# An Overview of P-Lingua 2.0

Manuel García-Quismondo, Rosa Gutiérrez-Escudero, Ignacio Pérez-Hurtado, Mario J. Pérez-Jiménez, Agustín Riscos-Núñez

Research Group on Natural Computing
Department of Computer Science and Artificial Intelligence
University of Sevilla
Avda. Reina Mercedes s/n, 41012, Sevilla, Spain
mangarfer2@alum.us.es, {rgutierrez,perezh,marper,ariscosn}@us.es

**Summary.** P–Lingua is a programming language for membrane computing which aims to be a standard to define P systems. In order to implement this idea, a Java library called pLinguaCore has been developed as a software framework for cell–like P systems. It is able to handle input files (either in XML or in P–Lingua format) defining P systems from a number of different cell–like P system models. Moreover, the library includes several built–in simulators for each supported model. For the sake of software portability, pLinguaCore can export a P system definition to any convenient output format (currently XML and binary formats are available). This software is not a closed product, but it can be extended to accept new input or output formats and also new models or simulators.

The term P–Lingua 2.0 refers to the software package consisting of the above mentioned library together with a user interface called pLinguaPlugin (more details can be found at http://www.p-lingua.org).

Finally, in order to illustrate the software, this paper includes an application using pLinguaCore for describing and simulating ecosystems by means of P systems.

## 1 Introduction

The initial definition of a *membrane system* as a computing device, introduced by Gh. Păun [14], can be interpreted as a flexible and general framework. Indeed, a large number of different models have been defined and investigated in the area: P systems with symport/antiport rules, with active membranes, with probabilistic rules, etc. There were some attempts to establish a common formalization covering most of the existing models (see e.g. [5]), but the membrane computing community is still using specific syntax and semantics depending on the model they work with.

Each model displays characteristic semantic constraints that determine the way in which rules are applied. Hence, the need for software simulators capable of taking into account different scenarios when simulating P system computations comes to the fore. Moreover, simulators have to precisely define the specific P system that is to be simulated. Along this paper, the term *simulator input* will

be used to refer to the definition (on a text file) of the P system to be simulated. One approach to implement the simulators input could be defining a specific input file format for each simulator. Nevertheless, this approach would require a great redundant effort. A second approach could be to standardize the simulator input, so all simulators need to process inputs specified in the same format. These two approaches raise up a trade-off: On the one hand, specific simulator inputs could be defined in a more straightforward way, as the used format is closer to the P system features to simulate. On the other hand, although the latter approach involves analyzing different P systems and models to develop a standard format, there is no need to develop completely a new simulator every time a new P system should be simulated, as it is possible to use a common software library in order to parse the standard input format. Moreover, users would not have to learn a new input format every time they use a different simulator and would not need to change the way to specify P systems which need to be simulated every time they move on to another model, as they would keep on using the standard input format.

This second approach is the one considered in P–Lingua project, a programming language whose first version, presented in [3], is able to define P systems within the active membrane P system model with division rules. The authors also provide software tools for compilation, simulation and debug tasks.

As P–Lingua is intended to become a standard for P systems definition, it should also consider other models. At the current stage, P–Lingua can define P systems within a number of different cell–like models: active membrane P systems with membrane division rules or membrane creation rules, transition P systems, symport/antiport P systems, stochastic P systems and probabilistic P systems. Each model follows semantics restrictions, which define several constraints for the rules (number of objects on each side, whether membrane creation and/or membrane division are allowed, and so on), and which indicate the way rules are applied on configurations.

A Java [22] library called pLinguaCore has been developed as a software framework for cell–like P systems. It includes parsers to handle input files (either in XML or in P–Lingua format), and furthermore the parsers check possible programming errors (both lexical/syntactical and semantical).

The library includes several built–in simulators to generate P system computations for the supported models, and it can export several output file formats to represent P systems (at the current stage, XML and binary file formats) in order to get interoperability between different software environments.

The term P–Lingua 2.0 refers to the software framework under GNU GPL license [21] consisting of the above mentioned library together with a user interface called pLinguaPlugin. It is not a closed software because developers with knowledge of Java can include new components to the library: new supported models, built–in simulators for the supported models, parsers to process new input file formats and generators for new output file formats. In order to facilitate those tasks, a website for users and developers of P–Lingua 2.0 [24] has been created. It

contains technical information about standard programming methods to expand the pLinguaCore library. These methods have been used on all the existent components. The website also contains a download section, tutorials, user manuals, information about projects using P–Lingua, and other useful stuff.

Furthermore, pLinguaCore is not a stand–alone product, it is created to be used inside other software applications. In order to illustrate this idea, the paper includes an application using pLinguaCore for describing and simulating ecosystems by means of P systems.

## 2 Models

### 2.1 Contemplating New models

The library pLinguaCore is able to accept input files (either in P–Lingua or XML file formats) that define P systems within the supported models. As mentioned in the Introduction, Java developers can include new models to the library by using standard programming methods, easing the task. The current supported models are enumerated below.

### 2.2 Transition P system model

The basic P systems were introduced in [14] by Gh. Păun.
A *transition P system* of degree $q \geq 1$ is a tuple of the form

$$\Pi = (\Gamma, L, \mu, \mathcal{M}_1, \ldots, \mathcal{M}_q, (R_1, \rho_1), \ldots, (R_q, \rho_q), i_o), \text{ where:}$$

- $\Gamma$ is an alphabet whose elements are called *objects*.
- $L$ is a finite set of labels.
- $\mu$ is a membrane structure consisting of $q$ membranes with the membranes (and hence the regions, the space between a membrane and the immediately inner membranes, if any) injectively labelled with elements of $L$; as usual, we represent the membrane structures by strings of matching labelled parentheses.
- $\mathcal{M}_i$, $1 \leq i \leq q$, are strings which represent multisets over $\Gamma$ associated with the $q$ membranes of $\mu$.
- $R_i$, $1 \leq i \leq q$, are finite sets of *evolution rules* over $\Gamma$, associated with the membranes of $\mu$. An evolution rule is of the form $u \to v$, where $u$ is a string over $\Gamma$ and $v = v'$ or $v = v'\delta$, being $v'$ a string over $\Gamma \times (\{here, out\} \cup \{in_j : 1 \leq j \leq q\})$.
- $\rho_i$, $1 \leq i \leq q$, are strict partial orders over $R_i$.
- $i_o$, $1 \leq i_o \leq q$, is the label of an elementary membrane (the *output membrane*).

The objects to evolve in a step and the rules by which they evolve are chosen in a non–deterministic manner, but in such a way that in each region we have a maximally parallel application of rules. This means that we assign objects to

rules, non–deterministically choosing the rules and the objects assigned to each rule, but in such a way that after this assignation no further rule can be applied to the remaining objects.

### 2.3 Symport/antiport P system model

Symport/antiport rules were incorporated in the framework of P systems in [13]. A *P system with symport/antiport rules* of degree $q \geq 1$ is a tuple of the form

$$\Pi = (\Gamma, L, \mu, \mathcal{M}_1, \ldots, \mathcal{M}_q, E, R_1, \ldots, R_q, i_o), \text{where:}$$

- $\Gamma$ is the alphabet of objects,
- $L$ is the finite set of labels for membranes (in general, one uses natural numbers as labels), $\mu$ is the membrane structure (of degree $q \geq 1$), with the membranes labelled in a one-to-one manner with elements of $L$,
- $\mathcal{M}_1, \ldots, \mathcal{M}_q$ are strings over $\Gamma$ representing the multisets of objects present in the $q$ compartments of $\mu$ in the initial configuration of the system.
- $E \subseteq \Gamma$ is the set of objects supposed to appear in the environment in arbitrarily many copies.
- $R_i$, $1 \leq i \leq q$, are finite sets of rules associated with the $q$ membranes of $\mu$. The rules can be of two types (by $\Gamma^+$ we denote the set of all non-empty strings over $\Gamma$, with $\lambda$ denoting the empty string):
  - *Symport rules*, of the form $(x, in)$ or $(x, out)$, where $x \in \Gamma^+$. When using such a rule, the objects specified by $x$ enter or exit, respectively, the membrane with which the rule is associated. In this way, objects are sent to or imported from the surrounding region – which is the environment in the case of the skin membrane.
  - *Antiport rules*, of the form $(x, out; y, in)$, where $x, y \in \Gamma^+$. When using such a rule for a membrane $i$, the objects specified by $x$ exit the membrane and those specified by $y$ enter from the region surrounding membrane $i$; this is the environment in the case of the skin membrane.
- $i_o \in L$ is the label of a membrane of $\mu$, which indicates the *output* region of the system.

The rules are used in the non-deterministic maximally parallel manner, standard in membrane computing.

### 2.4 Active membranes P system model

#### With membrane division rules

P systems with membrane division were introduced in [15], and in this model the number of membranes can increase exponentially in polynomial time. Next, we define P systems with active membranes using 2-division for elementary membranes,

with polarizations, but without cooperation and without priorities (and without permitting the change of membrane labels by means of any rule).
A *P system with active membranes* using 2-division for elementary membranes of degree $q \geq 1$ is a tuple $\Pi = (\Gamma, L, \mu, \mathcal{M}_1, \ldots, \mathcal{M}_q, R, i_o)$, where:

- $\Gamma$ is an alphabet of symbol-objects.
- $L$ is a finite set of labels for membranes.
- $\mu$ is a membrane structure, of $m$ membranes, labelled (not necessarily in a one-to-one manner) with elements of $L$.
- $\mathcal{M}_1, \ldots, \mathcal{M}_q$ are strings over $\Gamma$, describing the initial multisets of objects placed in the $q$ regions of $\mu$.
- $R$ is a finite set of rules, of the following forms:
  (a) $[\, a \to \omega\,]_h^\alpha$ for $h \in L$,$\alpha \in \{+, -, 0\}$, $a \in \Gamma$, $\omega \in \Gamma^*$: This is an object evolution rule, associated with a membrane labelled with $h$ and depending on the polarization of that membrane, but not directly involving the membrane.
  (b) $a[\ \ ]_h^{\alpha_1} \to [\, b\,]_h^{\alpha_2}$ for $h \in L$, $\alpha_1, \alpha_2 \in \{+, -, 0\}$, $a, b \in \Gamma$: An object from the region immediately outside a membrane labelled with $h$ is introduced in this membrane, possibly transformed into another object, and, simultaneously, the polarization of the membrane can be changed.
  (c) $[\, a\,]_h^{\alpha_1} \to b[\ \ ]_h^{\alpha_2}$ for $h \in L$, $\alpha_1, \alpha_2 \in \{+, -, 0\}$, $a, b \in \Gamma$: An object is sent out from membrane labelled with $h$ to the region immediately outside, possibly transformed into another object, and, simultaneously, the polarity of the membrane can be changed.
  (d) $[\, a\,]_h^\alpha \to b$ for $h \in L$, $\alpha \in \{+, -, 0\}$, $a, b \in \Gamma$: A membrane labelled with $h$ is dissolved in reaction with an object. The skin is never dissolved.
  (e) $[\, a\,]_h^{\alpha_1} \to [\, b\,]_h^{\alpha_2}\ [\, c\,]_h^{\alpha_3}$ for $h \in L$, $\alpha_1, \alpha_2, \alpha_3 \in \{+, -, 0\}$, $a, b, c \in \Gamma$: An elementary membrane can be divided into two membranes with the same label, possibly transforming some objects and the polarities.
- $i_o \in L$ is the label of a membrane of $\mu$, which indicates the *output* region of the system.

These rules are applied according to the following principles:

- All the rules are applied in parallel and in a maximal manner. In one step, one object of a membrane can be used by only one rule (chosen in a non-deterministic way), but any object which can evolve by one rule of any form, must do it (with the restrictions below indicated).
- If a membrane is dissolved, its content (multiset and internal membranes) is left free in the surrounding region.
- If at the same time a membrane labelled by $h$ is divided by a rule of type (e) and there are objects in this membrane which evolve by means of rules of type (a), then we suppose that the evolution rules of type (a) are used before division is produced. Of course, this process takes only one step.
- The rules associated with membranes labelled by $h$ are used for all copies of this membrane. At one step, a membrane can be the subject of *only one* rule of types (b)-(e).

**With membrane creation rules**

Membrane creation rules were first considered in [9], [10].
A *P system with membrane creation* of degree $q \geq 1$ is a tuple of the form

$$\Pi = (\Gamma, L, \mu, \mathcal{M}_1, \ldots, \mathcal{M}_q, R, i_o), \text{ where:}$$

- $\Gamma$ is the alphabet of objects.
- $L$ is a finite set of labels for membranes.
- $\mu$ is a membrane structure consisting of $q$ membranes labelled (not necessarily in a one-to-one manner) with elements of $L$.
- $\mathcal{M}_1, \ldots, \mathcal{M}_q$ are strings over $\Gamma$, describing the initial multisets of objects placed in the $q$ regions of $\mu$.
- $R$ is a finite set of rules of the following forms:
  (a) $[a \to v]_h$ where $h \in L$, $a \in \Gamma$, and $v$ is a string over $\Gamma$ describing a multiset of objects. These are *object evolution rules* associated with membranes and depending only on the label of the membrane.
  (b) $a[\,]_h \to [b]_h$ where $h \in L$, $a, b \in \Gamma$. These are *send-in communication rules*. An object is introduced in the membrane possibly modified.
  (c) $[a]_h \to [\,]_h\, b$ where $h \in L$, $a, b \in \Gamma$. These are *send-out communication rules*. An object is sent out of the membrane possibly modified.
  (d) $[a]_h \to b$ where $h \in L$, $a, b \in \Gamma$. These are *dissolution rules*. In reaction with an object, a membrane is dissolved, while the object specified in the rule can be modified.
  (e) $[a \to [v]_{h_2}]_{h_1}$ where $h_1, h_2 \in L$, $a \in \Gamma$, and $v$ is a string over $\Gamma$ describing a multiset of objects. These are *creation rules*. In reaction with an object, a new membrane is created. This new membrane is placed inside the membrane of the object which triggers the rule and has associated an initial multiset and a label.
- $i_o \in L$ is the label of a membrane of $\mu$, which indicates the *output* region of the system.

Rules are applied according to the following principles:

- Rules from (a) to (d) are used as usual in the framework of membrane computing, that is, in a maximally parallel way. In one step, each object in a membrane can only be used for applying one rule (non-deterministically chosen when there are several possibilities), but any object which can evolve by a rule of any form must do it (with the restrictions below indicated).
- Rules of type (e) are used also in a maximally parallel way. Each object $a$ in a membrane labelled with $h_1$ produces a new membrane with label $h_2$ placing in it the multiset of objects described by the string $v$.
- If a membrane is dissolved, its content (multiset and interior membranes) becomes part of the immediately external one. The skin membrane is never dissolved.

- All the elements which are not involved in any of the operations to be applied remain unchanged.
- The rules associated with the label $h$ are used for all membranes with this label, independently of whether or not the membrane is an initial one or it was obtained by creation.
- Several rules can be applied to different objects in the same membrane simultaneously. The exception are the rules of type $(d)$ since a membrane can be dissolved only once.

### 2.5 Probabilistic P system model

A probabilistic approach in the framework of P systems was first considered by A. Obtulowicz in [12].
A *probabilistic P system* of degree $q \geq 1$ is a tuple

$$\Pi = (\Gamma, \mu, \mathcal{M}_1, \ldots, \mathcal{M}_q, R, \{c_r\}_{r \in R}, i_o), \text{ where:}$$

- $\Gamma$ is the alphabet (finite and nonempty) of objects (the working alphabet).
- $\mu$ is a membrane structure, consisting of $q$ membranes, labeled $1, 2, \ldots, q$. The skin membrane is labeled by 0. We also associate electrical charges with membranes from the set $\{0, +, -\}$, neutral and positive.
- $\mathcal{M}_1, \ldots, \mathcal{M}_q$ are strings over $\Gamma$, describing the multisets of objects initially placed in the $q$ regions of $\mu$.
- $R$ is a finite set of evolution rules. An evolution rule associated with the membrane labelled by $i$ is of the form $r: \ u[\ v\ ]_i^\alpha \xrightarrow{c_r} u'[\ v'\ ]_i^\beta$, where $u, v, u', v'$ are a multiset over $\Gamma$, $\alpha, \beta \in \{0, +, -\}$ and $c_r$ is a real number between 0 and 1 associated with the rule such that:
  - for each $u, v \in M(\Gamma)$, $h \in H$ and $\alpha \in \{0, +\}$, if $r_1, \ldots, r_t$ are the rules whose left–hand side is $u[\ v\ ]_h^\alpha$, then $\sum_{j=1}^{t} c_{r_j} = 1$
- $i_o \in L$ is the label of a membrane of $\mu$, which indicates the *output* region of the system.

We assume that a global clock exists, marking the time for the whole system (for all compartments of the system); that is, all membranes and the application of all rules are synchronized.

The $q$-tuple of multisets of objects present at any moment in the $q$ *regions* of the system constitutes the *configuration* of the system at that moment. The tuple $(\mathcal{M}_1, \ldots, \mathcal{M}_q)$ is the initial configuration of the system.

We can pass from one configuration to another one by using the rules from $R$ as follows: at each transition step, the rules to be applied are selected according to the probabilities assigned to them, all applicable rules are simultaneously applied, and all occurrences of the left–hand side of the rules are consumed, as usual. Rules with the same left–hand side and whose right–hand side has the same polarization can be applied simultaneously.

### 2.6 Stochastic P System model

The original motivation of P systems was not to provide a comprehensive and accurate model of the living cell, but to imitate the computational nature of operations that take place in cell membranes. Most P system models have been proved to be Turing complete and computationally efficient, in the sense that they can solve computationally hard problems in polynomial time, by trading time for space. Most research in P systems focus on complexity classes and computational power.

However, P systems have been used recently to model biological phenomena very successfully. Models of oscillatory systems [4], signal transduction [18], gene regulation control [16], quorum sensing [17] and metapopulations [19] have been presented.

We introduce in this section the specification of stochastic P systems, that constitute the framework for modelling biological phenomena.
A *stochastic P system* of degree $q \geq 1$ is a tuple

$$\Pi = (\Gamma, L, \mu, \mathcal{M}_1, \ldots, \mathcal{M}_q, R_{l_1}, \ldots, R_{l_m}), \text{ where:}$$

- $\Gamma$ is a finite alphabet of symbols representing objects.
- $L = \{l_1, \ldots, l_m\}$ is a finite alphabet of symbols representing labels for the membranes.
- $\mu$ is a membrane structure containing $q \geq 1$ membranes identified in a one to one manner with values in $\{1, \ldots, q\}$ and labelled with elements from $L$.
- $M_i = (l_i, w_i, s_i)$, for each $1 \leq i \leq q$, initial configuration of the membrane $i$, $l_i \in L$ is the label, $w_i \in \Gamma^*$ is a finite multiset of objects and $s_i$ is a finite set of strings over $\Gamma$.
- $R_{l_t} = \{r_1^{l_t}, \ldots, r_{k_{l_t}}^{l_t}\}$, for each $1 \leq t \leq m$, is a finite set of rewriting rules associated with membranes of label $l_t \in L$. Rules are of one of the following two forms:
  - Multiset rewriting rules:

    $$r_j^{l_t} : u[w]_l \xrightarrow{c_j^{l_t}} u'[w']_l$$

    with $u, w, u', w' \in \Gamma^*$ some finite multisets of objects and $l$ a label from $L$. A multiset of objects, $u$ is represented as $u = a_1 + \cdots + a_m$, with $a_1, \ldots, a_m \in \Gamma$. The empty multiset will be denoted by $\lambda$ and we will write $o^n$ instead of $\overbrace{o + \cdots + o}^{n}$. The multiset $u$ placed outside of the membrane labelled with $l$ and the multiset $w$ placed inside of that membrane are simultaneously replaced with a multiset $u'$ and $w'$ respectively.
  - String rewriting rules:

    $$r_j^{l_t} : [u_1 + s_1; \ldots; u_p + s_p]_l \xrightarrow{c_j^{l_t}} [u'_1 + s'_{1,1} + \cdots + s'_{1,i_1}; \ldots; u'_p + s'_{p,1} + \cdots + s'_{p,i_p}]$$

A string $s$ is represented as $s = \langle o_1.o_2.\cdots.o_j\rangle$, where $o_1, o_2, \ldots, o_j \in \Gamma$. Each multiset of objects $u_j$ and string $s_j$, $1 \leq j \leq p$, are replaced by a multiset of objects $u'_j$ and strings $s'_{j,1}, \ldots, s_{j,i_j}$.

A constant $c_j^{l_t}$ is associated with each rule and will be referred to as *stochastic constant* and is needed to calculate the propensity of the rule according to the current context of the membrane to which this rule corresponds.

Rules in stochastic P systems model biochemical reactions. The *propensity* $a_j$ of a reaction $R_j$ is defined so that $a_j dt$ represents the probability that $R_j$ will occur in the infinitesimal time interval $[t, t + dt]$ [7].

Applications of the rules and the semantics of stochastic P systems can vary, depending on which algorithm is used to simulate the model. At the present stage, two algorithms have been implemented and integrated as simulators within the pLinguaCore library. They will be discussed in Section 3.2.

## 3 Simulators

### 3.1 Contemplating new simulators

In [3], only one simulator was implemented, since there was only one model to simulate. However, as new models have been included, new simulators have been developed inside the pLinguaCore library, providing at least one simulator for each supported model.

All the current simulators can step backwards, but this option should be set before the simulation starts.

The library also takes into account the existence of different simulation algorithms for the same model and provides means for selecting a simulator among the ones which are suitable to simulate the P system, by checking its model.

Next, simulation algorithms for Stochastic and Probabilistic P systems are explained, but pLinguaCore integrates simulators for all supported models.

### 3.2 Simulators for Stochastic P Systems

In the original approach to membrane computing P systems evolve in a non-deterministic and maximally parallel manner (that is, all the objects in every membrane that can evolve by a rule must do it [14]). When trying to simulate biological phenomena, like living cells, the classical non-deterministic and maximally parallel approach is not valid anymore. First, biochemical reactions, which are modeled by rules, occur at a specific rate (determined by the propensity of the rule), therefore they can not be selected in an arbitrary and non-deterministic way. Second, in the classical approach all time step are equal and this does not represent the time evolution of a real cell system.

The strategies to replace the original approach are based on Gillespie's Theory of Stochastic Kinetics [7]. As mentioned in Section 2.6, a constant $c_j^{l_t}$ is associated

to each rule. This provides P systems with a stochastic extension. The constant $c_j^{l_t}$ depends on the physical properties of the molecules involved in the reaction modeled by the rule and other physical parameters of the system and it represents the probability per time unit that the reaction takes place. Also, it is used to calculate the propensity of each rule which determines the probability and time needed to apply the rule.

Two different algorithms based on the principles stated above have been currently implemented and integrated in pLinguaCore.

### Multicompartimental Gillespie Algorithm

The Gillespie [7] algorithm or SSA (Stochastic Simulation Algorithm) was developed for a single, well-mixed and fixed volume/compartment. P systems generally contain several compartments or membranes. For that reason, an adaptation of this algorithm was presented in [20] and it can be applied in the different regions defined by the compartmentalised structure of a P system model. The next rule to be applied in each compartment and the waiting time for this application is computed using a *local* Gillespie algorithm. The Multicompartimental Gillespie Algorithm can be broadly summarized as follows:

Repeat until a prefixed simulation time is reached:

1. Calculate for each membrane $i, 1 \leq i \leq m$ and for each rule $r_j \in R_{l_i}$ the propensity, $a_j$, by multiplying the stochastic constant $c_j^{l_i}$ associated to $r_j$ by the number of distinct possible combinations of the objects and substrings present of the left-side of the rule with respect to the current contents of membranes involved in the rule.
2. Compute the sum of all propensities

$$a_0 = \sum_{i=1}^{m} \sum_{r_j \in R_{l_i}} a_j$$

3. Generate two random numbers $r_1$ and $r_2$ from the uniform distribution in the unit interval and select $\tau_i$ and $j_i$ according to

$$\tau_i = \frac{1}{a_0} \ln(\frac{1}{r_1})$$

$$j_i = \text{the smallest integer satisfying} \sum_{j=1}^{j_i} a_j > r_2 a_0$$

   In this way, we choose $\tau_i$ according to an exponential distribution with parameter $a_0$.
4. The next rule to be applied is $r_{j_i}$ and the waiting time for this rule is $\tau_i$. As a result of the application of this rule, the state of one or two compartments may be changed and has to be updated.

### Multicompartimental Next Reaction Method

The Gillespie Algorithm is an exact numerical simulation method appropiate for systems with a small number of reactions, since it takes time proportional to the number of reactions (i.e., the number of rules). An exact algorithm which is also efficient is presented in [6], the Next Reaction Method. It uses only a single random number per simulation event (instead of two) and takes time proportional to the logarithm of the number of reactions. We have adapted this algorithm to make it compartimental.

The idea of this method is to be extremely sensitive in recalculating $a_j$ and $t_i$, recalculate them only if they change. In order to do that, a data structure called *dependency graph* [6] is introduced.

Let $r : u[v]_l \xrightarrow{c} u'[v']_l$ be a given rule with propensity $a_r$ and let the parent membrane of $l$ be labelled with $l'$. We define the following sets:

- $\text{DependsOn}(a_r) = \{(b,t) : b$ is an object or string whose quantity affect the value
  $a_r$ and $t = l$ if $b \in v$ and $t = l'$ if $b \in u\}$
  Generally, $\text{DependsOn}(a_r) = \{(b,l) : b \in v\} \cup \{(b,l') : b \in u\}$
- $\text{Affects}(r) = \{(b,t) : b$ is an object or string whose quantity is changed when the rule
  $r$ is excuted and $t = l$ if $b \in v \vee b \in v'$ and $t = l'$ if $b \in u \vee b \in u'\}$
  Generally, $\text{Affects}(r) = \{(b,l) : b \in v \vee b \in v'\} \cup \{(b,l') : b \in u \vee b \in u'\}$

**Definition 1.** *Given a set of rules $R = R_{l_1} \cup \cdots \cup R_{l_m}$, the dependency graph is a directed graph $G = (V, E)$, with vertex set $V = R$ and edge set $E = \{(v_i, v_j) : \mathit{Affects}(v_i) \cap \mathit{DependsOn}(a_{v_j}) \neq \emptyset\}$*

In this way, if there exists an edge $(v_i, v_j) \in E$ and $v_i$ is executed, as some objects affected by this execution are involved in the calculation of $a_{v_j}$, this propensity would have to be recalculated. The dependency graph depends only on the rules of the system and is static, so it is built only once.

The times $\tau_i$, that represent the waiting time for each rule to be applied, are stored in an *indexed priority queue*. This data structure, discussed in detail in [6], has nice properties: finding the minimum element takes constant time, the number of nodes is the number of rules $|R|$, because of the indexing scheme it is possible to find any arbitrary reaction in constant time and finally, the operation of updating a node (only when $\tau_i$ is changed, which we can detect using to the dependency graph) takes $\log |R|$ operations.

The Multicompartimental Next Reaction Method can be broadly summarized as follows:

1. Build the dependency graph, calculate the propensity $a_r$ for every rule $r \in R$ and generate $\tau_i$ for every rule according to an exponential distribution with parameter $a_r$. All the values $\tau_r$ are stored in a priority queue. Set $t \leftarrow 0$ (this is the global time of the system).

2. Get the minimum $\tau_\mu$ from the priority queue, $t \leftarrow t + \tau_\mu$. Execute the rule $r_\mu$ (this is the next rule scheduled to be executed, because its waiting time is least).
3. For each edge $(\mu, \alpha)$ in the dependency graph recalculate and update the propensity $a_\alpha$ and
   - if $\alpha \neq \mu$, set
   $$\tau_\alpha \leftarrow \frac{a_{\alpha,old}(\tau_\alpha - \tau_\mu)}{a_{\alpha,new}} + \tau_\mu$$
   - if $\alpha = \mu$, generate a random number $r_1$, according to an exponential distribution with parameter $a_\mu$ and set $\tau_\mu \leftarrow \tau_\mu + r_1$

   Update the node in the indexed priority queue that holds $\tau_\alpha$.
4. Go to 2 and repeat until a prefixed simulation time is reached.

Both Multicompartimental Gillespie Algorithm and Multicompartimental Next Reaction Method are the core of the Direct Stochastic Simulator and Efficient Stochastic Simulator, respectively. One of them, which can be chosen in runtime, will be executed when compiling and simulating a P-Lingua file that starts with `@model<stochastic>`. See Section 4.1 for more details about the syntax.

### 3.3 Simulators for Probabilistic P Systems

Two different simulation algorithms have been created in this paper and integrates within the pLinguaCore library for the Probabilistic P system model. The first one is called Uniform Random Distribution Algorithm. The second one gives a better efficiency by using the binomial distribution, and it is called Binomial Random Distribution Algorithm.

#### Uniform Random Distribution Algorithm

Next, we describe how this algorithm determines the applicability of the rules to a given configuration.

(a) Rules are classified into sets so that all the rules belonging to the same set have the same left–hand side.
(b) Let $\{r_1, \ldots, r_t\}$ be one of the said sets of rules. Let us suppose that the common left-hand side is $u\ [v]_i^\alpha$ and their respective probabilistic constants are $c_{r_1}, \ldots, c_{r_t}$. In order to determine how these rules are applied to a give configuration, we proceed as follows:
   - It is computed the greatest number N so that $u^N$ appears in the father membrane of $i$ and $v^N$ appears in membrane $i$.
   - N random numbers $x$ such that $0 \leq x < 1$ are generated.
   - For each $k$ $(1 \leq k \leq t)$ let $n_k$ be the amount of numbers generated belonging to interval $[\ \sum_{j=0}^{k-1} c_{r_j}\ ,\ \sum_{j=0}^{k} c_{r_j})$ (assuming that $c_{r_0} = 0$).
   - For each $k$ $(1 \leq k \leq t)$, rule $r_k$ is applied $n_k$ times.

**Binomial Random Distribution Algorithm**

Next, we describe how this algorithm determines the applicability of the rules to a given configuration.

(a) Rules are classified into sets so that all the rules belonging to the same set have the same left–hand side.
(b) Let $\{r_1, \ldots, r_t\}$ be one of the said sets of rules. Let us suppose that the common left-hand side is $u\ [v]_i^\alpha$ and their respective probabilistic constants are $c_{r_1}, \ldots, c_{r_t}$. In order to determine how these rules are applied to a give configuration, we proceed as follows:
(c)) Let $F(N, p)$ a function that returns a discrete random number within the binomial distribution $B(N, p)$
   – It is computed the greatest number N so that $u^N$ appears in the father membrane of $i$ and $v^N$ appears in membrane $i$.
   – let $d = 1$
   – For each $k$ $(1 \leq k \leq t-1)$ do
      · let $c_{r_k}$ be $\frac{c_{r_k}}{d}$
      · let $n_k$ be $F(N, c_{r_k})$
      · let $N$ be $N - n_k$
      · let $q$ be $1 - c_{r_k}$
      · let $d$ be $d * q$
   – let $n_t$ be $N$
   – For each $k$ $(1 \leq k \leq t)$, rule $r_k$ is applied $n_k$ times.

## 4 Formats

As well as models and simulators, new file formats to define P systems have been included in P-Lingua 2.0. Although XML format and P–Lingua format were included on the first version of the software [3], those formats have been upgraded to allow representation of P systems which have cell-like structure. As P–Lingua 2.0 provides backwards compatibility, all valid actions in the first version are still valid. Furthermore, a new format has been included: the binary format (suitable for the forthcoming Nvidia CUDA simulator [11]).

Formats are classified in two sorts: **Input formats** (whose files can be read by pLinguaCore) and **Output formats** (whose files can be generated by pLinguaCore). Some formats may belong to both categories.

One format which is worth showing up is the P–Lingua format. This input format allows to specify P systems in a very intuitive, friendly and straightforward way. Another asset to bear in mind is that the parser for P–Lingua inside the pLinguaCore library is capable of locating errors on files specified on this format.

### 4.1 P-Lingua format

In the version of P-Lingua presented in [3] only P systems with active membranes and division rules were considered and therefore, possible to be defined in the P-Lingua language. New models have been added and consequently the syntax has been modified and extended, in order to support them. The current syntax of the P-Lingua language is defined as follows.

**Valid identifiers**

We say that a sequence of characters forms a `valid identifier` if it does not begin with a numeric character and it is composed by characters from the following:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9 _
```

Valid identifiers are widely used in the language: to define module names, parameters, indexes, membrane labels, alphabet objects and strings.

The following text strings are reserved words in the language: `def, call, @mu, @ms, @model, @lambda, @d, let, @inf, @debug, main, -->, #` and they cannot be used as valid identifiers.

**Variables**

Four kind of variables are permitted in P-Lingua: `Global variables`, `Local variables`, `indexes`, `Parameters`.

Variables are used to store numeric values and their names are valid identifiers. We use 64 bits (signed) in double precision.

**Global variables definition**

Global variables must be declared out of any program module and they can be accessed from all of the program modules (see 4.1). The name of a global variable `global_variable_name` must be a valid identifier. The syntax to define a global variable is the following:

```
global_variable_name = numeric_expression;
```

**Local variables definition**

Local variables can only be accessed from the module in which they were declared and they must only be defined inside module definitions. The name of a local variable `local_variable_name` must be a valid identifier. The syntax to define a local variable is the following:

```
let local_variable_name = numeric_expression;
```

Indexes and parameters can be consider local variables used in 4.1 and 4.1 respectively. **Identifiers for electrical charges**

In P-Lingua, we can consider electrical charges by using the `+` and `-` symbols for positive and negative charges respectively, and no one for neutral charge. It is worth mentioning that polarizationless P systems are included.

**Membrane labels**

There are three ways of writing membrane labels in P-Lingua: the first one is just a natural number; the second one is to denote the label as a valid identifier and the third one is by numeric expressions that represent natural numbers between brackets.

**Numeric expressions**

Numeric expressions can be written by using `*` (multiplication), `/` (division), `%` (module), `+` (addition), `-` (subtraction) and `^` (potence) operators with integer or real numbers and/or variables, along with the use of parentheses. It is possible to write numbers by using exponential notation. For example, $3 * 10^{-5}$ is written `3e-5`.

**Objects** The objects of the alphabet of a P system are written using valid identifiers, and the inclusion of sub-indexes is permitted. For example, $x_{i,2n+1}$ and $Yes$ are written as `x{i,2*n+1}` and `Yes` respectively.

The multiplicity of an object is represented by using the `*` operator. For example, $x_i^{2n+1}$ is written as `x{i}*(2*n+1)`.

**Strings** Strings are enclosed between `<` and `>` and made by concatenating valid identifiers with the character `.`, that is `<identifier1. ... .identifierN>`. For example, `<cap.RNAP.op>`.

**Substrings** Substrings are used in string rewriting rules and the syntax is similar to strings, but it is possible to use the character `?` to represent any arbitrary sequence of valid identifiers concatenated by `.`. The empty sequence is included. For example, `<cap.?.NAP.op>` is a substring of the string `<cap.op.op.op.NAP.op>` and of the string `<cap.NAP.op>`.

**Model specification** As this programming language supports more than one model, it is necessary to specify in the beginning of the file which is the model of the P system defined. Not each type of rule is allowed in every model, for example, membrane creation rules are not permitted in P systems with symport/antiport rules. The built-in compiler of P-Lingua detects such error. Models are specified by using `@model<model_name>` and at this stage, the allowed models are:

```
@model<membrane_division>
@model<membrane_creation>
@model<transition_psystem>
@model<probabilistic_psystem>
@model<stochastic_psystem>
@model<symport_antiport_psystem>
```

### Modules definition

Similarities between various solutions to **NP**-complete numerical problems by using families of recognizing P systems are discussed in [8]. Also, a cellular programming language is proposed based on libraries of subroutines. Using these ideas, a P-Lingua program consists of a set of programming modules that can be used more times by the same, or other, programs.

The syntax to define a module is the following.

```
def module_name(param1,..., paramN)
{
  sentence0;
  sentence1;
  ...
  sentenceM;
}
```

The name of a module, `module_name`, must be a valid and unique identifier. The parameters must be valid identifiers and cannot appear repeated. It is possible to define a module without parameters. Parameters have a numerical value that is assigned at the module call (see below).

All programs written in P-Lingua must contain a `main` module without parameters. The compiler will look for it when generating the output file.

In P-Lingua there are sentences to define the membrane structure of a P system, to specify multisets, to define rules, to define variables and to call to other modules. Next, let us see how such sentences are written.

### Module calls

In P-Lingua, modules are executed by using calls. The format of an sentence that calls a module for some specific values of its parameters is given next:

```
call module_name(value1, ..., valueN);
```

where `value`$i$ is a numeric expression or a variable.

### Definition of the initial membrane structure of a P system

In order to define the initial membrane structure of a P system, the following sentence must be written:

```
@mu = expr;
```

where `expr` is a sequence of matching square brackets representing the membrane structure, including some identifiers that specify the label and the electrical charge of each membrane.

Examples:

1. $[[\,]^0_2]^0_1 \equiv$ `@mu = [[]'2]'1`
2. $[[\,]^0_b[\,]^-_c]^+_a \equiv$ `@mu = +[[]'b, -[]'c]'a`

**Definition of multisets**

The next sentence defines the initial multiset associated to the membrane labelled by `label`.

```
@ms(label) = list_of_objects;
```

where `label` is a membrane label and `list_of_objects` is a comma-separated list of objects. The character `#` is used to represent an empty multiset.

If a stochastic P system is being defined (that is, the file starts with `@model<stochastic>`), strings are also permitted in the initial content of a membrane:

```
@ms(label) = list_of_objects_and_strings;
```

`list_of_objects_and_strings` is a comma-separated list of objects and/or strings.

**Union of multisets**

P-Lingua allows to define the union of two multisets (recall that the input multiset is "added" to the initial multiset of the input membrane) by using a sentence with the following format.

```
@ms(label) += list_of_objects;
```

For stochastic P systems, it would be

```
@ms(label) += list_of_objects_and_strings;
```

**Definition of rules**

The definition of rules has been significantly extended in this version of P-Lingua. A general rule is defined as follow (most elements are optional):

$$u[v[w_1]_{h_1}^{\alpha_1}\dots[w_n]_{h_n}^{\alpha_n}]_h^{\alpha} \xrightarrow{k} x[y[z_1]_{h_1}^{\beta_1}\dots[z_n]_{h_n}^{\beta_n}]_h^{\beta}[s]_h^{\gamma}$$

where $u, v, w_1, \dots, w_n, x, y, z_1, \dots, z_n$ are multisets of objects or strings, $h, h_1, \dots, h_n$ are labels, $\alpha, \alpha_1, \dots, \alpha_n, \beta, \beta_1, \dots, \beta_n, \gamma$ are electrical charges and $k$ is a numerical value.

The P-Lingua sintax for such a rule is:

`u`$\alpha$`[v`$\alpha_1$`[w1]'h1`$\dots$$\alpha_n$`[wN]'hN]'h --> x`$\beta$`[y`$\beta_1$`[z1]'h1`$\dots$$\beta_n$`[zN]'hN]'h `$\gamma$`[s]'h :: k`

where `u, v, w1`$\dots$`wN, x, y, z1`$\dots$`zN, s` are comma-separated list of objects or strings (it is possible to use the character `#` in order to represent the empty multiset), `h,h1,..., hN` are labels, $\alpha, \alpha_1, \dots, \alpha_n, \beta, \beta_1, \dots, \beta_n, \gamma$ are identifiers for electrical charges and `k` is a numeric expression.

As mentioned before, not each type of rule is permitted in every model. Below we enumerate the possible types of rules, classified by the model in which they are allowed.

`@model<mebrane_division>`

1. The format to define evolution rules of type $[a \rightarrow v]_h^\alpha$ is given next:

   $\alpha$`[a --> v]'h`

2. The format to define send-in communication rules of type $a\,[\,]_h^\alpha \rightarrow [b]_h^\beta$ is given next:

   `a`$\alpha$`[]'h -->`$\beta$`[b]`

3. The format to define send-out communication rules of type $[a]_h^\alpha \rightarrow b[\,]_h^\beta$ is given next:

   $\alpha$`[a]'h -->` $\beta$`[]b`

4. The format to define division rules of type $[a\,]_h^\alpha \rightarrow [b]_h^\beta[c]_h^\gamma$ is given next:

   $\alpha$`[a]'h -->`$\beta$`[b]`$\gamma$`[c]`

5. The format to define dissolution rules of type $[a]_h^\alpha \rightarrow b$ is given next:

   $\alpha$`[a]'h --> b`

`@model<membrane_creation>`

1. Rules 1, 2, 3 and 5 of `@model<membrane_division>` can be defined in this model, with the same format.
2. The format to define membrane creation rules of type $[a]_h^\alpha \rightarrow [[b]_{h_1}^\beta]_h^\alpha$ is given next:

   $\alpha$`[a]'h -->` $\alpha$`[`$\beta$`[b]'h1]'h`

`@model<transition_psystem>`

1. The format to define evolution rules of type $[u[u_1]_{h_1}, \ldots, [u_N]_{h_N} \rightarrow v[v_1]_{h_1}, \ldots, [v_N]_{h_N}, \lambda]_h$ is given next:

   `[u [u1]'h1 ... [uN]'hN --> v [v1]'h1, ... [vN]'hN, @d]'h`

   `@d` is a new keyword representing the containing membrane is marked to dissolved.

`@model<symport_antiport_psystem>`

1. The format to define symmetric communication rules of type $a[b]_h^\alpha \rightarrow b[a]_h^\alpha$ is given next:

   $\alpha$`a[b]'h -->` $\beta$`b[a]'h`

`@model<probabilistic_psystem>`

1. The format to define rules of type $u[v]_h^\alpha \xrightarrow{p} u_1[v_1]_h^\beta$ is given next:

   `u`$\alpha$`[v]'h --> u1`$\beta$`[v1]'h::p`

`@model<stochastic_psystem>`

1. The format to define multiset rewriting rules of type $u[v]_h \xrightarrow{c} u_1[v_1]_h$ is given next:

   ```
   u[v]'h --> u1[v1]'h::c
   ```

2. The format to define string rewriting rules of type $[u+s]_h \xrightarrow{c} [v+r]_h$ is given next:

   ```
   [u,s]'h --> [v,r]'h::c
   ```

- $\alpha$, $\beta$ and $\gamma$ are identifiers for electrical charges.
- `a`, `b` and `c` are objects of the alphabet.
- `u, u1, v, v1, ..., vN` are comma-separated lists of objects that represents a multiset.
- `s` and `r` are comma-separated lists of substrings.
- `h`, `h1, ..., hN` are labels.
- `p` and `c` are real numeric expressions. The result of evaluating `p` must be between 0 and 1, and the result of evaluating `c` must be greater or equal than 0.

Some examples:

- $[x_{i,1} \rightarrow r^4_{i,1}]^+_2 \equiv$ `+[x{i,1} --> r{i,1}*4]'2`
- $d_k[\,]^0_2 \rightarrow [d_{k+1}]^0_2 \equiv$ `d{k}[]'2 --> [d{k+1}]`
- $[d_k]^+_2 \rightarrow []^0_2 d_k \equiv$ `+[d{k}]'2 --> []d{k}`
- $[d_k]^0_2 \rightarrow [d_k]^+_2 [d_k]^-_2 \equiv$ `[d{k}]'2 --> +[d{k}]-[d{k}]`
- $[a]^-_2 \rightarrow b \equiv$ `-[a]'2 --> b`
- $Y_{i,j}[]_2 \xrightarrow{k_{i,8}} [B^{k_i,12}]_2 \equiv$ `Y{i,j}[]'2 --> [B*k{i,12}]'2::k{i,8}`
- $[RNAP+ < cap.\omega.op >]_m \xrightarrow{c} [< cap.\omega.RNAP.op >]_m \equiv$ `[RNAP,<cap.?.op>]'m --> [<cap.?.RNAP.op>]'m::c`

### Parametric sentences

In P-Lingua, it is possible to define parametric sentences by using the following format:

```
sentence : range1, ..., rangeN, restriction1, ...,
restrictionN;
```

where `sentence` is a sentence of the language, or a sequence of sentences in brackets, and `range1, ..., rangeN` is a comma-separated list of ranges with the format:

```
min_value <= index <= max_value
```

where `min_value` and `max_value` are numeric expressions, integer numbers or variables, and `index` is a variable that can be used in the context of the sentence. It is possible to use the operator < instead of <=.

And `restriction1, ..., restrictionN` are optional restrictions for the indexes values which the next syntax:

```
value1 <> value2
```

where `value1` and `value2` are numeric expressions, integer numbers or variables.

The sentence will be repeated for each possible values of each `index`.

Some examples of parametric sentences:

1. $[d_k]_2^0 \rightarrow [d_k]_2^+[d_k]_2^- : 1 \leq k \leq n \equiv$
   `[d{k}]'2 --> +[d{k}]-[d{k}] : 1<= k <= n;`
2. $[x_{i,j} \rightarrow x_{i,j-1}]_2^+ : 1 \leq i \leq m, 2 \leq j \leq n, i \neq j \equiv$
   `+[x{i,j} --> x{i,j-1}]'2 : 1<=i<=m,2<=j<=n,i<>j;`

**Inclusion of comments** The programs in P-Lingua can be commented by writing phrases into the text strings `/*` and `*/`. **Inclusion of debug information** Each rule sentence can optionally include a debug message which will be presented every time the rule is executed by the simulator. The syntax to write a debug message associated to a rule definition is defined as follows:

```
rule_definition @debug ‘‘debug message"
```

## 5 Command-line tools

P-Lingua 1.0 provided command-line tools for simulating P systems and compiling files which specify P systems [3]. In P-Lingua 2.0, the command-line tool general syntax has changed but, as it provides backwards compatibility, all valid actions in P-Lingua 1.0 are still valid in P-Lingua 2.0, as well.

### 5.1 Compilation command-line tool

The command-line tool general syntax for compiling input files is defined as follows:

```
plingua [-input_format] input_file [-output_format]
output_file [-v verbosity_level] [-h]
```

The command header `plingua` reports the system to compile the P system specified on a file to a file specified on another, whereas the file `input_file` contains the program that we want to be compiled, and `output_file` is the name of the file that is generated [3]. Optional arguments are in square brackets:

- The option `-input_format` defines the format followed by `input_file`, which should be an input format.

- At this stage, valid input formats are:
  - `P-Lingua`
  - `XML`
- If no input format is set, the P-Lingua format is assumed.
- The option `-output_format` defines the format followed by `output_file`, which should be an output format.
- At this stage, valid output formats are:
  - `XML`
  - `bin`
- If no input format is set, the XML format is assumed by default.
- The option `-v` verbosity level is a number between 0 and 5 indicating the level of detail of the messages shown during the compilation process [3].
- The option `-h` displays some help information [3].

### 5.2 Simulation command-line tool

The simulations are launched from the command line as follows:

```
plingua_sim [-input_format] input_file -o output_file [-v
verbosity level] [-h] [-to timeout] [-st steps] [-mode
simulatorID] [-a] [-b]
```

The command header `plingua_sim` reports the system to simulate the P system specified on a file, whereas `input_xml` is an XML document where a P system is formatted on, and output file is the name of the file where the report about the simulated computation will be saved [3]. Optional arguments are in brackets:

- The option `-input_format` defines the format followed by `input_file`, which should be an input format.
- The option `-v` verbosity level is a number between 0 and 5 indicating the level of detail of the messages shown during the compilation process [3]. If no value is specified, by default it is 3.
- The option `-h` displays some help information [3].
- The option `-to` sets a timeout for the simulation defined in timeout (in milliseconds), so when the time out has elapsed the simulation is halted. If the simulation has reached a halting configuration before the time out has elapsed this option has no effect.
- The option `-st` sets a maximum number of steps the simulation can take (defined in steps), so when the time out has elapsed the simulation comes to a halt. If the simulation has reached a halting configuration or the time out has elapsed (in case the option `-to` is set) before the specified number of steps have been taken this option has no effect.
- The option `-mode` sets the specific simulator to simulate the P system (defined in simulatorID). This option reports an error in case the simulator defined by simulatorID is not a valid simulator for the P system model.

- The option `-a` defines if the simulation can take alternative steps. This option reports an error if the simulator does not support alternative steps.
- The option `-b` defines if the simulation can step backwards. As every simulator supports stepping backwards, this option does not report errors.

## 6 pLinguaCore

pLinguaCore © is a JAVA library which performs all functions supported by P-Lingua 2.0, that is, models definition, simulators and formats. This library reports the rules and membrane structure read from a file where a P system is defined, detects errors in the file, reports them. And, if the P system is defined in P-Lingua language, locates the error on the file. This library performs simulations by using the simulators implemented as well as taking into account all options defined. It reports the simulation process, by displaying the current configuration as text and reporting the elapsed time. Eventually, this library translates files, which define a P system, between formats, for instance, from P-Lingua language format to binary format. For more information and library documentation, please browse the P-Lingua website [24]. This library is free software published under GNU GPL license [21], so everyone who is interested can change and distribute this library respecting the license conditions.

## 7 A tool for simulating ecosystems based on P-Lingua

The Bearded Vulture (Gypaetus barbatus) is an endangered species in Europe that feeds almost exclusively on bone remains of wild and domestic ungulates. In [1], it is presented a first model of an ecosystem related to the Bearded Vulture in the Pyrenees (NE Spain), by using probabilistic P systems where the inherent stochasticity and uncertainty in ecosystems are captured by using probabilistic strategies. In order to validate experimentally the designed P system (see figure 1) the authors have developed a simulator that allows them to analyze the evolution of the ecosystem under different initial conditions. That software application is focused on a particular P system, specifically, the initial model of the ecosystem presented in [1]. With the aim of improving the model, the authors are adding ingredients to it, such as new species and a more complex behaviour for the animals. In this sense, a second version of the model is presented in [2].

A new GPL [21] licensed JAVA application with a friendly user-interface sitting on the pLinguaCore library has been developed. This application provides a flexible way to check, validate and improve computational models of ecosystem based on P systems instead of designing new software tools each time new ingredients are added to the models. Furthermore, it is possible to change the initial parameters of the modelled ecosystem in order to make the virtual experiments suggested by experts (see figure 2). These experiments will provide results that

can be interpreted in terms of hypotheses. Finally, some of these hypotheses will be selected by the experts in order to be checked in real experiments.

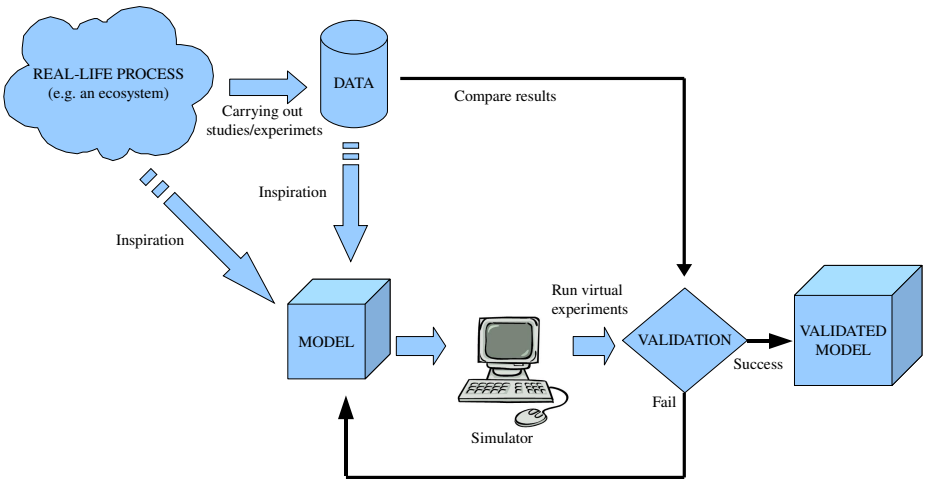


**Fig. 1. Validation proccess**

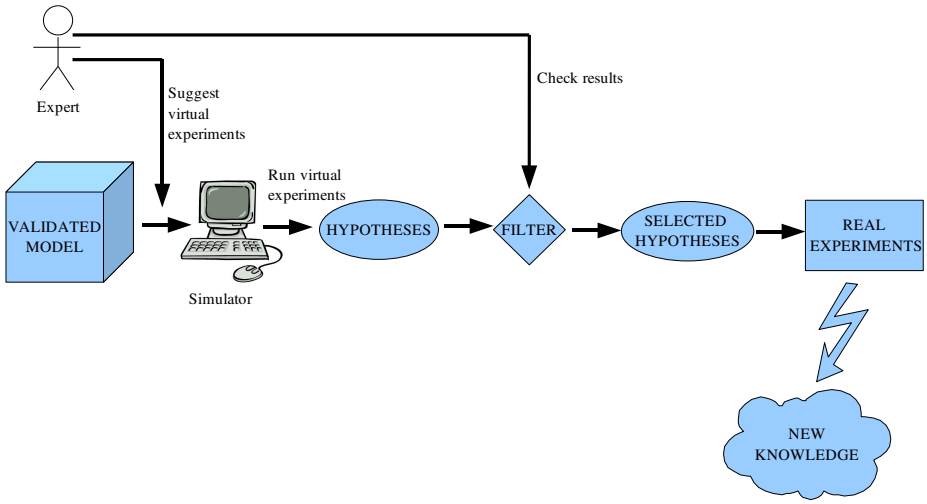


**Fig. 2. Virtual experimentation**

## 8 Conclusions and future work

Creating a programming language to specify P systems is an important task in order to facilitate the development of software applications for membrane computing.

In [3], P-Lingua was presented as a programming language to define active membrane P systems with division rules. The present paper extends that language to other models: transition P systems, symport/antiport P systems, active membrane P systems with division or creation rules, probabilistic P systems and stochastic P systems.

We have developed a JAVA library (pLinguaCore) that implements several simulators for each mentioned model and defines different formats to encode P systems, like the P-Lingua one or a new binary format. This library can be expanded to define new models, simulators and formats.

It is possible to select different algorithms to simulate a P system, for example, there are two different algorithms for stochastic P systems. The library can be used inside other software applications, in this sense, we present a tool for virtual experimentation of ecosystems.

An internet website [24] is available to download the applications, libraries and source-code, as well as provide information about the P-Lingua project. In addition, this site aims to be a meeting point for users and developers through the use of web-tools such as forums.

The syntax of P-Lingua language is standard enough for specifying several different models of cell–like P systems. However, a new version is necessary in order to specify tissue P systems and this will be aim of a future work.

Although P-Lingua 2.0 provides a way to simulate and compile P systems, command-line tools are usually not user-friendly. It means it is not easy and intuitive to use them. For this purpose, a new user interface called pLinguaPlugin has been developed. This one is integrated into the Eclipse platform [23], so it makes the most of Eclipse's capabilities to provide a framework for translating, developing and testing P systems. It aims to be user-friendly and useful for P system researchers.

### Acknowledgement

The authors acknowledge the support of the project TIN2006–13425 of the Ministerio de Educación y Ciencia of Spain, cofinanced by FEDER funds, and the support of the "Proyecto de Excelencia con Investigador de Reconocida Valía" of the Junta de Andalucía under grant TIC04200.

## References

1. M. Cardona, M.A. Colomer, M.J. Pérez–Jiménez, D. Sanuy and A. Margalida. Modeling Ecosystems Using P Systems: The Bearded Vulture, a Case Study. LNCS 5391, 137–156, 2009

2. M. Cardona, M.A. Colomer, A. Margalida, I. Pérez–Hurtado, M.J. Pérez–Jiménez, D. Sanuy. P System based model of an ecosystem of the Scavenger Birds. *In this volume.*
3. D. Díaz–Pernil, I. Pérez–Hurtado, M.J. Pérez–Jiménez, A. Riscos–Núñez. A P-lingua programming environment for Membrane Computing, *Proceedings of the 9th Workshop on Membrane Computing*, 155–172, 2008.
4. F. Fontana, L. Bianco and V. Manca. P Systems and the Modelling fo Biochemical Oscillations, Membrane Computing, Sixth international Workshop, WMC6, Vienna, Austria, LNCS 3850, 199–208, 2005.
5. R. Freund, S. Verlan. A Formal Framework for Static (Tissue) P Systems, LNCS 4860, 271–284, 2007.
6. M.A. Gibson and J. Bruck. Efficient Exact Stochastic Simulation of Chemical Systems with Many Species and Many Channels, *J. Phys. Chem.*, 104, 1876–1889, 2000.
7. D.T. Gillespie. Exact stochastic simulation of coupled chemical reactions, *J. Phys. Chem.*, 81, 2340–2361, 1977.
8. M.A. Gutiérrez–Naranjo, M.J. Pérez–Jiménez, A. Riscos–Núñez. Towards a programming language in cellular computing. LNCS 123, 93–110 2005.
9. M. Ito, C. Martín–Vide, Gh. Păun. A characterization of Parikh sets of ET0L languages in terms of P systems. In *Words, semigroups and transducers* (M- Ito, Gh. Păun, S. Yu, eds.), 239–254, Word Scientific, Singapore 2001.
10. M. Madhu, K. Krithivasan. P systems with membrane creation: Universality and efficiency. LNCS 2055, 276–287, 2001.
11. M.A. Martínez–del–Amor, I. Pérez–Hurtado, M.J. Pérez–Jiménez, J.M. Cecilia, G.D. Guerrero, J.M. García. Simulation of Recognizer P Systems by using Manycore GPUs. *In this volume.*
12. A. Obtulowicz. Probabilistic P systems. *Lecture Notes in Computer Science*, 2597, 377–387, 2002.
13. A. Păun, Gh. Păun. The power of communication: P systems with symport/antiport. *New Generation Computing*, 20, 3, 295–305, 2002.
14. Gh. Păun. Computing with Membranes, *Journal of Computer and System Sciences* 61(1) 108–143, 2000.
15. Gh. Păun. P systems with active membranes. *Journal of Automata, Languages and Combinatorics*, 6, 1, 75–90, 2001.
16. M.J. Pérez–Jiménez, F.J. Romero–Campero. Modelling Gene Expression Control using P systems: The Lac Operon, a case study. *BioSystems*, 91, 438–457, 2008.
17. M.J. Pérez–Jiménez, F.J. Romero–Campero. A model of the Quorum Sensing System in Vibrio Fischeri using P systems. *Artificial Life*, 14, 95–109, 2008.
18. M.J. Pérez–Jiménez, F.J. Romero–Campero. P Systems, a new computational modelling tool for systems biology. *Transactions on Computational Systems Biology VI*, LNBI, 4220, 176–197, 2006.
19. D. Pescini, D. Besozzi, G. Mauri and C. Zandron. Dynamical probabilistic P systems. *International Journal of Foundations of Computer Science*, 17(1), 183–195, 2006.
20. F.J. Romero–Campero. P Systems, a Computational Modelling Framework for Systems Biology, Doctoral Thesis, University of Seville, Department of Computer Science and Artificial Intelligence, 2008.
21. The GNU General Public License: `http://www.gnu.org/copyleft/gpl.html`
22. Java web page: `http://www.java.com/`
23. The Eclipse Project: `http://www.eclipse.org`
24. The P-Lingua website: `http://www.p-lingua.org`