Extended Simulation and Verification Platform for Kernel P Systems

Mehmet E. Bakir[1], Florentin Ipate[2], Savas Konur[1], Laurentiu Mierla[2], and Ionut Niculescu[3]

[1] Department of Computer Science, University of Sheffield
Regent Court, Portobello Street, Sheffield S1 4DP, UK
{mebakir1,s.konur}@sheffield.ac.uk
[2] Department of Computer Science, University of Bucharest
Str. Academiei nr. 14, 010014, Bucharest, Romania
florentin.ipate@ifsoft.ro, laurentiu.mierla@gmail.com
[3] Department of Computer Science, University of Pitesti
Str. Targul din Vale, nr.1, 110040 Pitesti, Arges, Romania
ionutmihainiculescu@gmail.com

Abstract. *Kernel P systems* integrate in a coherent and elegant manner many of the features of different P system variants, successfully used for modelling various applications. In this paper, we present our initial attempt to extend the software framework developed to support kernel P systems: a formal verification tool based on the NuSMV model checker and a large scale simulation environment based on FLAME. The use of these two tools for modelling and analysis of biological systems is illustrated with a synthetic biology example.

1 Introduction

Membrane computing [16] is a branch of natural computing inspired by the hierarchical structure of living cells. The central model, called *P systems*, consists of a membrane structure, the regions of which contain rewriting rules operating on multisets of objects [16]. P systems *evolve* by repeatedly applying rules, mimicking chemical reactions and transportation across membranes or cellular division or death processes, and halt when no more rules can be applied. The most recent developments in this field are reported in [17].

The origins of P systems make it highly suited as a formalism for representing biological systems, especially (multi-)cellular systems and molecular interactions taking place in different locations of living cells [7]. Different simple molecular interactions or more complex gene expressions, compartment translocation, as well as cell division and death are specified using multiset rewriting or communication rules, and compartment division or dissolution rules. In the case of *stochastic P systems*, constants are associated with rules in order to compute their probabilities and time needed to be applied, respectively, according to the Gillespie algorithm [18]. This approach is based on a Monte Carlo algorithm for the stochastic simulation of molecular interactions taking place inside a single volume or across multiple compartments.

The recently introduced class of *kernel P (kP) systems* [8] integrates in a coherent and elegant manner many of the features of different P system variants, successfully used for modelling various applications. The kP model is supported by a modelling language, called *kP-Lingua*, capable of mapping a kernel P system specification into a machine readable representation. Furthermore, the KPWorkbench framework that allows simulation and formal verification of the obtained models using the model checker Spin was presented in a recent paper [5].

In this paper, we present two new extensions to KPWorkbench: a formal verification tool based on the NuSMV model checker [4] and a large scale simulation environment using FLAME (Flexible Large-Scale Agent Modelling Environment) [6], a platform for agent-based modelling on

parallel architectures, successfully used in various applications ranging from biology to macroeconomics. The use of these two tools for modelling and analysis of biological systems is illustrated with a synthetic biology case study, the pulse generator.

The paper is structured as follows. Section 2 defines the formalisms used in the paper, stochastic and kernel P systems as well as stream X-machines and communicating stream X-machine systems, which are the basis of the FLAME platform. Section 3 presents an overview on the kP-lingua language and the simulation and model checking tools. The case study and the corresponding experiments are presented in Section 4 and 5, respectively, while conclusions are drawn in Section 6.

2 Basic definitions

2.1 Stochastic and kernel P systems

Two classes of P systems, used in this paper, will be now introduced. The first model is a **stochastic P system** with its components distributed across a lattice, called *lattice population P systems* [18, 3], which have been applied to some unconventional models e.g. the genetic Boolean gates [13, 12, 19]. For the purpose of this paper we will consider stochastic P systems with only one compartment and the lattice will be regarded as a tissue with some communication rules defined in accordance to its structure.

Definition 1. *A stochastic P system (SP system) with one compartment is a tuple:*

$$SP = (O, M, R) \tag{1}$$

where O is a finite set of objects, called alphabet; *M is the finite* initial multiset of objects *of the compartment, an element of O^*; R is a set of* multiset rewriting rules*, of the form $r_k : x \xrightarrow{c_k} y$, where x, y are multisets of objects over O (y might be empty), representing the molecular species consumed (x) and produced (y).*

We consider a finite set of labels, L, and a population of SP systems indexed by this family, SP_h, $h \in L$. A lattice, denoted by Lat, is a bi-dimensional finite array of coordinates, (a, b), with a and b positive integer numbers. Now we can define a lattice population P system, by slightly changing the definition provided in [3].

Definition 2. *A lattice population P system (LPP system) is a tuple*

$$LPP = (Lat, (SP_h)_{h \in L}, Pos, Tr) \tag{2}$$

where Lat, SP_h and L are as above and $Pos : Lat \to \{SP_h | h \in L\}$ is a function associating to each coordinate of Lat a certain SP system from the given population of SP systems. Tr is a set of translocation rules *of the form $r_k : [x]_{h_1} \xrightarrow{c_k} [x]_{h_2}$, where $h_1, h_2 \in L$; this means that the multiset x from the SP system SP_{h1}, at a certain position in Lat, will move to any of the neighbours (east, west, south, north) in Lat that contains an SP system SP_{h_2}.*

The stochastic constant c_k, that appears in both definitions above, is used by Gillespie algorithm [9] to compute the next rule to be applied in the system.

One can see the lattice as a tissue system and the SP systems as nodes of it with some communication rules defined according to the neighbours and also to what they consist of.

Another class of P systems, called **kernel P systems**, has been introduced as a unifying framework allowing to express within the same formalism many classes of P systems [8, 5].

Definition 3. *A kernel P system (kP system) of degree n is a tuple*

$$k\Pi = (O, \mu, C_1, \ldots, C_n, i_0) \tag{3}$$

where O is a finite set of objects, called alphabet*; μ defines the* membrane structure*, which is a graph, (V, E), where V are vertices indicating compartments, and E edges; $C_i = (t_i, w_i)$, $1 \leq i \leq n$, is a* compartment *of the system consisting of a compartment type from T and an* initial multiset*, w_i over O; i_0 is the* output compartment *where the result is obtained (this will not be used in the paper).*

Definition 4. *T is a* set of compartment types*, $T = \{t_1, \ldots, t_s\}$, where $t_i = (R_i, \sigma_i)$, $1 \leq i \leq s$, consists of a set of rules, R_i, and an execution strategy, σ_i, defined over $Lab(R_i)$, the labels of the rules of R_i.*

In this paper we will use only one execution strategy, corresponding to the execution of a rule in each compartment, if possible. For this reason the execution strategy will be no longer mentioned in the further definition of the systems. The rules utilised in the paper are defined below.

Definition 5. *A* rewriting and communication rule*, from a set of rules, R_i, $1 \leq i \leq s$, used in a compartment $C_{l_i} = (t_{l_i}, w_{l_i})$, $1 \leq i \leq n$, has the form $x \to y\ \{g\}$, where $x \in O^+$ and y has the form $y = (a_1, t_1) \ldots (a_h, t_h)$, $h \geq 0$, $a_j \in O$ and t_j indicates a compartment type from T – see Definition 3 – with instance compartments linked to the current compartment, C_{l_i}; 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 C_{l_i}, then one of them will be non-deterministically chosen.*

The definition of a rule from R_i, $1 \leq i \leq s$, is more general than the form provided above, see [8, 5], but in this paper we only use the current form. The guards, denoted by g, are Boolean conditions and their format will be discussed latter on. The guard must be true when a rule is applied.

2.2 X-machines and communicating stream X-machine systems

We now introduce the concepts of stream X-machine and communicating stream X-machine and also discuss how these are implemented in FLAME [6]. The definitions are largely from [11].

A stream X-machine is like a finite automaton 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 6. *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.*

A configuration of a SXM is a tuple (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 6, $q_0 \in I$ 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. 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 [11] since this is the closest to the model used in the implementation of FLAME [6] (there are however, a few differences that will be discussed later). The model defined in [11] 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 6. 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. For a more detailed discussion about the differences between various models of communicating SXMs see [15].

The following definitions are largely from [11].

Definition 7. *A* Communicating Stream X-Machine System *(*CSXMS *for short) with* n *components is a tuple* $S_n = ((Z_i)_{1 \le i \le n}, E)$*, 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* $i, 1 \le i \le n$*.*
- $E = (e_{ij})_{1 \le i,j \le n}$ *is a matrix of order* $n \times n$ *with* $e_{ij} \in \{0, 1\}$ *for* $1 \le i, j \le n$*,* $i \ne j$ *and* $e_{ii} = 0$ *for* $1 \le i \le n$*.*

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 $K_i = \{k \mid 1 \le k \le n, e_{ik} = 1\}$, for $1 \le i \le n$.

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.

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

A change of configuration happens when at least one of the X-machines changes its configuration, i.e., a processing function is applied. More formally, 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. 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*, f_{S_n}. More formally, $f_{S_n} : \Sigma \longleftrightarrow \Gamma$ 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 $z_f \models z$.

In [15] it is shown that for any kP system, $k\Pi$, of degree n, $k\Pi = (O, \mu, C_1, \ldots, C_n, i_0)$, using only rewriting and communication rules, there is a communicating stream X-machine system, $S_{n+1} = ((Z_{i,t_i})_{1 \leq i \leq n}, Z_{n+1}, E')$ 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 $s(w)$, the sequence corresponding to w. The first n CSXM components simulate the behaviour of the compartment C_i and the $(n+1)th$ component Z_{n+1} helps synchronising the other n CSXMs. The matrix $E' = (e'_{i,j})_{1 \leq i,j \leq n+1}$ is defined by: $e'_{i,j} = 1, 1 \leq i, j \leq n$, 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). Only one input symbol σ_0 is used; this goes into the input queue of Z_{n+1}, which, in turn, sends $[\sigma_0, i]$ to each CSXM Z_i and so initializes their computation, by processing the strings corresponding to their initial multisets. Each computation step in $k\Pi$ is reproduced by a number of transitions in S_{n+1}. Finally, when the kP system stops the computation, and the multiset w is obtained in C_{i_0}, then S_{n+1} moves to a final state and the result is sent out as an output sequence, $s(w)$.

We now briefly discuss the implementation of CSXMSs in FLAME. Basically, there are two restrictions that the FLAME implementation places on CSXMSs: (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. As explained above, a kP system is transformed into a communicating X-machine system by constructing, for each membrane, a communicating X-machine that simulates its behaviour; an additional X-machine, used for the synchronization of the others, is also used. In FLAME, however, the additional X-machine is no longer needed since the synchronization is achieved through message passing - for more details see Section 3.1 and Appendix.

3 Tools used for kP system models

The kP system models are specified using a machine readable representation, called *kP–Lingua* [5]. A slightly modified version of an example from [5] is presented below, showing how various kP systems concepts are represented in kP–Lingua.

Example 1. A type definition in kP–Lingua.

```
type C1 {
    choice {
        > 2b : 2b -> b, a(C2) .
         b ->  2b .
    }
  }
  type C2 {
    choice {
      a -> a, {b, 2c}(C1) .
    }
 }
m1 {2x, b} (C1) - m2 {x} (C2) .
```

Example 1 shows two compartment types, `C1`, `C2`, with corresponding instances `m1`, `m2`, respectively. The instance `m1` starts with the initial multiset `2x`, `b` and `m2` with an `x`. The rules of `C1` are selected non-deterministically, only one at a time. The first rule is executed only when its guard is true, i.e., only when the current multiset has at least three `b`'s. This rule also sends an `a` to the instance of the type `C2` linked to it. In `C2`, there is only a rule which is executed only when there is an `a` in the compartment.

The specifications written in kP–Lingua can be simulated and formally verified using a model checker called Spin. In this paper, we show two further extensions, another verification mechanism based on the NuSMV model checker [4] and a large scale simulation environment using Flame [6]. These two tools are are integrated into KPWorkbench, which can be downloaded from the KPWorkbench web page [14].

3.1 Simulation

The ability of simulating kernel P systems is one important aspect provided by a set of tools supporting this formalism. Currently, there are two different simulation approaches (a performance comparison can be found in [1]). Both receive as input a kP–Lingua model and outputs a trace of the execution, which is mainly used for checking the evolution of a system and for extracting various results out of the simulation.

KPWorkbench Simulator. KPWorkbench contains a simulator for kP system models and is written in the C# language. The simulator is a command line tool, providing a means for configuring the traces of execution for the given model, allowing the user to explicitly define the granularity of the output information by setting the values for a concrete set of parameters:

- *Steps* - a positive integer value for specifying the maximum number of steps the simulation will run for. If omitted, it defaults to 10.
- *SkipSteps* - a positive integer value representing the number of steps to skip the output generation. By using this parameter, the simulation trace will be generated from the step next to the currently specified one, onward. If not set, the default value is 0.

- *RecordRuleSelection* - defaulting to *true*, takes a boolean value on which to decide if the rule selection mechanism defined by the execution strategy will be generated into the output trace.
- *RecordTargetSelection* - if *true* (which is also the default value), traces the resolution of the communicating rules, outputting the non-deterministically selected membrane of a specified type to send the objects to.
- *RecordInstanceCreation* - defaulting to *true*, specifies if the membrane creation processes should be recorded into the output simulation trace.
- *RecordConfigurations* - if *true* (being also the default setting), generates as output, at the end of a step, the multiset of objects corresponding to each existing membrane.
- *ConfigurationsOnly* - having precedence over the other boolean parameters, sets the value of the above flags to *false*, except the one of the *RecordConfigurations*, causing the multiset configuration for each of the existing membranes to be the only output into the simulation trance. The default value is *false*.

FLAME-based Simulator. The agent-based modeling framework FLAME can be used to simulate kP–Lingua specifications. One of the main advantages of this approach is the high scalability degree and efficiency for simulating large scale models.

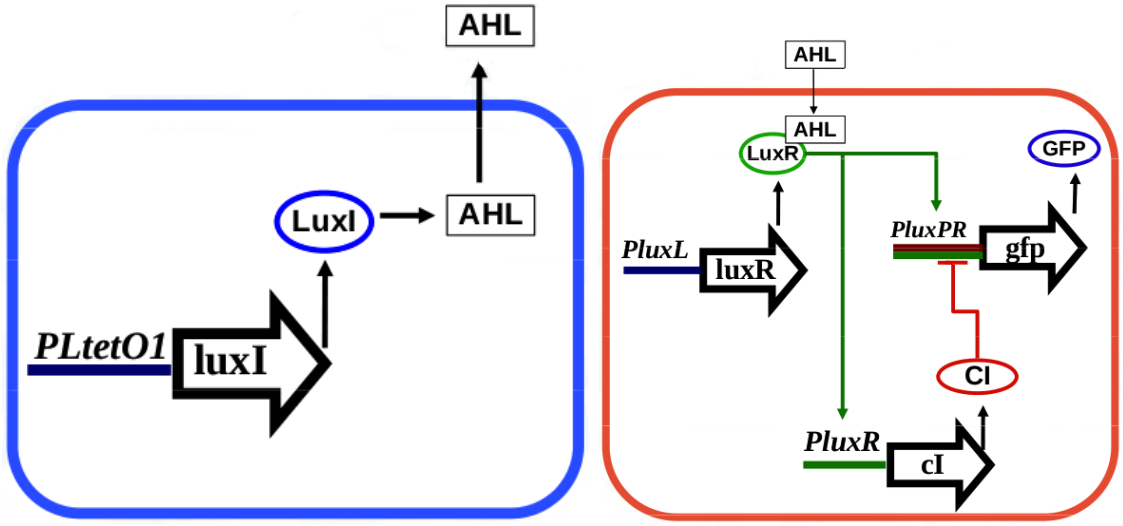
A general FLAME simulation requires the provision of a model for specifying the agents representing the definitions of communicating X-machines, whose behaviour is to be simulated, together with the input data representing the initial values of the memory for the generated X-machines. The model specification is composed of an xml file defining the structure of the agents, while their behaviour is provided as a set of functions in the C programming language.

In order to be able to simulate kernel P system models using the FLAME framework, an automated model translation has been implemented for converting the kP–Lingua specification into the above mentioned formats. Thus, the various compartments defined into the kP-Lingua model are translated into agent definitions, while the rule execution strategies corresponds to the transitions describing the behaviour of the agents. More specifically, each membrane of the kP system is represented by an agent. The rules are stored together with the membrane multiset as agent data. For each type of membrane from the kP system, a type of agent is defined, and for each execution strategy of the membrane, states are created in the X-machine. Transitions between the two states are represented by C functions that are executed in FLAME when passing from one state to another. Each type of strategy defines a specific function that applies the rules according to the execution strategy. A detailed description of the algorithm for translating a kP system into FLAME is given in the Appendix.

Each step of the simulation process modifies the memory of the agents, generating at the same time output xml files representing the configuration of the corresponding membranes at the end of the steps. The granularity level of the information defining the simulation traces is adjustable by providing a set of concrete parameters for the input data set.

3.2 Model Checking

KPWORKBENCH already integrates the SPIN model checker [10]. A more detailed account can be found in [5]. In this paper, we also integrate the NUSMV model checker [4] to the KPWORKBENCH platform to be able to verify branching-time semantics. NUSMV is designed to verify synchronous and asynchronous systems. Its high-level modelling language is based on *Finite State Machines* (FSM) and allows the description of systems in a modular and hierarchical manner. NUSMV supports the analysis of specification expressed in *Linear-time Temporal Logic* (LTL) and *Computation Tree Logic* (CTL). NUSMV employs *symbolic* methods, allowing a compact representation of the state space to increase the efficiency and performance. The tool also permits conducting



(a) Sender cell. (b) Pulsing cell.

Fig. 1: Two cell types of the pulse generator system (taken from [3]).

simulation experiments over the provided FSM model by generating traces either interactively or randomly.

We note that the NuSMV tool is currently considered for a restricted subset of the kP-Lingua language, and we only consider one execution strategy, *nondeterministic choice*.

4 Case study: Pulse Generator

The *pulse generator* [2] is a synthetic biology system, which was analysed stochastically in [3, 13]. It is composed of two types of bacterial strains: *sender* and *pulsing* cells (see Figure 1). The sender cells produce a signal (`3OC6-HSL`) and propagates it through the pulsing cells, which express the green fluorescent protein (`GFP`) upon sensing the signal. The excess of the signalling molecules are propagated to the neighbouring cells.

Sender cells synthesize the signalling molecule `3OC6-HSL` (`AHL`) through the enzyme `LuxI`, expressed under the constitutive expression of the promoter `PLtetO1`. Pulsing cells express `GFP` under the regulation of the `PluxPR` promoter, activated by the `LuxR_3OC6_2` complex. The `LuxR` protein is expressed under the control of the `PluxL` promoter. The `GFP` production is repressed by the transcription factor `CI`, codified under the regulation of the promoter `PluxR` that is activated upon binding of the transcription factor `LuxR_3OC6_2`.

4.1 Stochastic model

The formal model consists of two bacterial strains, each one is represented by an SP system model. So, $L = \{sender, pulsing\}$, describes these two labels and accordingly:

$$SP_h = (O_h, M_h, R_h), h \in L \tag{4}$$

where

$$O_{sender} = \{\texttt{PLtetO1\_geneLuxI}, \texttt{proteinLuxI}, \texttt{proteinLuxI\_Rib}, \texttt{rnaLuxI}, \texttt{rnaLuxI\_RNAP}, \texttt{signal3OC6}\}$$

$$M_{sender} = \texttt{PLtetO1\_geneLuxI}$$

$$
\begin{aligned}
R_{sender} = \{ & r_1 : \texttt{PLtetO1\_geneLuxI} \xrightarrow{\texttt{k}_1} \texttt{PLtetO1\_geneLuxI} + \texttt{rnaLuxI\_RNAP} \;\; \texttt{k}_1 = 0.1, \\
& r_2 : \texttt{rnaLuxI\_RNAP} \xrightarrow{\texttt{k}_2} \texttt{rnaLuxI} \;\; \texttt{k}_2 = 3.36, \\
& r_3 : \texttt{rnaLuxI} \xrightarrow{\texttt{k}_3} \texttt{rnaLuxI} + \texttt{proteinLuxI\_Rib} \;\; \texttt{k}_3 = 0.0667, \\
& r_4 : \texttt{rnaLuxI} \xrightarrow{\texttt{k}_4} \;\; \texttt{k}_4 = 0.004, \\
& r_5 : \texttt{proteinLuxI\_Rib} \xrightarrow{\texttt{k}_5} \texttt{proteinLuxI} \;\; \texttt{k}_5 = 3.78, \\
& r_6 : \texttt{proteinLuxI} \xrightarrow{\texttt{k}_6} \;\; \texttt{k}_6 = 0.067, \\
& r_7 : \texttt{proteinLuxI} \xrightarrow{\texttt{k}_7} \texttt{proteinLuxI} + \texttt{signal3OC6} \;\; \texttt{k}_7 = 5\}
\end{aligned}
$$

and

$$
\begin{aligned}
O_{pulsing} = \{ & \texttt{CI2}, \texttt{LuxR2}, \texttt{PluxL\_geneLuxR}, \texttt{PluxPR\_CI2\_geneGFP}, \\
& \texttt{PluxPR\_LuxR2\_CI2\_geneGFP}, \texttt{PluxPR\_LuxR2\_geneGFP}, \texttt{PluxPR\_geneGFP}, \\
& \texttt{PluxR\_LuxR2\_geneCI}, \texttt{PluxR\_geneCI}, \texttt{proteinCI}, \texttt{proteinCI\_Rib}, \texttt{proteinGFP}, \\
& \texttt{proteinGFP\_Rib}, \texttt{proteinLuxR}, \texttt{proteinLuxR\_3OC6}, \texttt{proteinLuxR\_Rib}, \texttt{rnaCI}, \\
& \texttt{rnaCI\_RNAP}, \texttt{rnaGFP}, \texttt{rnaGFP\_RNAP}, \texttt{rnaLuxR}, \texttt{rnaLuxR\_RNAP}, \texttt{signal3OC6}\}
\end{aligned}
$$

$$
M_{pulsing} = \texttt{PluxL\_geneLuxR}, \texttt{PluxR\_geneCI}, \texttt{PluxPR\_geneGFP}.
$$

The set of rules ($R_{pulsing}$) is presented in Table 6 (Appendix).
The translocation rules are:

$$
\begin{aligned}
Tr = \{ & r_1 : [\texttt{signal3OC6}]_{sender} \xrightarrow{\texttt{k}_1} [\texttt{signal3OC6}]_{pulsing} \;\; \texttt{k}_1 = 1.0, \\
& r_2 : [\texttt{signal3OC6}]_{sender} \xrightarrow{\texttt{k}_2} [\texttt{signal3OC6}]_{sender} \;\; \texttt{k}_2 = 1.0, \\
& r_3 : [\texttt{signal3OC6}]_{pulsing} \xrightarrow{\texttt{k}_3} [\texttt{signal3OC6}]_{pulsing} \;\; \texttt{k}_3 = 1.0\}.
\end{aligned}
$$

The lattice, given by *Lat*, is an array with n rows and m columns of coordinates (a, b), where $0 \leq a \leq n-1$ and $0 \leq b \leq m-1$. The values n and m will be specified for various experiments conducted in this paper. If we assume that the first column is associated with *sender* SP systems and the rest with *pulsing* systems, we formally express this as follows: $Pos(a, 0) = SP_{sender}$, $0 \leq a \leq n-1$, and $Pos(a, b) = SP_{pulsing}$, $0 \leq a \leq n-1$ and $1 \leq b \leq m-1$.

4.2 Nondeterministic model

Non-deterministic models are used for qualitative analysis. They are useful for detecting the existence of molecular species rather than for measuring their concentration. A typical non-deterministic model can be obtained from a stochastic model by removing the kinetics constants.

More precisely, one can define two types corresponding to the two bacterial strains in accordance with Definition 4, namely $T = \{sender, pulsing\}$, and the corresponding rule sets, R'_{sender} and $R'_{pulsing}$. The rules from R'_{sender} are obtained from R_{sender} and $r_1, r_2 \in Tr$, and those from $R'_{pulsing}$ are obtained from $R_{pulsing}$ and $r_3 \in Tr$, by removing the kinetic rates. For each rule from the set Tr, namely $r_k : [x]_{h_1} \xrightarrow{c_k} [x]_{h_2}$, the corresponding rule of the kP system will be $r_k : x \rightarrow x(t)$, where $t \in T$. The execution strategies are those described in the associated definitions of the kP systems.

The kP system with $n \times m$ components is given, in accordance with Definition 3, by the graph with vertices $C_{a,b} = (t_{a,b}, w_{a,b})$, where $t_{a,b} \in T$ and $w_{a,b}$ is the initial multiset, $0 \leq a \leq n-1$, $0 \leq b \leq m-1$; and edges where each component $C_{a,b}$, with $0 \leq a \leq n-1$, $0 \leq b \leq m-2$, is connected to its east neighbor, $C_{a,b+1}$, and each component $C_{a,b}$, with $0 \leq a \leq n-2$, $0 \leq b \leq m-1$ is connected to the south neighbor, $C_{a+1,b}$. The types of these components are $t_{a,0} = sender$, $0 \leq a \leq n-1$, and $t_{a,b} = pulsing$, $0 \leq a \leq n-1$ and $1 \leq b \leq m-1$. The initial multisets are $w_{a,0} = M_{sender}$, $0 \leq a \leq n-1$, and $w_{a,b} = M_{pulsing}$, $0 \leq a \leq n-1$ and $1 \leq b \leq m-1$.

So, one can observe the similitude between the lattice and function Pos underlying the definition of the LPP system and the graph and the types associated with the kP system.

4.3 Nondeterministic simplified model

In order to relieve the state explosion problem, models can also be simplified by replacing a long chain of reactions by a simpler rule set which will capture the starting and ending parts of this chain, and hence eliminating species that do not appear in the new rule set. With this transformation we achieve a simplification of the state space, but also of the number of transitions associated with the model.

The non-deterministic system with a set of compacted rules for the sender cell is obtained from the kP system introduced above and consists of the same graph with the same types, T, and initial multisets, $w_{a,b}$, $0 \leq a \leq n-1$, $0 \leq b \leq m-1$, but with simplified rule sets obtained from R'_{sender} and $R'_{pulsing}$, denoted R''_{sender} and $R''_{pulsing}$, respectively, where R''_{sender} is defined as follows:

$$\begin{aligned} R''_{sender} = \{ & r_1 : \texttt{PLtetO1\_geneLuxI} \rightarrow \texttt{PLtetO1\_geneLuxI} + \texttt{rnaLuxI\_RNAP}, \\ & r_2 : \texttt{proteinLuxI} \rightarrow, \\ & r_3 : \texttt{proteinLuxI} \rightarrow \texttt{proteinLuxI} + \texttt{signal3OC6}, \\ & r_4 : \texttt{signal3OC6} \rightarrow \texttt{signal3OC6}\ (\texttt{pulsing}), \\ & r_5 : \texttt{signal3OC6} \rightarrow \texttt{signal3OC6}\ (\texttt{sender})\} \end{aligned}$$

and $R''_{pulsing}$ is defined in Table 7 (Appendix).

5 Experiments

5.1 Simulation

The simulation tools have been used to check the temporal evolution of the system and to infer various information from the simulation results. For a kP system of 5×10 components, which comprises 25 sender cells and 25 pulsing cells, we have observed the production and transmission of the signalling molecules from the sender cells to the furthest pulsing cell and the production of the green florescent protein.

FLAME Results. As explained before, in FLAME each agent is represented by an X-machine. When an X-machine reaches its final state, the data is written to the hard disk and then used as input for the next iteration. Since the volume of data increases with the number of membranes, the more membranes we have, the more time for reading and writing the data (from or to the hard disk) is required. Consequently, when the number of membranes is large, the time required by the read and write operations increases substantially, so the simulation may become infeasible[4]. For example, for the pulsing generator system it was difficult to obtain simulation results after 100,000 steps; the execution time for 100,000 steps was approximately one hour.

The signalling molecule `signal3OC6` appeared for the first time in the sending cell $\texttt{sender}_{1,1}$ at the 27th step; after that, it appeared and disappeared many times. In the pulsing cell $\texttt{pulse}_{5,9}$, the signalling molecule appeared for the first time at 4963 steps, while `proteinGFP` was produced for the first time after 99,667 steps.

The results of the FLAME simulation show that the signaling molecules produced in the sending cells are propagated to the pulsating cells which, in turn, produce `proteinGFP`. The results of the simulation are given in Table 1. In 100,000 steps (the maximum number of steps considered for the FLAME simulation), the farthest cell in which `proteinGFP` was produced was $\texttt{pulse}_{5,9}$ - this was produced after 99,667 steps.

[4] On the other hand, the distributed architecture of FLAME allows the simulation to be run on parallel supercomputers with great performance improvements, but this is beyond the scope of this paper.

Table 1: FLAME results.

	$\texttt{sender}_{1,1}$	$\texttt{pulse}_{5,9}$	
Step Interval	`signal3OC6`	`signal3OC6`	`proteinGFP`
0 – 10,000	Exist	Exist	None
10,001 – 20,000	Exist	Exist	None
20,001 – 30,000	Exist	Exist	None
30,001 – 40,000	Exist	Exist	None
40,001 – 50,000	Exist	Exist	None
50,001 – 60,000	Exist	Exist	None
60,001 – 70,000	Exist	Exist	None
70,001 – 80,000	Exist	Exist	None
80,001 – 90,000	Exist	Exist	None
90,001 – 99,666	Exist	Exist	None
99,667 – 100,000	Exist	Exist	Exist

Table 2: KPWorkbench Simulator results.

	$\texttt{sender}_{1,1}$	$\texttt{pulse}_{5,10}$	
Step Interval	`signal3OC6`	`signal3OC6`	`proteinGFP`
0 – 300,000	Exist	Exist	None
300,001 – 600,000	Exist	Exist	None
600,001 – 900,000	Exist	Exist	None
900,001 – 1,200,000	Exist	Exist	None
1,200,001 – 1,500,000	Exist	Exist	None
1,500,001 – 1,800,000	Exist	Exist	None
1,800,001 – 2,100,000	Exist	Exist	None
2,100,001 – 2,400,000	Exist	Exist	None
2,400,001 – 2,700,000	Exist	Exist	Exist
2,700,001 – 3,000,000	Exist	Exist	None

KPWorkbench Simulator Results. KPWorkbench Simulator is a specialised simulation tool and provides better results, in terms of execution time, then a general purpose simulation environment like FLAME. This is mainly due to the fact that this approach makes the simulation to be performed in a single memory space, that scales according to the number of membranes used in the model and the number of objects resulting from applying the rules in each simulation step.

Table 2 presents the production and availability of the signaling molecule at the first sender cell (i.e. $\texttt{sender}_{1,1}$) and the transmission of the signaling molecule and the production of the green florescent protein at the furthest pulsing cell (i.e. $\texttt{pulse}_{5,10}$).

We have run the simulator for 3,000,000 time steps. The $\texttt{sender}_{1,1}$ cell was able to produce the signaling molecule at 22 steps, and later produced more signaling molecules. The $\texttt{pulse}_{5,10}$ cell, as the furthest pulse generator cell, was able to receive the signaling molecule at the 5474 step. But, the production of `proteinGFP` was possible at the 2,476,813 step, and it remained inside the cell until the 2,476,951 step, then it was consumed.

The simulation results show that the signaling molecule can be produced and transmitted by a sender cell. In addition, a pulse generator cell can have a signaling molecule only after a sender cell sends it, and can use the signal for the production of `proteinGFP` in later steps.

Table 3: Property patterns used in the verification experiments.

Prop.	Informal specification and the corresponding CTL translations
1	`X` *will eventually be produced.* `EF X>0`
2	*The availability/production of* `X` *will eventually lead to the production of* `Y`. `AG (X` $\Rightarrow$ `EF Y)`
3	`Y` *cannot be produced before* `X` *is produced.* $\neg$ `E (X=0 U Y>0)`

5.2 Verification

The properties of the system are verified using the NuSMV model checker, fully integrated into the KPWorkbench platform. In this section, we first verify individual cell properties, and then verify the properties of the whole kP system, involving multiple cells that are distributed within the lattice and interact with each other via the translocation rules.

Our verification results show that when the cell population is small, the properties can be verified using reasonable computational resources. However, given that the complete rule set is used, when the number of cell increases, verification becomes no longer feasible due to the state explosion problem. To mitigate this problem, we have used a simplified rule set to verify the cell interaction properties when the cell population is large.

Complete Rule Sets. Experiments under this section are conducted on a small population of multi-cellular systems including the complete set of rules. We have analysed two pulse-generator systems that differ only in the number of pulse generator cells. The first group consists of one sender cell and one pulse generator cell, i.e. 1×2 components, whereas the second group has one more pulse generator cell, i.e. 1×3 components.

For our experiments, we use the property patterns provided in Table 3. Table 4 shows the verification results for the properties given in Table 3 using two different groups. NuSMV has returned True for all the properties. In the group with 1×2 components, we have verified that the sender cell ($\texttt{sender}_1$) can produce a signalling molecule and transmit it to the pulsing cell ($\texttt{pulsing}_1$). In addition, the pulse generator cell can use that signal to produce the green florescent protein (`proteinGFP`). In the group with 1×3 components, we have verified similar properties. In addition, we have verified that the first pulsing cell ($\texttt{pulsing}_1$) can transmit the signalling molecule to the second pulsing cell ($\texttt{pulsing}_2$).

Reduced Rule Sets. Using a larger sets of components, we want to prove that the signalling molecules can be transmitted to the furthest pulsing cells. However, when we increase the number of cells, verification becomes no longer feasible due to the state explosion problem. In order to achieve the verification results within a reasonable time, we have compacted the rules sets such that an entire chain of reactions is replaced by a fewer simple rules. Consequently, the overall number of interactions is reduced and all the species which do not appear in the new set of rules are removed from the model. These changes are made in the non-deterministic models as these are used for qualitative analyses where the concentration of certain molecules is not significant or chain of reaction already analysed can be replaced by some abstractions mimicking their behaviour through simpler rewriting mechanisms.

Here, we define a group of cells with 1×5 components, where 1 sender and 4 pulsing cells are placed in row. For this scenario, we could verify the same properties in Table 4 using the reduced rule sets (as defined in Section 4.3). In addition, we have verified additional properties to analyse

Table 4: Verification experiments for the complete rule sets.

Lattice	Property	X, Y
1×2	Prop. 1	X = sender$_1$.signal3OC6 X = pulsing$_1$.signal3OC6
	Prop. 2	X = pulsing$_1$.signal3OC6, Y = pulsing$_1$.proteinGFP X = sender$_1$.signal3OC6, Y = pulsing$_1$.proteinGFP
	Prop. 3	X = pulsing$_1$.signal3OC6, Y = pulsing$_1$.proteinGFP X = sender$_1$.signal3OC6, Y = pulsing$_1$.proteinGFP
1×3	Prop. 1	X = pulsing$_2$.signal3OC6 X = pulsing$_2$.proteinGFP
	Prop. 2	X = pulsing$_1$.signal3OC6, Y = pulsing$_2$.proteinGFP X = sender$_1$.signal3OC6, Y = pulsing$_2$.proteinGFP
	Prop. 3	X = pulsing$_1$.signal3OC6, Y = pulsing$_2$.signal3OC6 X = sender$_1$.signal3OC6, Y = pulsing$_2$.signal3OC6

Table 5: Verification experiments for the reduced rule sets.

Lattice	Property	X, Y
1×5	Prop. 1	X = pulsing$_4$.signal3OC6 X = pulsing$_4$.proteinGFP
	Prop. 2	X = pulsing$_1$.signal3OC6, Y = pulsing$_4$.proteinGFP X = sender$_1$.signal3OC6, Y = pulsing$_4$.proteinGFP
	Prop. 3	X = pulsing$_3$.signal3OC6, Y = pulsing$_4$.signal3OC6 X = pulsing$_3$.signal3OC6, Y = pulsing$_4$.proteinGFP

the other pulsing cells. Table 5 shows these properties, for which NuSMV has returned True. The verification results show that the sender cell can produce the signalling molecule and transmit it to the adjacent pulsing cell, and the pulsing cells can use the signalling molecule to produce `proteinGFP` and transmit it to the its neighbour pulsing cells.

6 Conclusions

In this paper, we have presented two extensions to KPWorkbench: a formal verification tool based on the NuSMV model checker and a large scale simulation environment using FLAME, a platform for agent-based modelling on parallel architectures. The use of these two tools for modelling and analysis of biological systems is illustrated with a pulse generator, a synthetic biology system. We have provided both the stochastic model as stochastic P systems and the non-deterministic model as kernel P systems as well as a reduced model. We have also provided both simulation and verification results, confirming the desired behaviour of the pulse generator system.

The NuSMV tool currently works for a restricted subset of the kP-Lingua language, and we only consider one execution strategy, the nondeterministic choice. As a future work, we will extend the compatibility of the tool to cover the full language and the other execution strategies, e.g. sequence and maximal parallelism. The future work will also involve modeling, simulation and verification of even more complex biological systems as well as performance comparisons of simulators and model checking tools integrated within KPWorkbench.

Acknowledgements. The work of FI, LM and IN was supported by a grant of the Romanian National Authority for Scientific Research, CNCS-UEFISCDI (project number: PN-II-ID-PCE-2011-3-0688). SK acknowledges the support provided for synthetic biology research by EPSRC

ROADBLOCK (project number: EP/I031812/1). MB is supported by a PhD studentship provided by the Turkey Ministry of Education.

The authors would like to thank Marian Gheorghe for his valuable comments to this paper.

References

1. Bakir, M.E., Konur, S., Gheorghe, M., Niculescu, I., Ipate, F.: High performance simulations of kernel P systems. In: The 16th IEEE International Conference on High Performance Computing and Communications (2014)
2. Basu, S., Mehreja, R., Thiberge, S., Chen, M.T., Weiss, R.: Spatio-temporal control of gene expression with pulse-generating networks. PNAS 101(17), 6355–6360 (2004)
3. Blakes, J., Twycross, J., Konur, S., Romero-Campero, F.J., Krasnogor, N., Gheorghe, M.: Infobiotics Workbench: A P systems based tool for systems and synthetic biology. In: [7], pp. 1–41. Springer (2014)
4. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV version 2: An open source tool for symbolic model checking. In: Proc. International Conference on Computer-Aided Verification (CAV 2002). LNCS, vol. 2404. Springer, Copenhagen, Denmark (July 2002)
5. Dragomir, C., Ipate, F., Konur, S., Lefticaru, R., Mierlă, L.: Model checking kernel P systems. In: 14th International Conference on Membrane Computing. LNCS, vol. 8340, pp. 151–172. Springer (2013)
6. FLAME: Flexible large-scale agent modeling environment (available at `http://www.flame.ac.uk/`)
7. Frisco, P., Gheorghe, M., Pérez-Jiménez, M.J. (eds.): Applications of Membrane Computing in Systems and Synthetic Biology. Springer (2014)
8. Gheorghe, M., Ipate, F., Dragomir, C., Mierlă, L., Valencia-Cabrera, L., García-Quismondo, M., Pérez-Jiménez, M.J.: Kernel P systems - Version 1. 12th BWMC pp. 97–124 (2013)
9. Gillespie, D.: A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. Journal of Computational Physics 22(4), 403–434 (1976)
10. Holzmann, G.J.: The model checker SPIN. IEEE Transactions on Software Engineering 23(5), 275–295 (1997)
11. Ipate, F., Bălănescu, T., Kefalas, P., Holcombe, M., Eleftherakis, G.: A new model of communicating stream X-machine systems. Romanian Journal of Information Science and Technology 6, 165–184 (2003)
12. Konur, S., Gheorghe, M., Dragomir, C., Ipate, F., Krasnogor, N.: Conventional verification for unconventional computing: a genetic XOR gate example. Fundamenta Informaticae (2014)
13. Konur, S., Gheorghe, M., Dragomir, C., Mierla, L., Ipate, F., Krasnogor, N.: Qualitative and quantitative analysis of systems and synthetic biology constructs using P systems. ACS Synthetic Biology (2014)
14. KPWorkbench (available at `http://kpworkbench.org`)
15. Niculescu, I.M., Gheorghe, M., Ipate, F., Stefanescu., A.: From kernel P systems to X-machines and FLAME. Journal of Automata, Languages and Combinatorics To appear (2014)
16. Păun, G.: Computing with membranes. Journal of Computer and System Sciences 61(1), 108–143 (2000)
17. Păun, G., Rozenberg, G., Salomaa, A. (eds.): The Oxford Handbook of Membrane Computing. Oxford University Press (2010)
18. Romero-Campero, F.J., Twycross, J., Cao, H., Blakes, J., Krasnogor, N.: A multiscale modeling framework based on P systems. In: Membrane Computing, LNCS, vol. 5391, pp. 63–77. Springer (2009)
19. Sanassy, D., Fellermann, H., Krasnogor, N., Konur, S., Mierlă, L., Gheorghe, M., Ladroue, C., Kalvala, S.: Modelling and stochastic simulation of synthetic biological boolean gates. In: The 16th IEEE International Conference on High Performance Computing and Communications (2014)

Appendix

Algorithm 1 Transforming a kP Systems into Flame algorithm

```
1: procedure AddTtransition(startState, stopState, strategy, guard)
   ▷ procedure adding the appropriate transition strategy to the current agent stack given as parameter
     and FLAME function applying rules conforming to execution strategy
   ▷ guard is an optional parameter that represents the transition guard
2:     if strategy is Sequence then
3:         agentTtransitions.Push(startState, stopState, SequenceFunction, guard)
           ▷ FLAME function SequenceFunction applies rules in sequentially mode
4:     else if strategy is Choice then
5:         agentTtransitions.Push(startState, stopState, ChoiceFunction, guard)
           ▷ FLAME function ChoiceFunction applies rules in choice mode
6:     else if strategy is ArbitraryParallel then
7:         agentTtransitions.Push(startState, stopState, ArbitraryParallelFunction, guard)
           ▷ FLAME function ArbitraryParallelFunction applies rules in arbitrary parallel mode
8:     else if strategy is MaximalParallel then
9:         agentTtransitions.Push(startState, stopState, MaximalParallelFunction, guard)
           ▷ FLAME function MaximalParallelFunction applies rules in maximal parallel mode
10:    end if
11: end procedure
12:
   ▷ main algorithm for traforming a kP system into Flame
13:
14: agentsStates.Clear()
15: agentsTtransitions.Clear()
   ▷ empty state and transition stacks of agents
16: foreach membrane in kPSystem do
   ▷ for each membrane of kP system build corresponding agent, consisting of states and transitions
17:    agentStates.Clear()
18:    agentTtransitions.Clear()
       ▷ empty state and transition stacks of agent that is built for the current membrane
19:    agentStates.Push(startState)
       ▷ adding the initial state of the X machine
20:    agentStates.Push(initializationState)
       ▷ adding initialization state
21:    agentTtransitions.Push(startState, initializationState)
       ▷ adding transition between the initial and initialization states; this transition performs objects
         allocation on rules and other initializations
22:    foreach strategy in membrane do
       ▷ for each strategy of the current membrane the corresponding states and transitions are built
23:        previousState = agentStates.Top()
           ▷ the last state is stored in a temporary variable
24:        if is first strategy and strategy.hasNext() then
           ▷ when the strategy is the first of several, state and transition corresponding to the execution
             strategy are added
25:            agentStates.Push(strategy.Name)
26:            AddTtransition(previousState, strategy.Name, strategy)
27:        else
28:            if not strategy.hasNext() then
               ▷ if it is the last strategy, the transition corresponding to the execution strategy is added
29:                AddTtransition(previousState, applyChangesState, strategy)
30:            else
```

Algorithm 1 Transforming a kP Systems into Flame algorithm (continued)

```
31:                 agentStates.Push(strategy.Name)
                  ▷ add corresponding state of the current strategy
32:                 if strategy.Previous() is Sequence then
                  ▷ verify that previous strategy is of sequence type
33:                     AddTtransition(previousState, strategy.Name, strategy, IsApplyAllRules)
                      ▷ add transition from preceding strategy state to the current strategy state. The guard
                        is active if all the rules have been applied in the previous strategy transition.
34:                     agentTtransitions.Push(previousState, applyChangesState, IsNotApplyAllRules)
                      ▷ add transition from preceding strategy state to state where changes produced by rules
                        are applied. The guard is active if not all rules have been applied in the previous
                        strategy transition
35:                 else
36:                     AddTtransition(previousState, strategy.Name, strategy)
                      ▷ add transition from preceding strategy state to the current strategy state
37:                     agentTtransitions.Push(previousState, applyChangesState, IsApplyStructureRule)
                      ▷ add transition from preceding state strategy to state in which changes produced by the
                        applied rules are committed. The guard is active when the structural rule has been
                        applied on the previous strategy transition
38:                 end if
39:             end if
40:         end if
41:     end for
42:     agentStates.Push(applyChangesState)
      ▷ adding state in which changes produced by the applied rules are committed
43:     agentTtransitions.Push(applyChangesState, receiveState)
      ▷ adding transition on which changes produced by the applied rules are committed
44:     agentStates.Push(receiveState)
      ▷ add state that receives objects sent by applying the communication rules in other membranes
45:     agentTtransitions.Push(receiveState, s0State)
      ▷ add transition that receives objects sent by applying the communication rules in other membranes
46:     agentStates.Push(s0State)
      ▷ add an intermediary state
47:     agentTtransitions.Push(s0State, endState, IsNotApplyStructureRule)
      ▷ add transition to the final state in which nothing happens unless a structural rule was applied
48:     agentTtransitions.Push(s0State, endState, IsApplyStructureRule)
      ▷ add the transition to the final state on which structural changes are made if the structure rule has
        been applied
49:     agentStates.Push(endState)
      ▷ add the final state
50:     agentsStates.PushAll(agentStates.Content())
      ▷ add the contents of the stack that holds the current agent states to the stack that holds the states
        of all agents
51:     agentsTtransitions.PushAll(agentStates.Content())
      ▷ add the contents of the stack that holds the current agent transitions to the stack that holds the
        transitions of all agents
52: end for
```

Table 6: Multiset rules ($R_{pulsing}$) of the SP systems model of the pulsing cell.

Rule	**Kinetic constant**
r_1 : PluxL_geneLuxR $\xrightarrow{k_1}$ PluxL_geneLuxR + rnaLuxR_RNAP	$k_1 = 0.1$
r_2 : rnaLuxR_RNAP $\xrightarrow{k_2}$ rnaLuxR	$k_2 = 3.2$
r_3 : rnaLuxR $\xrightarrow{k_3}$ rnaLuxR + proteinLuxR_Rib	$k_3 = 0.3$
r_4 : rnaLuxR $\xrightarrow{k_4}$	$k_4 = 0.04$
r_5 : proteinLuxR_Rib $\xrightarrow{k_5}$ proteinLuxR	$k_5 = 3.6$
r_6 : proteinLuxR $\xrightarrow{k_6}$	$k_6 = 0.075$
r_7 : proteinLuxR + signal3OC6 $\xrightarrow{k_7}$ proteinLuxR_3OC6	$k_7 = 1.0$
r_8 : proteinLuxR_3OC6 $\xrightarrow{k_8}$	$k_8 = 0.0154$
r_9 : proteinLuxR_3OC6 + proteinLuxR_3OC6 $\xrightarrow{k_9}$ LuxR2	$k_9 = 1.0$
r_{10} : LuxR2 $\xrightarrow{k_{10}}$	$k_{10} = 0.0154$
r_{11} : LuxR2 + PluxR_geneCI $\xrightarrow{k_{11}}$ PluxR_LuxR2_geneCI	$k_{11} = 1.0$
r_{12} : PluxR_LuxR2_geneCI $\xrightarrow{k_{12}}$ LuxR2 + PluxR_geneCI	$k_{12} = 1.0$
r_{13} : PluxR_LuxR2_geneCI $\xrightarrow{k_{13}}$ PluxR_LuxR2_geneCI + rnaCI_RNAP	$k_{13} = 1.4$
r_{14} : rnaCI_RNAP $\xrightarrow{k_{14}}$ rnaCI	$k_{14} = 3.2$
r_{15} : rnaCI $\xrightarrow{k_{15}}$ rnaCI + proteinCI_Rib	$k_{15} = 0.3$
r_{16} : rnaCI $\xrightarrow{k_{16}}$	$k_{16} = 0.04$
r_{17} : proteinCI_Rib $\xrightarrow{k_{17}}$ proteinCI	$k_{17} = 3.6$
r_{18} : proteinCI $\xrightarrow{k_{18}}$	$k_{18} = 0.075$
r_{19} : proteinCI + proteinCI $\xrightarrow{k_{19}}$ CI2	$k_{19} = 1.0$
r_{20} : CI2 $\xrightarrow{k_{20}}$	$k_{20} = 0.00554$
r_{21} : LuxR2 + PluxPR_geneGFP $\xrightarrow{k_{21}}$ PluxPR_LuxR2_geneGFP	$k_{21} = 1.0$
r_{22} : PluxPR_LuxR2_geneGFP $\xrightarrow{k_{22}}$ LuxR2 + PluxPR_geneGFP	$k_{22} = 1.0$
r_{23} : LuxR2 + PluxPR_CI2_geneGFP $\xrightarrow{k_{23}}$ PluxPR_LuxR2_CI2_geneGFP	$k_{23} = 1.0$
r_{24} : PluxPR_LuxR2_CI2_geneGFP $\xrightarrow{k_{24}}$ LuxR2 + PluxPR_CI2_geneGFP	$k_{24} = 1.0$
r_{25} : CI2 + PluxPR_geneGFP $\xrightarrow{k_{25}}$ PluxPR_CI2_geneGFP	$k_{25} = 5.0$
r_{26} : PluxPR_CI2_geneGFP $\xrightarrow{k_{26}}$ CI2 + PluxPR_geneGFP	$k_{26} = 0.0000001$
r_{27} : CI2 + PluxPR_LuxR2_geneGFP $\xrightarrow{k_{27}}$ PluxPR_LuxR2_CI2_geneGFP	$k_{27} = 5.0$
r_{28} : PluxPR_LuxR2_CI2_geneGFP $\xrightarrow{k_{28}}$ CI2 + PluxPR_LuxR2_geneGFP	$k_{28} = 0.0000001$
r_{29} : PluxPR_LuxR2_geneGFP $\xrightarrow{k_{29}}$ PluxPR_LuxR2_geneGFP + rnaGFP_RNAP	$k_{29} = 4.0$
r_{30} : rnaGFP_RNAP $\xrightarrow{k_{30}}$ rnaGFP	$k_{30} = 3.36$
r_{31} : rnaGFP $\xrightarrow{k_{31}}$ rnaX + proteinGFP_Rib	$k_{31} = 0.667$
r_{32} : rnaGFP $\xrightarrow{k_{32}}$	$k_{32} = 0.04$
r_{33} : proteinGFP_Rib $\xrightarrow{k_{33}}$ proteinGFP	$k_{33} = 3.78$
r_{34} : proteinGFP $\xrightarrow{k_{34}}$	$k_{34} = 0.0667$

Table 7: Multiset rules ($R''_{pulsing}$) of the kP systems model of the pulsing cell.

Rule
r_1 : PluxL_geneLuxR → PluxL_geneLuxR + rnaLuxR_RNAP
r_2 : proteinLuxR →
r_3 : proteinLuxR + signal3OC6 → proteinLuxR_3OC6
r_4 : proteinLuxR_3OC6 →
r_5 : proteinLuxR_3OC6 + PluxPR_geneGFP → PluxPR_LuxR2_geneGFP
r_6 : PluxPR_LuxR2_geneGFP → PluxPR_LuxR2_geneGFP + proteinGFP
r_7 : proteinGFP →
r_8 : signal3OC6 → signal3OC6 (pulsing)