;
- $\Gamma_i = \{[a, t] \mid a \in A, t \in T\} \cup A \cup \{[1, i], [0, i]\}$;
- $Q_i = \{q_{i,j} \mid 0 \leq j \leq 4\} \cup \{q_{i,f}\}$;
- $M_i = S_{i,1} \times S_{i,2} \times S_{i,3}$, where $S_{i,1}$, $j = 1, 2$, are subsets of A^*, $S_{i,3}$ is a subset of $(A \cup \{!\} \cup A \times T)^*$;
- $\Phi_i = \{Init_i, Select\_rule_i, Rewrite_i, Send\_symbol_i, Remove\_rule_i, No\_rule_i, Bring\_comm\_symb_i, Return_i, Stop_i\}$ see below;
- F_i is given below; $I_i = \{q_{i,0}\}$; $T_i = \{q_{i,f}\}$ and $m_{i,0} = (s(w_{i,0}), \epsilon, \epsilon)$.

The set Φ_{i_0} contains also Out_{i_0}.

Z_{n+1} consists of

– $\Sigma_{n+1} = \{[a,i] \mid a \in \{\sigma_0, 1, 0\}, 1 \leq i \leq n\}$;
– $\Gamma_{n+1} = \{[a,i] \mid a \in \{\sigma_0, \sigma_0'\}, 1 \leq i \leq n\}$;
– $Q_{n+1} = \{q_{n+1,0}\}$;
– M_{n+1} is a subset of $\{\sigma_0, 0, 1\}^*$;
– $\Phi_{n+1} = \{Read\_inp_{n+1}, Get\_1\_or\_0_{n+1}\} \cup \{Init_{n+1,i}, Signal_{n+1,i} \mid 1 \leq i \leq n\}$ see below;
– F_{n+1} is given below; $I_{n+1} = \{q_{n+1,0}\}$; $T_{n+1} = \{q_{n+1,0}\}$ and $m_{n+1,0} = \sigma_0^n$.

Initial step. The initial configuration is $z_0 = (z_{1,0}, z_{2,0}, \ldots, z_{n+1,0}, \sigma_0, \epsilon)$ with $z_{i,0} = (m_{i,0}, q_{i,0}, \epsilon, \epsilon)$, $1 \leq i \leq n+1$. The function $Read\_inp_{n+1}$ from Φ_{n+1} reads the input σ_0 and appends it to Z_{n+1} input queue. According to Definition 7, we have

$$z_0 = (z_{1,0}, \ldots, z_{n,0}, z_{n+1,0}, \sigma_0, \epsilon) \models_{inp} z_1 = (z_{1,0}, \ldots, z_{n,0}, z_{n+1,1}, \epsilon, \epsilon),$$

where $z_{n+1,1} = (m_{n+1,0}, q_{n+1,0}, \sigma_0, \epsilon)$. Now a function $Init_{n+1,i}$, $1 \leq i \leq n$, from Φ_{n+1} will send $[\sigma_0, i]$ to Z_i. The following transition takes place

$$z_h = (z_{1,h}, \ldots, z_{n+1,h}, \epsilon, \epsilon) \models z_{h+1} = (z_{1,h}, \ldots, z_{i,h+1}, \ldots, z_{n+1,h+1}, \epsilon, \epsilon),$$

where $z_{i,h} = ((s(w_{i,0}), \epsilon, \epsilon), q_{i,0}, \epsilon, \epsilon)$, $z_{i,h+1} = ((s(w_{i,0}), \epsilon, \epsilon), q_{i,0}, [\sigma_0, i], \epsilon)$, $z_{n+1,h} = (\sigma_0^k, q_{n+1,0}, \epsilon, \epsilon)$, $z_{n+1,h+1} = (\sigma_0^{k-1}, q_{n+1,0}, \epsilon, [\sigma_0, i])$. When $k = 1$, σ_0 is replaced in the memory of Z_{n+1} by σ_0^n. Finally, in each CSXM, Z_{i,t_i}, $1 \leq i \leq n$, a function $Init_i$ will start the computation by consuming $[\sigma_0, i]$ from the input queue and moving to a state $q_{i,1}$; the following change of configuration in Z_{i,t_i} describes this computation $(m_{i,0}, q_{i,0}, [\sigma_0, i], \epsilon) \vdash (m_{i,0}, q_{i,1}, \epsilon, \epsilon)$.

Iterative step. We show now how a computation step in $k\Pi$, $w_h = (w_{1,h}, \ldots, w_{n,h}) \Longrightarrow w_{h+1} = (w_{1,h+1}, \ldots, w_{n,h+1})$, i. e., there is at least an execution strategy ρ_i, $1 \leq i \leq n$, such that $w_{i,h} \Longrightarrow_{\rho_i} w_{i,h+1}$, leads to a sequence of transitions in CSXMS, starting with CSXM, Z_{i,t_i}, $1 \leq i \leq n$, in configuration $z_{i,h} = ((s(w_{i,h}), \epsilon, \epsilon), q_{i,1}, \epsilon, \epsilon)$ and Z_{n+1} in configuration $z_{n+1,h} = ((\epsilon, \epsilon), q_{i,0}, \epsilon, \epsilon)$, and arriving with Z_{i,t_i}, in configuration $z_{i,h+1} = ((s(w_{i,h+1}), \epsilon, \epsilon), q_{i,1}, \epsilon, \epsilon)$ and Z_{n+1} in configuration $z_{n+1,h+1} = z_{n+1,h}$. Initially, the first iterative step, $h = 0$, starts with the initial set of multisets, w_0.

Each iterative step consists of the following distinct stages: (i) Rule selection and execution: in each CSXM, Z_i, $1 \leq i \leq n$, the rules defined by the execution strategy ρ_i are selected one by one; each rule is executed in a number of sub-steps and the objects that have to be communicated are kept in the input queues of the target machines; after all the rules of the CSXM Z_i are executed, a signal is sent to Z_{n+1}; (ii) Next step preparation: when at least one rule has been applied in a component and all Z_i have stopped then Z_{n+1} signals the beginning of the next iterative step; (iii) Stop: when no rule is applicable, the process stops.

(i) Rule selection and execution. The following functions are involved in this stage. We start with functions in Z_{i,t_i}. The function $Select\_rule_i$ selects from ρ_i the next rule to be executed; if more than one can be selected – the case of iteration – then one is non-deterministically chosen; let it be denoted by $r : x \to u(a_1, t_1) \ldots (a_p, t_p)$, where x, u are multisets over A and $a_j \in A$, $1 \leq j \leq p$, and t_j indicates the target

type. $Select\_rule_i$ is updating the third component of the current memory with the string corresponding to r and moves the machine to state $q_{i,2}$; the new configuration will become $((s(w_{i,h}), \epsilon, s_r), q_{i,2}, \epsilon, \epsilon)$, where $s_r = s(x)!s(u)!(a_1, t_1) \dots (a_p, t_p)$. One can observe that substrings of s_r are delimited by ! in order to allow further analysis.

The next function, $Rewrite_i$, rewrites x by u. If the current memory is $(s(w_{i,h}), \epsilon, s_r)$, then $Rewrite_i$ leads to $(\alpha, s(u), s_r)$, where α is obtained from $s(w_{i,h})$ by removing the symbols occurring in x; the new configuration is $((\alpha, s(u), s_r), q_{i,3}, \beta_i, \epsilon)$. One can note that $s(u)$ is kept in the second component of the memory and β is either ϵ, as before, or a string of symbols sent by other CSXMs.

The function $Send\_symbol_i$ will send the symbols a_d to target t_d, $1 \leq d \leq p$, as $[a_d, t_d]$. These symbols will be accepted only by Z_{j,t_d} associated with type t_d and if there are more than one CSXMs then one is non-deterministically chosen. The configuration of Z_{i,t_i} is now $(m_{i,h''}, q_{i,3}, \beta_i, [a_d, t_d])$ and for Z_j is $(m_{j,h''}, q_{j,k}, \beta_j[a_d, t_d], \epsilon)$.

$Remove\_rule_i$ is used to remove s_r from memory and return to state $q_{i,1}$, ready to select a new rule.

When no rule is left to be selected then No_rule_i brings the machine to state $q_{i,4}$ and sends to Z_{n+1} the symbol $[1, i]$; when ρ_i is not applicable then $[0, i]$ is sent to Z_{n+1}.

In state $q_{i,4}$ the function $Bring\_comm\_symb_i$ brings symbols, one by one, from the input queue, appending them to the second component of the memory, hence $((\alpha_i, \beta_i, s_r), q_{i,4}, [a, t_i]x_i, \epsilon) \models ((\alpha_i, \beta_i a, s_r), q_{i,4}, x_i, \epsilon)$.

The last function of this stage is from Z_{n+1}, $Get\_1\_or\_0_{n+1}$, which receives through the input queue either $[1, i]$ or $[0, i]$, $1 \leq i \leq n$, from Z_{i,t_i} and replaces the corresponding σ_0 from the memory by 1 or 0, depending on the current input; the function will not change the state, $q_{n+1,0}$.

(ii) Next step preparation. If the memory of Z_{n+1} contains at least one 1 then $Signal_{n+1,i}$, $1 \leq i \leq n$, is used in order to replace 0 or 1 in position i of the current memory by σ_0 and sends $[\sigma_0, i]$ to Z_{i,t_i}. This symbol arrives in the input queue after all the communicating symbols have been processed.

A function $Return_i$ in Z_{i,t_i} receiving $[\sigma_0, i]$ in state $q_{i,4}$ will append the second memory component to the first, and will return to $q_{i,1}$ ready for a new iteration.

(iii) Stop. If the memory of Z_{n+1} contains only 0′s then $Signal_{n+1,i}$, $1 \leq i \leq n$, will send $[\sigma_0', i]$ to Z_{i,t_i}.

In each Z_{i,t_i} a function $Stop_i$ will bring the CSXM to a final state $q_{i,f}$.

For $Z_{i_0,t_{i_0}}$ there is another function Out_{i_0} which will send the string from the first memory component as an output sequence. This is the string representing the result of the computation. □

Remark 3 The above result can be easily extended to some other classes of kernel P systems using structure changing rules. *Membrane dissolution* can be directly implemented in the above proof by extending the memory with another component showing whether the corresponding CSXM is active or dissolved. The *link creation* and *link destruction* options can be introduced into the above proof by simply considering that the matrix E' can be changed during the computation such that edges are available

or not at certain steps. *Membrane division* cannot be implemented in the current framework as the structure of the system cannot be expanded.

Remark 4 The translation to FLAME using the method described in the proof of Theorem 1 provides a better description of the basic types of execution strategies [20]. Some examples illustrate the results of some translations, as FLAME X-machines [20].

4. Conclusions

In this paper we have presented a result showing how a class of kernel P systems using only rewriting and communication rules can be translated into a communicating stream X-machine system and how this translation can be extended to some other classes of kernel P systems using structure changing rules. An implementation of the translator in FLAME, an agent-based environment using the communicating stream X-machine model, is briefly discussed. More details about the translation into FLAME, as well as examples and some initial experiments are shown at [20].

As future developments, we intend to provide a version of the translation process which is closer to the FLAME model and to run experiments with both the kernel P systems environment [7, 8] and the FLAME translator in order to assess the performances of the translation mechanism and its FLAME simulator.

Acknowledgements

The authors were supported by a grant of the Romanian National Authority for Scientific Research, CNCS-UEFISCDI (project number: PN-II-ID-PCE-2011-3-0688). MG acknowledges the support provided by EPSRC ROADBLOCK (project number: EP/I031812/1).

References

[1] J. Aguado, T. Bălănescu, A. J. Cowling, M. Gheorghe, M. Holcombe, F. Ipate, P systems with replicated rewriting and stream X-machines (Eilenberg machines). *Fundamenta Informaticae* **49** (2002), 17–33.

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

[3] G. Ciobanu, M. J. Pérez-Jiménez, Gh. Păun (eds.), *Applications of Membrane Computing*. Springer, 2006.

[4] E. Csuhaj-Varjú, J. Dassow, On cooperating/distributed grammar systems. *Elektronische Informationsverarbeitung und Kybernetic* **26** (1990), 49–63.

[5] S. Eilenberg, *Automata, Languages and Machines*. Academic Press, New York, 1974.

[6] M. Gheorghe, Generalised stream X-machines and cooperating distributed grammar systems. *Formal Aspects of Computing* **12** (2000), 459–472.

[7] M. Gheorghe, F. Ipate, C. Dragomir, Kernel P systems. In: M.A. Martínez-del-Amor, Gh. Păun, F. Romero-Campero (eds.), *Tenth Brainstorming Week on Membrane Computing, BWMC 2012, Sevilla, Spain, February 2012*. Universidad de Sevilla, 2012, 153–170.

[8] M. Gheorghe, F. Ipate, C. Dragomir, L. Mierlă, L. Valencia-Cabrera, M. García-Quismondo, M. J. Pérez-Jiménez, Kernel P systems – Version I. In: L. Valencia-Cabrera, M. García-Quismondo, L.F. Macías-Ramos. M.A. Martínez-del-Amor, Gh. Păun, A. Riscos-Núñez (eds.), *Eleventh Brainstorming Week on Membrane Computing, BWMC 2013, Sevilla, Spain, February 2013*. Universidad de Sevilla, 2013, 97–124.

[9] M. Gheorghe, F. Ipate, R. Lefticaru, M.J. Pérez-Jiménez, A. Ţurcanu, L. Valencia, M. García-Quismondo, L. Mierlă, 3-COL problem modelling using simple kernel P systems. *International Journal of Computer Mathematics* **90** (2013), 816–830.

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

[11] M. Holcombe, F. Ipate, *Correct Systems – Building a Business Process Solution*, Springer, 1998.

[12] F. Ipate, T. Bălănescu, P. Kefalas, M. Holcombe, G. Eleftherakis, A new model of communicating stream X-machine systems. *Romanian Journal of Information Science and Technology* **6** (2003), 165–184.

[13] P. Kefalas, G. Eleftherakis, E. Kehris, Communicationg X-machines: from theory to practice. In: Y. Manolopoulos, S. Evripidou, A. Kakas (eds.), Advances in Informatics, LNCS 2563, 2003.

[14] P Kefalas, I. Stamatopoulou, I. Sekellariou, G. Eleftherakis, Transforming communicating X-machines into P systems. *Natural Computing* **8** (2009), 817–832.

[15] Gh. Păun, Computing with membranes. *Journal of Computer and System Sciences* **61** (2000), 108–143 (see also TUCS Report 208, November 1998, `www.tucs.fi`).

[16] Gh. Păun, *Membrane Computing: An Introduction*. Springer, 2002.

[17] Gh. Păun, G. Rozenberg, A. Salomaa (eds.), *The Oxford Handbook of Membrane Computing*. Oxford University Press, 2010.

[18] I. Stamatopoulou, P. Kefalas, M. Gheorghe, OPERAS: A framework for formal modelling of multi-agent systems and its application to swarm-based systems. *Proc. of the Eighth Intern. Workshop on Engineering Societies in the Agents World, ESAW 2007*. Springer-Verlag, 2007, 158–174.

[19] FLAME website (available at `http://www.flame.ac.uk/`).

[20] kP systems examples translated to FLAME (available at `http://www.muvet.ifsoft.ro/kP-to-flame/`).

(Received: December 20, 2013; revised: April 9, 2014)