Information Sciences 587 (2022) 1–22

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

A new P-Lingua toolkit for agile development in membrane computing

Ignacio Pérez-Hurtado [a,*], David Orellana-Martín [a], Miguel A. Martínez-del-Amor [a,b], Luis Valencia-Cabrera [a,b], Agustín Riscos-Núñez [a,b]

[a] Research Group on Natural Computing, Department of Computer Science and Artificial Intelligence, Universidad de Sevilla, Seville 41012, Spain
[b] SCORE lab, I3US, Universidad de Sevilla, Seville 41012, Spain

ARTICLE INFO

Article history:
Received 29 June 2021
Received in revised form 24 September 2021
Accepted 2 December 2021
Available online 16 December 2021

Keywords:
Membrane computing
Computer languages
Computer simulation
Software tools
P-Lingua

ABSTRACT

Membrane computing is a massively parallel and non-deterministic bioinspired computing paradigm whose models are called P systems. Validating and testing such models is a challenge which is being overcome by developing simulators. Regardless of their heterogeneity, such simulators require to read and interpret the models to be simulated. To this end, P-Lingua is a high-level P system definition language which has been widely used in the last decade. The P-Lingua ecosystem includes not only the language, but also libraries and software tools for parsing and simulating membrane computing models. Each version of P-Lingua supported new types or *variants* of P systems. This leads to a shortcoming: Only a predefined list of variants can be used, thus making it difficult for researchers to study custom ones. Moreover, derivation modes cannot be user-defined, i.e, the way in which P system computations should be generated is determined by the simulation algorithm in the source code.

The main contribution of this paper is a completely new design of the P-Lingua language, called P-Lingua 5, in which the user can define custom variants and derivation modes, among other improvements such as including procedural programming and simulation directives. It is worth mentioning that it has backward-compatibility with previous versions of the language. A completely new set of command-line tools is provided for parsing and simulating P-Lingua 5 files. Finally, several examples are included in this paper covering the most common P system types.

1. Introduction

Membrane computing [1–5] is an unconventional model of computation inspired by the structure and functions of living cells. The computational devices in membrane computing are called *membrane systems* or *P systems*. They consist of a structure of compartments containing objects that evolve along a computation according to a set of rewriting rules. The compartments, a.k.a. *membranes*, can also evolve by the application of special rules that are able to divide, separate or create membranes. In Fig. 1, a simple cell-like membrane system with its characteristic hierarchical structure is depicted. There are many types of P systems, called P system variants or just *variants*.

* Corresponding author.
E-mail addresses: perezh@us.es (I. Pérez-Hurtado), dorellana@us.es (D. Orellana-Martín), mdelamor@us.es (M.A. Martínez-del-Amor), lvalencia@us.es (L. Valencia-Cabrera), ariscosn@us.es (A. Riscos-Núñez).

https://doi.org/10.1016/j.ins.2021.12.003
0020-0255/

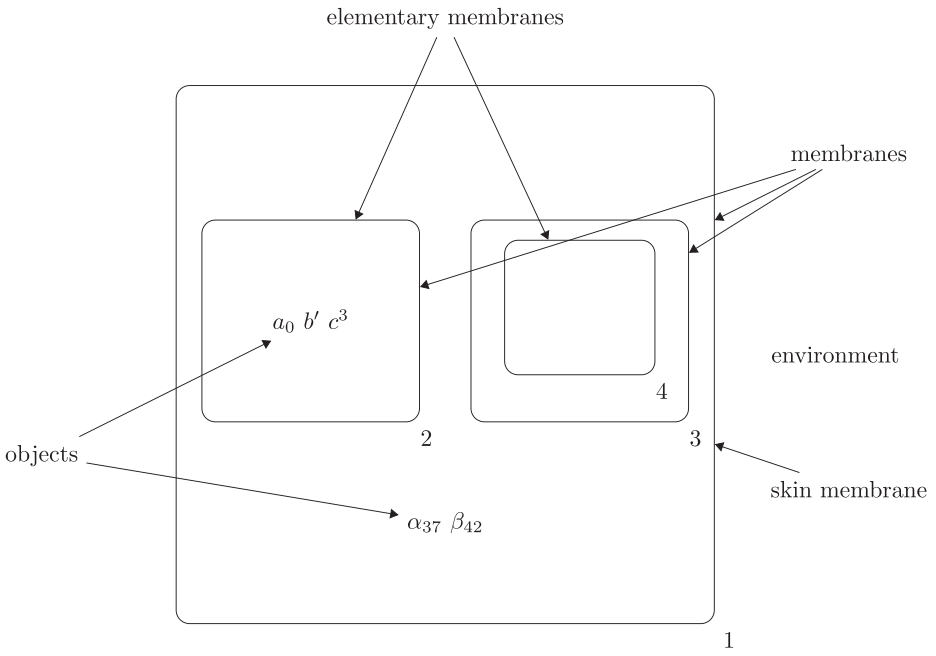

Fig. 1. A simple cell-like membrane system.

From the beginning, membrane computing has been applied to a large variety of research areas [6]: The study of the **P** vs **NP** problem [5,7–9]; the modelling of biological systems from the microscopic level [10,11] to the ecosystem level [12,13]; the modelling of certain types of economics systems [14]; the study of neural models [15,16] incorporating fuzzy reasoning [17]; the simulation of robot controllers [18–20]; the application to robot motion planning problems [21,22], and many more.

Nevertheless, it is important to remark that today there is no biological implementation of such devices, neither *in vivo* nor *in vitro*. Even though there are some promising solutions *in silico* by using parallel architectures [23], they must deal with problems such as genuine non-determinism, generation of computational resources on demand and other challenges at the frontiers of tractability. Despite researchers can always study P system models *on the whiteboard*, additional software/hardware tools to design, debug and apply such models have been proven to be useful in order to accelerate and improve the investigation on membrane computing. In this paper, the term *P system simulator* refers to a software or hardware tool capable of reproducing the behavior of one or more P systems. A simulator can be incorporated as a module or library in a larger software according to a particular application. On the other hand, a simulator can be used just to assist in the creation of new P system models.

The computation of P systems differs significantly from the way in which conventional computers (i.e. machines based on Von Neumann architecture) operate. First, P systems are non-deterministic machines in which the computation proceeds according to a multi-branch computational tree. Second, P systems are massively parallel devices in which rules can be executed in parallel for all compartments. Moreover, it is possible to create new compartments by applying special rules, thus generating new computational resources on demand dynamically. These features make it a real challenge to simulate certain types of P systems in an efficient manner [24].

Despite these challenges, today there is a wide variety of P system simulators [25] based on different architectures: From sequential simulators [26], in which only one thread of execution is applied, to parallel simulators based on multi-threading programming [22], FPGAs [23,27] or GPUs [28–31].

A common element required in all these simulators is the *input module*. It is the software or hardware module used to define the P systems to be simulated. There are several approaches: From *ad hoc simulators* [30,21] in which the definition is integrated in the source code, to definition languages [32] specifically designed to express P systems, which is the strategy followed by P-Lingua [32–37]. From more than a decade, P-Lingua language has been progressively extended to accept new variants of P systems. A GNU GPLv3 Java library called *pLinguaCore* [38] includes the necessary code to parse P-Lingua files, produce P system simulations and generate output files. A fork of the library was integrated in the project *MeCoSim* (Membrane Computing Simulator) [39,40] in order to offer the users a general purpose application to design, simulate, analyse and verify different types of models based on P systems (and to define custom user interfaces oriented to the end users). This library, pLinguaCore, can also transform P-Lingua files defining P systems to other formats such as XML and Binary after ana-

lysing the files and ensuring they are error-free. This is the approach used for the input module in PMCGPU [41], a project aimed to develop parallel simulators for P systems.

P-Lingua has been successfully applied to several fields [32]: From assisting the design process of new P system models [37,36] to the application to ecosystem modelling [13] and robot path planning [21]. In general terms, P-Lingua and its related tools and libraries have provided to the research community an agile development solution for software based on membrane computing.

Each P system variant has its own syntax and semantics. The syntax defines the set of valid expressions in the variant. On the other hand, the semantics or *derivation mode* describes how P system computations must be generated. The traditional approach in P-Lingua was to progressively extend the language for new variants. Each supported variant is related to *a unique identifier* which has to be included at the beginning of the P-Lingua file. Thus, after reading the identifier, the pLinguaCore library can apply a specific *parser* to recognize the corresponding syntax. This led to several shortcomings: It is not possible for the P-Lingua user to work directly with a custom P system variant, they must work with those types of P systems whose syntax has been specifically defined in pLinguaCore. Moreover, there are several versions and forks of the library, making it difficult to maintain them. On the other hand, the semantics for each variant is implemented in a P system simulator. The library pLinguaCore includes one or more simulators for each supported variant. It is also possible to use external simulators. In any case, the semantics is *hard-coded* in the simulator and it is not possible for the P-Lingua user to experiment with a custom derivation mode.

In this paper, we propose a completely new design of the P-Lingua framework in order to solve the shortcomings mentioned above. It includes a new version of the language, called P-Lingua 5, as well as a set of software tools called the P-Lingua 5 toolkit. In such a new version of the language, the syntax and semantics of the variants, instead of being hard-coded, can be defined together with the corresponding P system definitions. Therefore, it is not necessary to implement a custom parser for each P system variant. A stand-alone compiler for P-Lingua 5 developed in C++ under GNU GPLv3 license is included in the toolkit, somehow replacing the parsers in pLinguaCore. On the other hand, the user can define derivation modes in P-Lingua 5, being possible to use generic simulators that are able to simulate P systems according to custom derivation modes. Indeed, the toolkit includes a simulator implemented in C++ under GNU GPLv3 license which is able to produce P System computations for the derivation modes that can be defined in P-Lingua 5, replacing most of the simulators in pLinguaCore. This approximation has backward-compatibility with previous versions of the language. Thus, old P-Lingua files can be used with little or no modification. Examples for the most common variants are included in this paper. Other useful improvements have also been included in the language, such as procedural programming, simulation directives, header files, mathematical functions, string functions and user input/output functions.

The rest of this paper is structured as follows: Section 2 introduces the necessary preliminaries for the paper. Section 3 is focused on the new design of the framework, explaining how to write custom syntax and semantics for P systems, along with the rest of improvements. The next Section is dedicated to the software presented in this work, i.e., the P-Lingua 5 compiler and the P-Lingua 5 simulator. Section 5 explains how to connect the output of the compiler to external simulators. Section 6 analyses the advantages and disadvantages of our new design and compares it with other similar solutions in the literature. Finally, Section 7 enumerates the conclusions of this work and describes possible lines of future work.

2. Background

2.1. Membrane Computing

As mentioned in the previous Section, membrane computing is a bio-inspired unconventional computing paradigm, taking inspiration from the way cells interact with the environment (sending/sensing signals) and process information within them (biochemical reactions transforming and transporting molecules across inner vesicles/compartments). More precisely, a P system is a kind of automata whose behaviour is governed by rewriting rules that are applied over multisets of objects distributed along its compartments. A P system is defined basically by three components:

- A set of rewriting rules, that change the objects or their location within the membrane system (possibly also altering the structural elements).
- The structure of the system, defined (implicitly or explicitly) by a graph.
- The initial configuration, with both the initial contents of the compartments and the initial structure of the system.

A configuration of a P system is an instantaneous description of itself, defining both the multiset of objects present inside each compartment and the structure of the system at a certain moment of the computation. Let Π be a P system. By the application of a set of rules of Π at a certain moment t we can pass from configuration C_t to C_{t+1} in a transition or computation step, and we denote it by $C_t \Rightarrow_{\Pi} C_{t+1}$. If no rules can be applied to a certain configuration, we call it a *halting configuration*.

A computation $\mathscr{C}$ of length $n+1$ of a P system is a sequence $(C_0, C_1, \ldots, C_n)$ where $C_i \Rightarrow_{\Pi} C_{i+1}$, for $0 \leqslant i \leqslant n-1$. If C_n is a halting configuration, we call $\mathscr{C}$ a *halting computation*.

P systems are non-deterministic machines; that is, from a certain configuration, two different rules can be applicable to a certain object. In that moment, a decision is made and a certain rule will be applied to such object. It can be viewed as the

computation branches and that one rule is chosen in the first branch and the other rule is chosen in the second branch. Due to the non-deterministic nature of these systems, different "runs" can lead to different computations. We say that a P system Π has a *computation tree*, and each "run" goes through a *computation branch*.

The applicable rules are defined by the semantics of the system. In a certain configuration, different multisets of rules can be applied. Let **R** be the set of multisets of possible applicable rules to the P system Π. A derivation mode is a way to select certain multisets of rules from **R** according to some criteria. For instance, *maximally parallel mode* is a derivation mode that selects all the non-extensible multisets of rules from **R**. Finally, a non-deterministic choice is made from the set of these selected multisets of rules. A large study about derivation modes can be found in [42,43].

From the beginning of the discipline, several variants of P systems have been defined, differing in their structure, types of rules, derivation mode and other aspects.

2.2. Simulation in Membrane Computing

Along the present paper, it is widely used the term *P system simulator*, or just *simulator*, denoting a software or hardware tool capable of reproducing the behavior of one or more P systems. More precisely, according to the implementations in pLinguaCore [38], PMCGPU [41] and MeCoSim [40]: given a P system Π, a simulator is a tool which receives (1) the set of rules R defined in Π; (2) a starting configuration C_0 compatible with Π, not necessarily being the initial configuration of Π; and (3) a maximum number $n \in \mathbb{N}$ of computation steps to simulate. The simulator produces a finite sequence $(C_0, \ldots, C_m)$ with $0 \leqslant m \leqslant n$, following a computation branch of Π from C_0 to C_m. The sequence ends when a halting configuration is included or when $m = n$. Therefore, n is an upper bound in order to avoid an infinite simulation if the halting condition cannot be reached in a reasonable number of steps. For a generic implementation matching a wide range of P system variants, the halting condition can be triggered when no rules can be applied to the current configuration. The output of the simulation consists of the sequence $(C_0, \ldots, C_m)$ and the corresponding multisets of rules $(R_0, \ldots, R_{m-1})$ applied at each step of the computation. From a software development point of view, a minimal P system simulator can be obtained by implementing the following three functions:

1. `selectRules`(C_i, R): R_i
2. `executeRules`(C_i, R_i): C_{i+1}
3. `isHalting`(C_i): *Boolean*

The first function, `selectRules`, computes a multiset of applicable rules for a particular configuration given the set of rules of the P system. Such a function should follow a derivation mode according to the variant of the P system being simulated. It is important to remark that, due to the non-deterministic nature of P systems, different calls of the function with the same input can return different outputs. In that case, it is not required a particular probability distribution of outputs, but for debugging purposes, the uniform random distribution could be appropriate. Another possible implementation is to obtain always the same output for the same input. This approach can be implemented in a more efficient manner and it is enough to simulate certain types of P systems such as those with confluent computations (i.e. P systems returning the same results independently on the computation branch followed). This view would be useful when the focus is in the result, not in the full simulation output (sequence of transitions plus multisets of rules). The `executeRules` function applies a multiset of rules (generally, the one obtained by the previous function) to a given configuration, returning the next configuration. Such a function can be implemented in a generic manner, regardless of the P system variant. Finally, the function `isHalting` returns *True* if a given configuration is a halting configuration, and *False* otherwise. This function should be implemented depending on the variant being used.

In general terms, the expression *simulation algorithm* will refer to the algorithm which is implemented by a simulator to produce a P system computation. As an example, Algorithm 1 shows the general scheme of a sequential simulation algorithm implemented by the simulators in pLinguaCore [36]. It can be used for a wide range of P system variants just changing the code of the functions.

Algorithm 1: A sequential simulation algorithm

bf Require: Set of rules R; starting configuration C; maximum number n of computation steps to simulate.
  $i \leftarrow 0$
  $C_i \leftarrow C$
  while $i < n$ and not `isHalting`(C_i) **do**
    $R_i \leftarrow$ `selectRules`(C_i, R)
    $C_{i+1} \leftarrow$ `executeRules`(C_i, R_i)
    $i \leftarrow i + 1$
  end while
  return the sequences $(C_0, \ldots, C_i)$ and $(R_0, \ldots, R_i)$

According to the way in which the input of the simulator is read, we can classify them into three categories: *ad hoc simulators*, where the P system definition, i.e., its rules and/or its starting configuration, is totally or partially integrated together with the source code of the simulator; *simulators for a P system variant*, where the P system definition is given by external files, but the derivation mode is hard-coded in the simulator (which is the approximation of P-Lingua 4) and *generic simulators*, when the whole input (syntax and semantics of the P system to be simulated) is given by one or more external files. In this paper, we focus on the third category, being P-Lingua 5 a language to define such an input and being the P-Lingua 5 toolkit the software required to parse and simulate the corresponding P system definitions.

2.3. P-Lingua

The approximation of P-Lingua is to use a programming language to define P systems. The files written in P-Lingua are close to the standard scientific notation, minimizing the effort of writing them. P-Lingua code is parametric, using variables and iterators, modular, using reusable source blocks, and can be translated to other formats by using parsers. We could say that the main goal of the P-Lingua language is to minimize the time from the *whiteboard* to the *computer*.

P-Lingua aims at flexibility and extensibility to simulate P systems. The term itself is employed at different extents, so it is usually confused. P-Lingua can be used to denote:

- *the software project*: P-Lingua is an open-source software project published under license GNU GPL license. All the developed code is freely available to the community, and we hope that it helps other researchers to run their models and to develop new simulators. It includes the parser, the generation of files, and the simulation framework.
- *the simulation framework*: more commonly known as pLinguaCore, it is Java framework that aims at simulating P systems in a flexible and extensible way. Several design patterns of object oriented programming were employed to help developing new simulations in a clear and clean manner.
- *the programming language*: instead of having a specific GUI to enter the information of the models, P-Lingua enables P-system designers to write their models into plain-text files in a very similar way (i.e. with similar syntax) than they write them down in paper or whiteboard.
- *the files with.pli extension*: files including models described in P-Lingua programming language have the.pli extension, and they are the input for the simulation software framework.
- *the command-line tool*: pLinguaCore can be employed from the Terminal (Linux) or Cmd or equivalent (Windows), using a command-line tool with specific parameters to define the input.pli files, the output files (when used as a compiler) or the simulation options (when used as a simulator).
- *the translator/compiler to binary files*: efficient, parallel simulators came after P-Lingua, looking for small runtimes. They harness the P-Lingua parsing engine in order to process.pli files, and specific binary files are generated for them.

PLinguaCore [38] is a GNU GPL library written in Java for parsing P-Lingua files, simulating computations and translating input P-Lingua files to other file formats. The library detects errors in the file and reports them such as a regular compiler tool. For each supported P system variant, several simulation algorithms are included. Eventually, the library translates files, which define a P system, between formats; for instance, from P-Lingua format to binary format.

Each version of the library is associated with an extension of the programming language and simulation engines to cover more types of P systems [32]:

- **pLinguaCore 1.0**: The initial version. It was able to define active membrane P systems with rules to create new membranes by division. pLinguaCore was in very early state.
- **pLinguaCore 2.0**: Several cell-like P system models and built-in simulators for each supported model were added.
- **pLinguaCore 2.1**: Support for tissue-like P systems with division rules were added.
- **pLinguaCore 3.0**: The simulation algorithm called DCBA for Population Dynamics P systems (PDP systems) was added [44], along with a new binary output file for PDP systems. In this version, stochastic P systems were discontinued. Some bugs were fixed.
- **pLinguaCore 4.0**: Support for Spiking Neural P systems was added. Moreover, Tissue-Like P systems with Cell Separation Rules were also added. Some bugs were also fixed.

The development of pLinguaCore continued as part of MeCoSim tool [39], a flexible GUI to solve specific problems by means of P systems. Among the main computing models added within P-Lingua version inside MeCoSim we can find:

- sep1pt
- Simple kernel P systems.
- Probabilistic Guarded (Scripted) P systems.
- Evolutional-communication P systems.
- Fuzzy Reasoning Spiking Neural P systems.
- Cell-like Spiking Neural P systems.
- P systems with minimal cooperation.

- Tissue P systems with promoters.
- Spiking Neural P systems with delay on synapses.
- Spiking Neural P systems with autapses.

It is worth mentioning that MeCoSim is fully compatible with any version of P-Lingua, and uses pLinguaCore as its engine for parsing and simulation (additionally, it allows the simulation through other external simulators, but its predominant use is definitely based on pLinguaCore). Thus, MeCoSim mainly exposes the types of computing models, with their parsers and simulators, provided by the version of pLingua we decide to deploy with MeCoSim. Therefore, depending on which version of pLinguaCore (see previous sections) we use, a different set of models and simulators will be available to use in MeCoSim.

3. Design

The whole P-Lingua framework has been redesigned in this paper, i.e., not only the language but also its related tools and the simulation workflow [1]. The main ideas of the new design are the following:

- The user can define in P-Lingua 5 custom P system variants by means of three mechanisms: Definition of P system rule patterns (SubSection 3.1), definition of derivation modes (SubSection 3.2) and definition of simulation directives (SubSection 3.7). The first one is used to define the syntactic constraints for the rules allowed in the variant. The second one is used to define the way in which the rules should be executed. Finally, simulation directives can optionally be included to provide additional information to the simulator that cannot be expressed by the former methods. Most of the existing variants in the literature can be expressed in P-Lingua 5. In the website and in A there are several examples.
- Other improvements have been included to the language, such as procedural programming (SubSection 3.3), mathematical functions (SubSection 3.4), string functions (SubSection 3.5), user input/output functions (SubSection 3.6) and inclusion of header files (SubSection 3.8).
- A compiler has been implemented to parse P-Lingua 5 files, detect syntactic errors and generate the corresponding P system definitions in XML, JSON, Binary and low-level P-Lingua formats (SubSection 4.1). It is important to remark that the compiler does not simulate the defined P systems, it only validates the syntax and translates the definitions to low-level formats that external simulators can read and simulate. The information about the derivation mode and simulation directives is passed to the external simulator, which is in charge of interpreting this information and producing the P system computations requested.
- A P-Lingua 5 generic simulator is included in the toolkit (SubSection 4.2) which is able to simulate P systems according to the custom derivation modes that can be expressed in P-Lingua 5. External simulators for specific variants can also be used as explained in Section 5.

3.1. P system rule patterns

One of the most relevant syntactic differences among the existing P system variants is related to the type of P system rules that can be used on each of them. From the point of view of P-Lingua 5, the user can write patterns to define types of rules. Patterns are grouped into sets and included together with the corresponding P system definitions. In this way, the parser only recognizes the rules matching the included patterns. It is worth mentioning that the user does not have to write the corresponding patterns each time a P system is defined, since the files can be reused as explained in SubSection 3.8. Moreover, such patterns do not contain information about the derivation mode, i.e., they are used to recognize valid rules, not to provide information about how to apply the rules. Next, a brief description of the P-Lingua 5 syntax to define a set of rule patterns is included. A set of rule patterns can be written as follows:

```
!name
{patternl;
pattern2;
...
patternN;
}
```

where `name` is a unique identifier for the set of rule patterns (also called *rule type*) and `patternl` to `patternN` are rule patterns with the following syntax:

```
(X) LHR RELATION RHR :: ID(TYPE)
```

[1] For a more extensive description of the P-Lingua 5 framework, including examples and tutorials, please visit the P-Lingua 5 website [45] https://github.com/RGNC/plingua/.

where:

- `(X)` is optional. The symbol `X` can be an integer number or the question mark character: '`?`'. If such a string is included in the pattern, the corresponding matching rules must include an integer number between parentheses at the left of the rule. If `X` is the question mark character, such a number can be any integer number. Otherwise, such a number must be exactly the value in `X`. It can be used as a priority value, among other possibilities.
- `LHR` and `RHR` are patterns for the left-hand and right-hand sides of the matching rules, respectively. Let us note that rewriting rules usually include a left-hand side with objects that are consumed, a relational operation (typically a right arrow) and a right-hand side with the objects that are produced. Of course, there exist other more complex types of rules. Special characters can be used to define or constrain the electrical charges of the membranes involved in the rule. Membrane dissolution, creation, separation and/or division flags can be included. The allowed multiplicities of all the multisets of objects in the rule can be defined or constrained, as well as the identifiers for all the membrane labels. For the sake of a brief description, a more detailed explanation is omitted in this paper, but it can be found in the P-Lingua 5 website.
- `RELATION` can be the strings '`-->`', '`->`', '`<->`' or '`<-->`' representing the relation between the left-hand and right-hand sides of the rule. The meaning of such a relation is not fixed by the parser, it will be given by the simulator according to the semantics of the P system.
- `ID` is an optional string identifier and `TYPE` can be: '`double_t`', '`int_t`', '`long_t`' or '`string_t`', among other data types. The idea is to optionally associate a value of the data type to each rule. It can be used as a probability constant, a kinetic constant or a regular expression, among other possibilities.

Different sets of rule patterns can be grouped together in order to define a new P system variant. This is done using the command `@model`, where the derivation mode for such rules is also specified. Details on the syntax of this command will be explained in the next section.

Rules in a given P system definition will be recognized by P-Lingua 5 parser if and only they match any of the rule types included in the declaration of the corresponding variant. Any non-matching rule will throw a compiler-time error. Moreover, the parser associates a simulator directive to each rule with the name of the corresponding pattern set. In this manner, the simulator knows exactly what type of rule is each one.

3.2. Derivation modes

From an informal point of view, a *derivation mode* is a set of indications explaining, for a given configuration, which are the possible multisets of rules that the system can execute in the next step of computation. In other words, the derivation mode guides the behaviour of the function `selectRules`(C_i, R) from Algorithm 1. The underlying logic of this function can be better understood when presented as a sequence of three steps:

1. Checking rules eligibility: given a configuration C of a P system Π, each membrane can discard all its rules whose LHR is not satisfied by C (e.g. because of some object is missing, mismatching electrical charges, etc). Rules not discarded will be referred to as *eligible*.
2. Applicability filter with respect to derivation mode: eligible rules are not handled independently. Typically, derivation mode includes conditions over the whole multiset of rules to be applied.
3. Selection of the multiset of rules which will be actually applied: there may exist several multisets of rules complying with the applicability conditions. One of them is selected non-deterministically, and passed as an argument to the next function in Algorithm 1, namely `executeRules`(C, R').

Let us try to formalize this process. Given a configuration C of a P system Π, let R be the set of rules from Π, and let us assume that each rule $r \in R$ is associated with one and only one membrane in C. This assumption can be made without loss of generality, since rules associated with a label not appearing in C can be ignored, and if several membranes in C share the same label we can create a copy of the corresponding rules for each of such membranes. Then, we denote by $Appl(\Pi, C)$ the set of all multisets over R, such that the multiplicity of each rule ranges from 0 to the maximum number of times that such rule could be applied in C. That is, for each membrane in C, the multisets in $Appl(\Pi, C)$ may contain any of the rules associated to that membrane such that the "requisites" of their LHR are fulfilled in C, and for each rule its multiplicity is bounded by the maximum number of times that it could be applied if competition and conflicts were not taken into account.

The first restriction to be taken into account is the competition (several rules may want to use the same symbol). Depending on the variant of P system and the selected derivation mode, some further restrictions should be taken into account. A derivation mode indicates which of all the multisets of applicable rules are eligible to be applied to a given configuration. Classical examples of derivation modes are maximally parallel mode (*max*), where only non-extendable multisets of rules are applicable. Another derivation mode is sequential mode (*sequ*), that only allows one rule per configuration.

In P-Lingua 5, derivation modes can be specified in the code. A maximally parallel behaviour is assumed by default, although it can be restrained by adding restrictions for some of the rule types, via a number of special expressions, called *bound-terms*. More precisely, the definition of the derivation modes in P-Lingua 5 is integrated within the declaration of the P system variant, by means of the following sentence:

$$@\texttt{model(id)} = \texttt{boundTerm}_1, \ldots, \texttt{boundTerm}_n$$

where `@model` is a reserved keyword, `id` is the name of the variant being defined, and $\texttt{boundTerm}_1, \ldots, \texttt{boundTerm}_n$ is a sequence of bound terms with the additional condition that each rule type can appear only once in the whole sequence.

The syntax for such bound-terms can be described in a recursive way as follows:

1. A rule type (i.e. the `name` of a set of rule patterns previously declared) is a bound-term, which is interpreted as "unbounded";
2. `@xor(lb)` is a bound-term, where `lb` is a list of bound-terms;
3. `@n(lb)` is a bound-term, where $\texttt{n} \in \mathbb{N}$ and `lb` is a list of bound-terms such that each $B \in \texttt{lb}$ is either a rule type, or is the form `@m(lb')`, with $m \leqslant n$. In this case, `n` is said to be the *bound* of the bound-term.

Cases 2 and 3 will be referred to as *non-elementary* bound-terms.

A type of rule a is in the *context* of a bound-term B if:

- $B = a$; or
- $B = @\beta(\texttt{lb})$, with $\beta \in \mathbb{N} \cup \{\texttt{xor}\}$ and $\exists B\prime \in \texttt{lb}$ such that a is in the context of $B\prime$.

Therefore, given a P system Π of a variant X, defined as $@\texttt{model(X)} = \texttt{boundTerm}_1, \ldots, \texttt{boundTerm}_n$, the *applicability* of the rules of Π on a configuration C is defined as follows:

- If $\textit{boundTerm}_i = a$, for any $i = 1, \ldots, n$, then rules of rule type a can be applied in a maximal parallel way; and
- If $\textit{boundTerm}_i = @\texttt{s}(\texttt{lb})$ with $\texttt{s} \in \mathbb{N}$, for any $i = 1, \ldots, n$, then the multiset of rules to be applied will contain at most `s` rules of rule types within the context of $\textit{boundTerm}_i$ for each membrane in C. Note that the multiset of rules must be maximal (up to size `s` on each membrane, subject to availability of objects on each of them), and any other non-elementary bound $B\prime \in \texttt{lb}$ will involve stronger restrictions that must also be fulfilled.
- If $\textit{boundTerm}_i = @\texttt{xor}(\texttt{lb})$, for any $i = 1, \ldots, n$, then the system will non-deterministically choose one of the bound-terms $B\prime \in \texttt{lb}$ and will try to apply the rules of the types in the context of $B\prime$ accordingly, while ignoring the rest (that is, rule types in other elements of `lb` will not be applied in that step).

This syntax was created in a flexible way, but obviously it cannot cover all possible derivation modes that one may think of. Actually, as it was explained above, derivation modes can be customized, but there will always be an underlying "*bounded-by-rule maximally parallel*" behaviour.

The classical example for a non-trivial derivation mode is the one used in P systems with active membranes. Let a be object evolution rules, b send-in communication rules, c send-out communication rules, d dissolution rules, e division rules for elementary membranes and f division rules for non-elementary membranes. Taking into account that, while rules of type a can be applied in a maximal parallel way, only one rule of type b, c, d or e can be applied at most once for each membrane. Therefore, this derivation mode could be expressed as follows:

```
@model(active_membranes) = evolution_rule, @1(send_in_rule, send_out_rule,
dissolution_rule,
division_elem_rule, division_non_elem_rule);
```

Another interesting non-trivial derivation mode is the one used in tissue P systems with symport/antiport rules and division rules. In this model, communication rules can be applied multiple times in each cell, but if a division rule is applied in a cell, this cell is called to be blocked, and no communication with this cell is permitted. Therefore, this derivation mode could be defined as follows:

```
@model(tissue_sa_division) = @xor(communication_rule, @1(division_rule));
```

3.3. Procedural programming

From its early versions, P-Lingua supports modular and parametric programming to define P systems. A P-Lingua definition contains one or more independent modules defining P system rules, multisets of objects and membrane structures. There must be a main module to begin the P system definition. The rest of the modules can be called one or more times from any other module.

Moreover, P-Lingua uses a technique known as *parametric sentences* in order to easily define sets of rules. For instance, the next P-Lingua sentence defines 1000 rules in only one line of code:

```
[ai --> bi + 1]'h: 1 <= i <= 1000;
```

Indeed, the concept of parametric sentences can be applied to other types of P-Lingua declarations, including module calls. For instance, the next sentence executes 1000 calls to module `f` with parameter `i`:

```
f(i): 1 <= i <= 1000;
```

Furthermore, it is possible to define blocks of sentences by using curly braces. A block of sentences can also be associated to parameters. The next example combines all the concepts described above:

```
{
a{j} [b{i}]'1 --> b{j + 1} [a{i-1}]'1;
+[c{i + j} --> c{i-j}, d{k}]'2: 1<=k<=5;
f(i,j);
}:1 <= i <= 10, 1<=j<=10;
```

This example describes a block parametrized with i and $j, 1 \leqslant i,j \leqslant 10$, with two rules: the first one is associated with membrane labeled by 1 and neutral charge, taking object a_j in the parent and b_i inside membrane 1, and evolve them to object b_{j+1} in the parent and object a_{i-1} in the membrane; the second rule is associated with membrane labeled by 2 and positive charge, and evolves object c_{i+j} to objects c_{i-j}, d_k inside that membrane, with $1 \leqslant k \leqslant 5$. Moreover, function f is called with parameters i and j. This function is defined elsewhere in the P-Lingua file.

Modular and parametric programming in P-Lingua can be seen as a constrained type of procedural programming, in which procedures (modules) contains a series of definitions to be carried out. But procedural languages are also imperative languages in which control structures can be used to specify the flow of control in the program.

In P-Lingua 5, control structures have been included in order to fully support the paradigm of procedural programming. Control structures can be written by using standard C syntax. The next control structures are allowed in P-Lingua 5: Multiple alternatives structure (*if-else*), repeat-while structure (*while*) and repeat-for structure (*for*). The next example defines a set of rules by using control structures in P-Lingua 5:

```
for (i = 1; i<= 10; i++) {
  j = 0;
  while (j < 10) {
    if ( (i+j) % 2 == 0) {
      // A rule definition inside a C style code.
      [a{i} --> b{j}]'1;
    }
    j++;
  }
  }
```

On the other hand, modules in P-Lingua 5 can optionally return a value by using the *return* sentence (as in standard C). In this way, the user can write algorithms returning values. Such values can be used, for example, as parameters for rule definitions. Variables (numerical, logical and string) as well as arithmetic, string, logical and bitwise operations are supported. For more information about procedural programming in P-Lingua 5, please visit the website [45].

3.4. Mathematical functions

P-Lingua 5 natively includes a library of mathematical functions. Such functions can be used in any numerical expression. For example:

```
[a{i} --> b{i}]'h: l<=i <= floor(sqrt(N));
```

The next functions can be used in P-Lingua 5, with the same meaning than the corresponding functions in the standard C library *math.h*: abs(a), acos(a), acosh(a), asin(a), asinh(a), atan(a), atanh(a), atan2(b,a), ceil(a), cos(a), cosh(a), exp(a), exp2(a), floor(a), log(a), pow(a), round(a), sin(a), sinh(a), sqrt(a), tan(a), tanh(a). The next functions have a special meaning:

- random(a): This function returns an i.i.d. random natural number in the range [0,a).
- log(a,b): This function returns the logarithm of a in base b.
- logN(a): This function returns the logarithm of a in base N, where N is a natural number > 1. For instance, log2(3).

3.5. String functions

The next string functions are supported in P-Lingua 5:

- len(str): This function returns the length of a string.
- pos(str,n): This function returns the character at position n in string *str*.
- cat(str0,str1): This function returns the string resulting of concatenating strings *str*0 and *str*1.
- cmp(str0,str1): This function returns *true* if strings *str*0 and *str*1 are equals, *false* otherwise.
- substr(str,a,n): This function returns the substring of *str* starting at position a with length n.

3.6. Input/output functions

In P-Lingua 5 it is possible to generate P systems in an interactive way. In order to achieve this, three special input/output functions are natively supported:

- print(msg): This function prints a message to the standard output during the definition of the P system.
- printn(msg): This function is equals to the previous one, but including a line break.
- scan(): This function reads a numerical value from the standard input.

As example, the next P-Lingua fragment asks the user for a number n and then generates n rules.

```
print(``Enter a number: ");
n = scan();
[a{i} --> a{i + 1}]'h: l<=i <=n;
printn(n, " rules have been generated");
```

3.7. Simulator directives

P-Lingua can be seen as a programming language to declare P systems. Its syntax allows to define the ingredients employed in most of the existent variants: membranes, objects and rules. Today, many general-domain programming languages include elements for high-level information that can be passed to the compiler (e.g. in C++/OpenMP they are called directives, in Python decorators, etc.). These directives can be either dismissed by the compiler if they are not supported, or used to improve the behavior otherwise.

A novel feature of P-Lingua 5 is the introduction of *simulator directives*. That is, high-level information that can be provided along with the definition of the P system, and that are passed straight to the simulator. If the simulator understands them, they will be used to enhance the simulation or just skipped if they are not supported. In this way, the P system designer can provide more information to the simulator rather than only the P system elements. In fact, simulation directives can be used to optimize the simulation algorithm by providing information about how to simulate the execution of certain rules. For instance, ecosystem models on PDP systems are usually designed by cycles which are repeated over the simulation time (representing years, seasons, etc.), and these cycles are composed of modules of rules (representing phases like reproduction, migration, etc.). The information of cycles and modules are not part of PDP system syntax, so it cannot be represented in P-Lingua, and hence, given to the simulator. However, using simulation directives, P system designers can attach this information to the P-Lingua file, so that the simulator knows, for each step, which rules can be executed according

the design of modules [30]. Moreover, directives can be used to improve the expressivity of the language, opening the way to include more complex elements introduced by variants whose syntax is far from the classic models.

Simulator directives can be defined as follows:

```
@directive_name = directive_value;
```

A directive can be also defined without value; in such a case, the default value of 1 is assigned. Simulation directives are numeric or string literals that can be associated to the rules in the P system definition and/or the whole P system definition. Specifically:

- *Global directives* are declared at the global scope and are attached to the P system. For example:

```
@number_of_stages = 5;
@sequence = ‘‘2, 1, 4, 3, 5";
@levels = ‘‘high, low"
```

- *Local directives* are declared at the rule scope after providing the rest of the syntax (i.e. before the semicolon). Thus, these directives are only associated to particular rules. For example:

```
[A]’1 --> [B]’1 [C]’1 @phase = ‘‘generation";
[S{i} G]’1 --> [SP{i}]’1 :: 0.8: 0 < i<N @stage = ‘‘feed";
```

An example is shown in Section 5, based on the simulator published in [30].

3.8. Header files

In P-Lingua 5, definition modules can be written in different files. A file can include the modules and definitions in another file by using the next include directive: `@include < file>`, where *file* is the absolute or relative path to the file to be included. Default directories can be configured in the P-Lingua 5 compiler as described in SubSection 4.1.

This approximation produces more reusable and legible code. In particular, it is recommended to write rule patterns and derivation modes, i.e., definition of variants, in separate files. In A, there are some examples of such files.

4. The P-Lingua 5 toolkit

A software toolkit for the Linux Operating System is included in this work. The toolkit has been developed in C++ under GNU/GPLv3 license. It is composed by two command-line programs: a compiler and a simulator. The toolkit can be downloaded from the P-Lingua 5 website, including instructions about the installation and its dependencies.

4.1. The P-Lingua 5 compiler

The compiler is used to parse P-Lingua 5 files and generate output files codifying the defined P systems (including derivation modes) in four alternative types of file formats: XML, JSON, Binary and low-level P-Lingua. The next properties are guaranteed after the execution of the compiler, if no errors occur:

- Only one output file is generated, if no errors. Such a unique file contains the definition of the P system given by the input P-Lingua 5 files.
- Neither procedural programming nor parametric sentences nor functions or modules are included in the output file. That is, the compiler follows the flow control in the input files, executing all the functions, modules, input/output calls and *unrolling* all the parametric sentences.
- The output file is free of syntactic and semantic errors, since the compiler has checked the input files. This implies that all rules have been successfully matched to a defined rule pattern.
- A simulation directive containing the name of the corresponding rule pattern is automatically associated to each rule, for example, "send-in", "send-out", "division", etc. This information can be used by the simulator to classify and sort the rules in an efficient manner.

The user can set several parameters by using the command-line options, such as the default input directories, the output file format, the verbosity level, and others. A detailed description of the command-line options can be found in the P-Lingua 5 website, as well as the description of the output file formats. As stated above, the output file formats are XML, JSON, Binary and low-level P-Lingua. XML and JSON are well-known data-interchange file formats. The Binary file format generates compressed binary files, i.e., non-text files. The *Cereal* library[2] was used for the serialization to XML, JSON and Binary. Examples on how to use the Cereal library in order to easily deserialise the XML, JSON and Binary files can be found in the website.

The low-level P-Lingua is, indeed, a subset of P-Lingua 5. The idea is to write the P system definition after unrolling sentences, executing functions, following procedural programming instructions, assigning automatic simulation directives, etc. Low-level P-Lingua can be used to debug P systems. For example, it is easy to count the number of rules in a low-level P-Lingua definition. The next example shows a P-Lingua 5 definition using the variant described in Appendix A.1, and the corresponding output in low-level P-Lingua for $n = 2$:

```
@include "p_active.pli"// See Appendix A.1.
@model < active_membranes>
def main() {
  @mu = [[]'2]'1;
  print("Input n: ");
  n  = scan();
  multisets(n);
  rules(n);
}
def multisets(n) {
  @ms(1) = a1;
  @ms(2) = bn;
}
def rules(n)
[a{i}]'1 --> [ai + 1, a{i + 1}]'1: 1<=i<=n;
[b{i}]'2 --> [bi-1]'2 [b{i-1}]'2: 1<=i<=n;
a{n + 1} []'2 --> +[c]'2;
}
```

The corresponding low-level P-Lingua is:

```
@model < active_membranes>
@model(active_membranes)
  = evolution_rule, @l(send_in_rule, send_out_rule,
  dissolution_rule, division_elem_rule, division_non_elem_rule);
def main() {
  @mu = [[]'2]'1;
  @ms(1) = a1;
  @ms(2) = b2;
  [a{1}]'1 --> [a{2}*2]'1 @ pattern = ''evolution_rule";
  [a{2}]'1 --> [a{3}*2]'1 @ pattern = ''evolution_rule";
  [b{1}]'2 --> [b{0}]'2 [b{0}]'2 @ pattern = ''division_elem_rule";
  [b{2}]'2 --> [b{1}]'2 [b{1}]'2 @ pattern = ''division_elem_rule";
  a{3} []'2 --> +[c]'2 @ pattern = ''send_in_rule";
}
```

4.2. The P-Lingua 5 simulator

The toolkit includes a command-line simulator which is able to simulate P system computations. The simulator receives the XML/JSON/Binary output file from the compiler and interprets the derivation mode in the file. Only derivation modes that can be defined in P-Lingua 5 can be simulated by this simulator. For other derivation modes or special simulation algorithms, the output of the compiler can be connected to external simulators (see Section 5).

[2] https://uscilab.github.io/cereal/

The parameters and instructions of the simulator are described in the P-Lingua 5 website. The simulator follows Algorithm 1. It is able to simulate one of more steps of computation following a branch of the computation tree given by the defined P system. The user can set how many steps to simulate or if the simulation should run until a halting configuration. The simulator used pseudo-random numbers to simulate the non-determinism. The user can set the random seed for each simulation in order to obtain the same results. Indeed, the simulator has an efficient mode in which no pseudo-random numbers are required because the first option is always selected for each decision in which randomization should be used. Such a mode produces always the same computation for a given P system.

The P system configuration can be saved in XML, JSON or Binary file format after each simulation step. In this manner, it is possible to continue a previous simulation by using such a configuration file together with the P system definition. The user can obtain a trace about the simulation if the verbosity parameter is set. The next example is the trace of a simulation until a halting configuration for the P system defined in SubSection 4.1:

```
    CONFIGURATION: 0
    SKIN MEMBRANE ID: 0, Label: 1, Charge: 0. Multiset: a{1}
    MEMBRANE ID: 1, Label: 2, Charge: 0. Multiset: b{2}
-----------------------------------------------
    STEP 1:
    Membrane ID: 0
      1 * [ a{1} ]'1 --> [ a{2}*2 ]'1 @ pattern = "evolution_rule"
    Membrane ID: 1
      1 * [ b{2} ]'2 --> [ b{1} ]'2 [ b{1} ]'2 @ pattern = "division_elem_rule"
***********************************************
    CONFIGURATION: 1
    SKIN MEMBRANE ID: 0, Label: 1, Charge: 0. Multiset: a{2}*2
    MEMBRANE ID: 1, Label: 2, Charge: 0. Multiset: b{1}
    MEMBRANE ID: 2, Label: 2, Charge: 0. Multiset: b{1}
-----------------------------------------------
    STEP 2:
    Membrane ID: 0
      2 * [ a{2} ]'1 --> [ a{3}*2 ]'1 @ pattern = "evolution_rule"
    Membrane ID: 1
      1 * [ b{1} ]'2 --> [ b{0} ]'2 [ b{0} ]'2 @ pattern = "division_elem_rule"
    Membrane ID: 2
      1 * [ b{1} ]'2 --> [ b{0} ]'2 [ b{0} ]'2 @ pattern = "division_elem_rule"
***********************************************
    CONFIGURATION: 2
    SKIN MEMBRANE ID: 0, Label: 1, Charge: 0. Multiset: a{3}*4
    MEMBRANE ID: 1, Label: 2, Charge: 0. Multiset: b{0}
    MEMBRANE ID: 2, Label: 2, Charge: 0. Multiset: b{0}
    MEMBRANE ID: 3, Label: 2, Charge: 0. Multiset: b{0}
    MEMBRANE ID: 4, Label: 2, Charge: 0. Multiset: b{0}
-----------------------------------------------
    STEP 3:
    Membrane ID: 1
     1 * a{3} [ ]'2 --> +[ c ]'2 @ pattern = "send_in_rule"
    Membrane ID: 2
     1 * a{3} [ ]'2 --> +[ c ]'2 @ pattern = "send_in_rule"
    Membrane ID: 3
     1 * a{3} [ ]'2 --> +[ c ]'2 @ pattern = "send_in_rule"
    Membrane ID: 4
     1 * a{3} [ ]'2 --> +[ c ]'2 @ pattern = "send_in_rule"
***********************************************
    CONFIGURATION: 3

    SKIN MEMBRANE ID: 0, Label: 1, Charge: 0
    MEMBRANE ID: 1, Label: 2, Charge: +. Multiset: b{0},c
    MEMBRANE ID: 2, Label: 2, Charge: +. Multiset: b{0},c
    MEMBRANE ID: 3, Label: 2, Charge: +. Multiset: b{0},c
    MEMBRANE ID: 4, Label: 2, Charge: +. Multiset: b{0},c
```

5. Connection to external simulators

Some P system variants require very sophisticated simulation algorithms. For example, three different algorithms (BBB, DNDP and DCBA) were defined in the literature for Population Dynamics P (PDP) systems [30]. This variant is employed as a formal modelling framework for population dynamics, and their semantics was first implemented with BBB algorithm. The

behaviour of PDP systems when rules compete for resources (i.e. two rules have intersections in their left-hand sides) was refined with DNDP and later with DCBA. In such a situation, a generic simulator cannot be extended to support special algorithms without loosing the achieved generality. Therefore, simulators for specific variants are needed. Moreover, expert developers can prefer to implement their own simulators because of several reasons: using different languages rather than C++, exploiting parallel architectures such as GPUs, using different technologies, freedom of coding, etc. In such situations, the P-Lingua 5 framework can still help since it can provide the input module for the simulators, offering a high-level input language and a compiler. In this way, developers can forget about parsing input files, declaring syntax, rule pattern matching, etc.

As described in Section 4.1, the compiler included in the toolkit translates from P-Lingua 5 files to either XML, JSON, binary or even low-level P-Lingua file format. With such a compiled file, the external simulator has the full description of the P system to be simulated with the guarantee that the described P system is conformed to the constraints of the variant. Since the syntax has been already checked by the compiler, the external simulator is only in charge of implementing the corresponding derivation mode.

The concepts described above were applied in [30], where a preliminary version of P-Lingua 5 was used as input language for a PDP system simulator running on GPUs called ABCDGPU [41]. A previous version of the compiler presented in this paper was applied to translate the P system definitions to a Binary file format. The required data structures by ABCDGPU are optimized for the CUDA programming model. Therefore, a translation from the serialized data structures in the Binary file format to the required data structures by ABCDGPU was implemented.

The implementation of ABCDGPU in [30] employed the concept of simulator directives to pass high-level information to the simulator. Certain types of P systems are usually designed by applying an algorithmic scheme, where the computation of the model is divided into stages or modules. For instance, the computation of P systems solving SAT usually contains a stage for generation of membranes and another for checking the solution. Regarding ecosystem modelling, the models work in cycles of a fixed size (in terms of transitions). The execution of rules along one cycle is subdivided in modules (e.g., for reproduction, feeding, etc.). This means that the model designer knows which rules are eligible at each step, because one rule can be only in one module. This information is not part of the P system definition, and so far P-Lingua did not allow a way to pass this kind of data. By using P-Lingua 5's directives, the information of modules can be sent to the ABCDGPU simulator, and hence, it is able to discard rules from selection stages when they belong to non-active modules in a step, achieving a speedup.

6. Comparison with other simulation frameworks

In the previous sections the P-Lingua 5 framework has been presented, enriching the features available in its predecessors while keeping the original spirit of a standard language and powerful parsers making sure that the computational models used meet the specifications of the theoretical computing models they belong to.

Other software products have pursued some converging goals, while preserving significant differences. For instance, UPsimulator [46] was conceived to provide a flexible simulator that would accept the specification of systems and their simulation, but not putting the focus on the correctness, reliability with respect to the formal models being properly captured, nor similarity with the syntax of the P systems in the academic papers of the field. On the contrary, the main idea in their case was to provide a general simulator able to combine a number of possible types of rules and features and make them run in a general simulator. For sure, that approach can be very helpful for some people to simulate arbitrary combinations of rules with a specific execution strategy. However, most of the types and variants of P systems present particular simulation algorithms and very specific combinations of rules and semantic aspects to consider, what makes it difficult to express the right properties being adopted by some general oracle, generic simulator able to simulate with different strategies depending on many aspects of the variants adopted in the framework. Additionally, the syntax UPsimulator proposes for the introduction of P systems is looking for common patterns and simplicity, in a more programmatic flavor, more developer-oriented than researcher-oriented.

In this context, P-Lingua 5 aims to take the best of both approaches. Thus, it keeps the focus of P-Lingua 4 [34,32] with regards of similarity with P system designs expressed in papers; errors detection with respect to the intended variants of P systems being covered by the approach; and reliability in the perfect matching between the semantics and the dynamic aspects detailed in the proposal of each new type or variant of P system. Along with that, it also expands the scope, similarly to UPSimulator, in the sense of providing generic tools allowing the use of the parsing and simulation tools for new, unseen, novel types and variants of P systems, with some extensions of P-Lingua language to accept the meta-definition of model types, including both syntactic and semantic aspects, and including certain elements in the definition of the model types covering dynamic aspects for the simulation. With these extensions, a specification of a P-Lingua model can be expressed for each type and variant of P system, in such a way that the P system designers providing specific solutions will be able to detect if their designs are compliant with those specifications, but this mechanism will also allow to include future model description with this meta-language, making the framework well adapted to accept many possible combination of features for future variants.

Despite the fact that many other software products have been developed within membrane computing community, it is worth highlighting that most of them are focused on specific frameworks (as metabolic P systems, kernel P systems or multicompartmental stochastic P systems), or directly addressing particular problems with ad hoc simulators, as extensively

described in [25]. To the best of our knowledge, no other software tools have been developed with the wide scope we can find in UPSimulator or P-Lingua frameworks.

7. Conclusions

P-Lingua 5 has been presented along the paper, extending the scope of P-Lingua 4 and its predecessors. Originally, the idea was to provide a number of tools for P system designers to describe or specify their P systems of different types using a *standard* language, instead of building software tools on their own. Thus, the focus was in bringing them the technology to make their lives easier, with a syntax being as close as possible to the way the researchers usually wrote their membrane systems in papers. Along with this primary intent, it was crucial to detect any possible error in the designs prepared by the authors, in such a way that, for each well-known type or variant of P system, P-Lingua would provide a parser to detect all possible syntactic or semantic errors. Finally, it would be necessary to test the P systems designed in motion, in order to automatically simulate the behavior of the system.

In order to achieve such goals, the P-Lingua framework has been redesigned, as detailed along the paper, including rich syntactic features to define custom computing models (with their own P system rules accepted and the derivation modes capturing their semantics). Besides, interesting programming structures and simulator directives (high-level information to be bypassed to the simulator) have been added to the language. Additionally, the P-Lingua toolkit has been provided, including a compiler to parse the source files and generate the representation of the systems in several possible output formats, and a simulator to receive a P system in some of such formats and emulate certain computations, according to the syntactic and semantic elements defined in its specification. Finally, some steps forward has been made regarding the connection with external simulators, following a decoupled model, as described in previous sections.

It is important to note that P-Lingua 5 is a generic tool for membrane computing, but it is not a *panacea*. Whilst most of the common P system variants can be defined in P-Lingua 5 by writing rule patterns and derivation modes, there are types of membrane systems that do not fit in the current syntax, e.g., Enzymatic Numerical P systems (ENPS) or Dendrite P systems (DeP) and their inclusion might be analysed in further P-Lingua releases.

Looking at the future, our new proposal should be further tested against new possible variants of P systems not covered by previous versions of the framework. In fact, it would be interesting to define some kind of benchmarking sets of models for particular problems of different nature and size, that stress the tools at both functional and performance level, and serve as a valuable tool for this and possible future versions of the infrastructure provided by P-Lingua or other software projects within Membrane computing community. Additionally, the integration with other simulation tools could be explored in detail through some standardized protocol, increasing real inter-operability serving relevant problems being solved by means of P systems.

CRediT authorship contribution statement

Ignacio Pérez-Hurtado: Conceptualization, Investigation, Software, Writing - original draft, Writing - review & editing. **David Orellana-Martín:** Conceptualization, Investigation, Writing - review & editing. **Miguel A. Martínez-del-Amor:** Conceptualization, Investigation, Writing - review & editing. **Luis Valencia-Cabrera:** Methodology, Writing - review & editing. **Agustín Riscos-Núñez:** Funding acquisition, Supervision, Project administration.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgement

This work was supported by "FEDER/Ministerio de Ciencia e Innovación – Agencia Estatal de Investigación/ _Proyecto (TIN2017-89842-P)" - MABICAP. D. Orellana-Martín also acknowledges Contratación de Personal Investigador Doctor. (Convocatoria 2019) 43 Contratos Capital Humano Línea 2. Paidi 2020, supported by the European Social Fund and Junta de Andalucía.

Appendix A. Case studies

A.1. Cell-like P systems

P systems with active membranes is one of the most active areas in, for instance, the search of frontiers of efficiency. In the process of design of an efficient solution to presumably hard problems, a simulator is crucial to see if the family of P systems is functioning correctly. There are different kinds of rules in this variant:

- Object evolution rules: $[a \to u]^{\alpha_h}$, where $a \in \Gamma, u \in M_f(\Gamma), h \in H$ and $\alpha \in \{+, -, 0\}$.
- Send-in communication rules: $a[\ \ ]_h^\alpha \to [b]_h^\beta, a, b \in \Gamma, \alpha, \beta \in \{+, -, 0\}$ and $h \in H$.
- Send-out communication rules: $[a]_h^\alpha \to b[\ \ ]_h^\beta, a, b \in \Gamma, \alpha, \beta \in \{+, -, 0\}$ and $h \in H$.
- Dissolution rules: $[a]_h^\alpha \to b, a, b \in \Gamma, \alpha \in \{+, -, 0\}$ and $h \in H$.
- Division rules for elementary membranes: $\mathrm{x}[a]_h^\alpha \to [b]_h^{\beta_1}[c]_h^{\beta_2}$, where $a, b, c \in \Gamma, \alpha, \beta_1, \beta_2 \in \{+, -, 0\}$ and $h \in H$.
- Division rules for non-elementary membranes: $[[\ \ ]_{h_1}^{\alpha_1}[\ \ ]_{h_2}^{\alpha_2}]_h^\alpha \to [[\ \ ]_{h_1}^{\beta_1}]_h^\beta[\ \ ]_{h_2}^{\beta_2}]_h^\beta$, where $\alpha, \alpha_1, \alpha_2, \beta, \beta_1, \beta_2 \in \{+, -, 0\}$ and $h \in H$.

We can define the syntax of the rules of P systems with active membranes in P-Lingua as follows:

```
!evolution_rule
{
  ?[a --> v]'h;
  ?[a -->]'h;
}
!send_in_rule
{
  a?[]'h -->?[b]'h;
}
!send_out_rule
{
  ?[a]'h --> b?[]'h;
}
!dissolution_rule
{
  ?[a]'h --> b;
  ?[a]'h -->;
} !division_elem_rule
{
  ?[a]'h -->?[]'h?[]'h;
  ?[a]'h -->?[b]'h?[]'h;
  ?[a]'h -->?[]'h?[b]'h;
  ?[a]'h -->?[b]'h?[c]'h;
}
!division_non_elem_rule
{
  ?[?[]'hl?[]'h2]'h -->?[?[]'hl]'h?[?[]'h2]'h;
}
@model(active_membranes) =
evolution_rule, @l(send_in_rule, send_out_rule, dissolution_rule, division_elem_rule,
  division_non_elem_rule);
```

This model should be saved in a file called, for instance, `p_active.pli`. A simple P system with active membranes (as the one in Fig. 2) could be described as follows:

```
@model < active_membranes>
@include "p_active.pli"
def main()
{
@mu = [[]'2]'1;
@ms(1) = c;
@ms(2) = a, b{0};
[a]'2 --> +[a]'2 -[a]'2;
+[a]'2 --> +[a]'2 [a]'2;
[b{i} --> b{i + 1}]'2: 0 <= i  < 100;
+[b{i} --> b{i + 1}]'2: 0 <= i  < 100;
-[b{i} --> b{i + 1}]'2: 0 <= i  < 100;
```

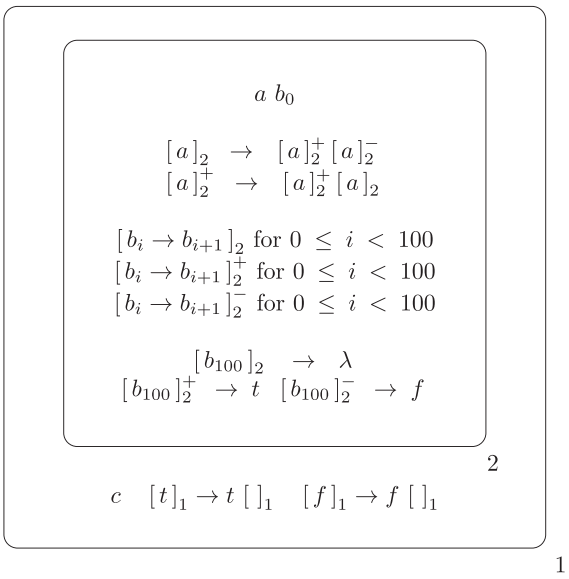

Fig. 2. Cell-like P system with active membranes.

```
[b{100}]'2 -->;
+[b{100}]'2 --> t;
-[b{100}]'2 --> f;
[t]'1 --> t []'1;
[f]'1 --> f []'1];
```

A.2. Tissue-like P systems

The main difference between tissue P systems and cell-like membrane systems is that, while the underlying structure is explicitly given as a rooted tree, the graph of tissue P systems is implicitly given by the rules of the systems. Rules of a tissue-like membrane system with symport/antiport rules and division rules can be the following ones:

- Communication rules: $(i, u/v, j)$, where $u, v \in M_f(\Gamma)$ and $0 \leqslant i, j \leqslant q$.
- Division rules: $[a]_h \rightarrow [b]_h[c]_h$, where $a, b, c \in \Gamma$ and $h \in H$.

These systems can be defined as follows:

```
!communication_rule
{
[a]'h1 <--> []'h2;
}
!division_rule
{
?[a]'h -->?[]'h?[]'h; ?[a]'h -->?[b]'h?[]'h; ?[a]'h -->?[]'h?[b]'h; ?[a]'h -->?[b]'h?[c]'h;
}
@model(tissue_symport_antiport_division) = @xor(communication_rule, @1(division_rule));
```

Once the model has been implemented (let us think that it is saved in a file called `p_tissue.pli`. Then, we can use it to define a tissue P system (the one shown in Fig. 3) as follows:

```
@model < tissue_symport_antiport_division>
@include "p_tissue.pli"
def main() {
@mu = [[]'1 []'2 []'3]'0;
@ms(0) = app{i}: 0 <= i  < 100;
@ms(0) = a{i}: 1 <= i  < 100;
@ms(0) += b; @ms(1) = a0;
[a{i}]'1 --> [api]'1 [ap{i}]'1: 0 <= i  < 100;
[ap{i}]'1 <--> [app{i},b]'0: 0 <= i  < 100;
[app{i}]'1 <--> []'2: 0 <= i  < 100;
[app{i}]'2 <--> []'3: 0 <= i  < 100;
[app{i}]'3 <--> [a{i + 1}]'0: 0 <= i  < 100;
[a{i}]'3 <--> [b]'1: 0 <= i  < 100;
}
```

A.3. Spiking Neural P systems

Spiking Neural P systems are brain-inspired membrane systems whose working alphabet contains only one object a, called *spike*, and represents the electrical impulses that flow through the neurons of the system. The rules of a basic SNP system are the following:

- Spiking rules: $E/a^c \rightarrow a; d$, where $a \in O$ E is a regular expression over $\{a\}, c, d \in \mathbb{N}$ and $c \geqslant 1$.
- Forgetting rules: $a^c \rightarrow \lambda$, where $c \in \mathbb{N}$ and $c \geqslant 1$.

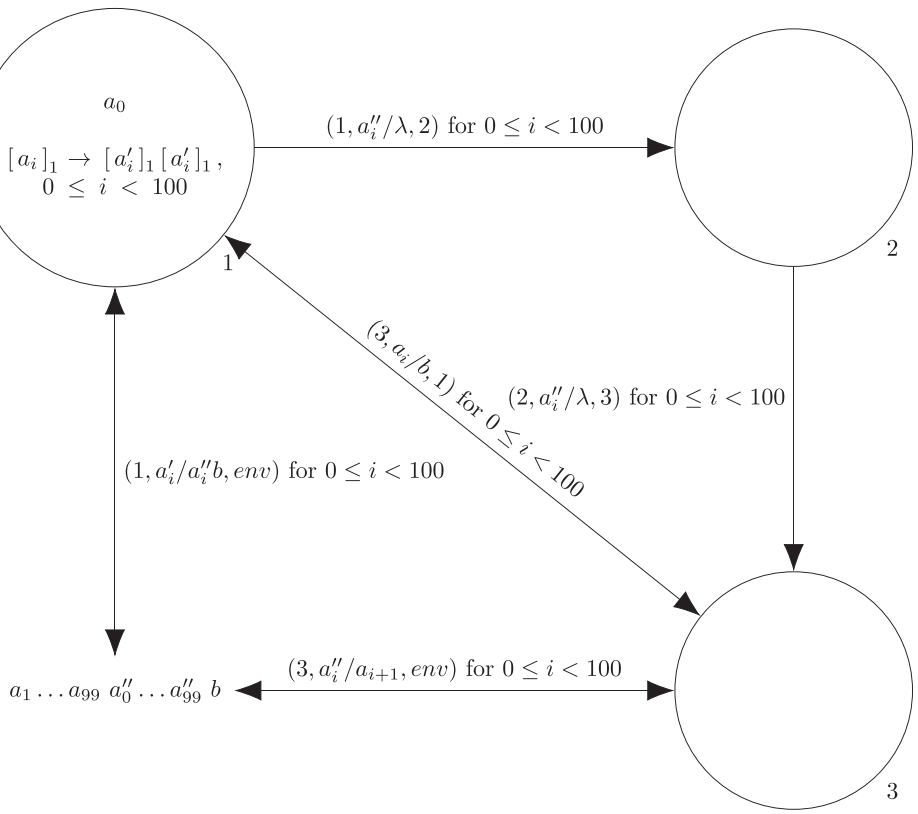

Fig. 3. Tissue P system.

As the syntax is quite different from the previous ones, new features must be evidenced in the corresponding P-Lingua rules:

```
!spiking_rule
{
[u --> a]'h;
d [u --> a]'h;
[u --> a]'h :: E(string_t);
d [u --> a]'h :: E(string_t);
}
!forgetting_rule
{
[u -->]'h;
}
@model(spiking_neural) = @l(spiking_rule, forgetting_rule);
```

The structure of SNP systems is defined by a set of ordered pairs *syn* representing the synapses of the system. Let `p_spiking.pli` be the file where the model is saved. Then, to create a test spiking neural P system (as the one shown in Fig. 4), a new file must be created with the following contents:

```
@model < spiking_neural>
@include "p_spiking.pli"
def main()
{
@mu = [[]'1 []'2 []'3 []'4]'0;
@syn = "(1, 2), (1, 3), (2, 4), (3, 4)"
@in = "1";
@out = "4";
@ms(1) = a*30; [a*3 --> a]'1;
1 [a --> a]'2;
1 [a*2 --> a]'3 :: "a*3";
[a -->]'3;
}
```

Let us recall again that the aim of P-Lingua 5 is providing a useful generic tool for the specification of the syntax and semantics of P systems. However, it is not possible and is therefore out of its scope covering every possible type and variant of P systems conceived by researchers. In any case, the project is not closed to exploring its capabilities to specify other variants of P systems. For instance, as SN P systems have attracted major attention over the last few years (including recent applications [47,48]), some recent variants of such systems may be explored, as it might be the case of Spiking Neural P Systems with Communication on Request [49,50], among others.

A.4. PDP systems

Population Dynamics P systems is a hybrid variant consisting of a directed graph of n environments, and inside each environments there is only one cell-like P system with q membranes. PDP systems have two types of rules (see [30] for more details):

- Skeleton rules: $u[\,v]_i^{\alpha} \xrightarrow{f_{rj}} u\prime[\,v\prime]_i^{\alpha\prime}$ where $u, v, u\prime, v\prime$ are multisets over the working alphabet associated to the P systems $(M_f(\Gamma)), 1 \leqslant i \leqslant q$ (label of membrane in the P system), $u + v \neq \varnothing$ and $\alpha, \alpha\prime \in \{0, +, -\}, f_{rj}$ is a computable function that depends on the rule r and environment j $(1 \leqslant j \leqslant n)$ from $\{1, \ldots, T\}$ to $[0, 1]$..
- Communication rules (between environments): $(x)_{e_j} \xrightarrow{p} (y_1)_{e_{j_1}} \cdots (y_h)_{e_{j_h}}$ where $x, y_1, \ldots, y_h$ are objects from the alphabet associated to the environments (Σ), $(e_j, e_{j_l}) \in S$ (part of the directed graph), $(1 \leqslant l \leqslant n)$ and p is a computable function from $\{1, \ldots, T\}$ to $[0, 1]$.

We can define the syntax of the rules of PDP systems in P-Lingua as follows:

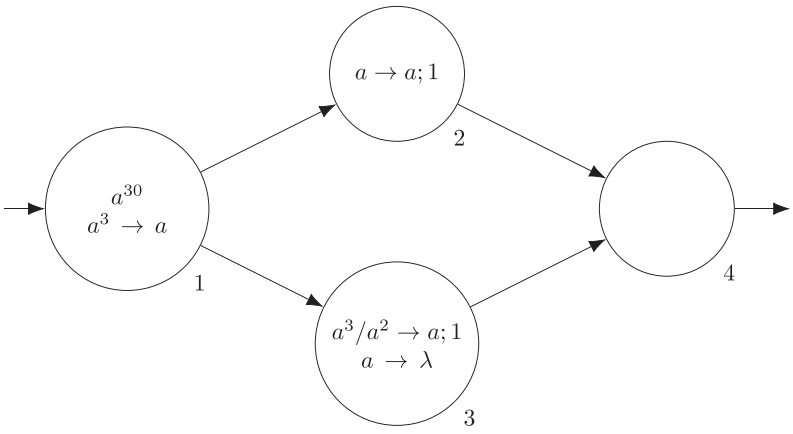

Fig. 4. SN P system.

```
!skeleton_rule
{
ul?[vl]'h -> u2?[v2]'h :: probability(double_t);
}
!communication_rule
{
[[a]'el []'e2]'p -> [[]'el [b]'e2]'p :: probability(double_t);
[[a]'e]'p -> [[b]'e]'p :: probability(double_t);
[[a]'e]'p -> [[]'e]'p :: probability(double_t);
[[a]'el]'p -> [[b]'e2]'p :: probability(double_t);
}
@model(probabilistic) = skeleton_rule,communication_rule;
```

Next, we will illustrate how to simulate the model with P-Lingua 5 and the external simulator ABCDGPU [41,30], by using the test example (see Fig. 5) employed to analyse the DCBA algorithm in [44]. The file was taken directly from P-Lingua 4 and we just added the second line that includes the model of PDP systems. We add in comments at the beginning of each rule the id read by the simulator from the generated Binary file, while the order in which they are defined is the same than in the experiments in [44].

```
@model < probabilistic>
@include "pdp_model.pli"
def main()
{
@mu = [[[[]'2]'1]'l0l,l0l]'p;
@ms(2,l0l) = a*90, b*72, c*66, d*30;
@ms(1,l0l) = a*60;
@ms(l0l,l0l) = b;
/*r3*/ [a*4, b*4, c*2]'2 --> e*2 []'2 :: 0.7;
/*r4*/ [a*4, b*4, c*2]'2 --> [e*2]'2 :: 0.2;
/*r5*/ [a*4, b*4, c*2]'2 --> [e, f]'2 :: 0.1;
/*r6*/ [a*4, d*1]'2 --> f*2 []'2 :: 1;
/*r7*/ [b*5, d*2]'2 --> g*2 []'2 :: 1;
/*r0*/ b -[a*7]'1 --> -[h*100]'1 :: 1;
/*r1*/ a*3 []'2 --> [e*3]'2 :: 1;
/*r2*/ a, b []'2 --> -[g*3]'2 :: 1;
}
```

Finally, we will assume that both P-Lingua 5 and ABCDGPU simulator are compiled and installed in the system. The steps to reproduce the experiments are the following (tested only in Ubuntu 18.04, 20.04 and CentOS 7 systems with CVMFS):

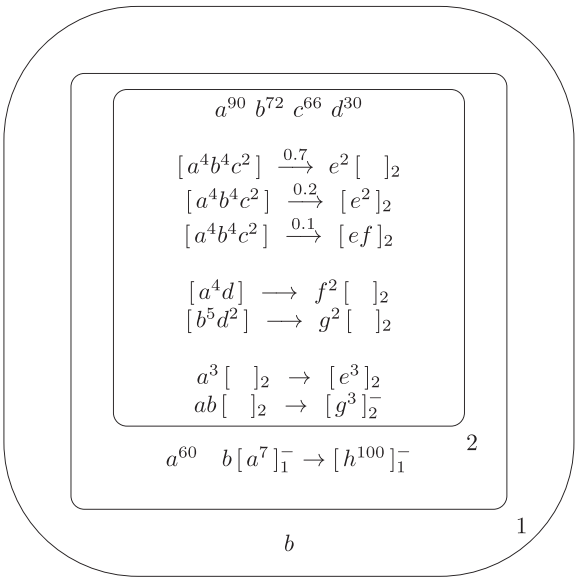

Fig. 5. PDP system.

- Execute `plingua dcba.pli -f bin -o dcba.bin`. This generates the Binary file dcba.bin. The model file for PDP systems must be placed along with the input file dcba.pli.
- Execute `abcdgpu -f l -i dcba.bin -s 5 -t l -a l -v 5`. This launches the external simulator for PDP systems on the CPU with the largest verbosity level to check the execution of rules (we are interested only on the distribution of rule selection). We also launch 5 simulations (*-s 5*), give 1 to A parameter of DCBA (*-a 1*), simulate just one step (*-t 1*), and indicate that we want to read P-Lingua 5 Binary file (*-f 1 -i dcba.bin*).

References

[1] Gh. Păun, A quick introduction to membrane computing, The Journal of Logic and Algebraic Programming 79 (6) (2010) 291–294, https://doi.org/10.1016/j.jlap.2010.04.002.
[2] Gh. Păun, Computing with Membranes, Journal of Computer and System Sciences 61 (1) (2000) 108–143, https://doi.org/10.1006/jcss.1999.1693.
[3] Gh. Păun, Membrane Computing: An Introduction, Springer-Verlag, Berlin, Heidelberg, 2002.
[4] Gh. Păun, G. Rozenberg, A. Salomaa, The Oxford Handbook of Membrane Computing, Oxford University Press Inc, New York, NY, USA, 2010.
[5] M.J. Pérez-Jiménez, A. Riscos-Núñez, L. Valencia-Cabrera, D. Orellana-Martín, Results on Computational Complexity in Bio-inspired Computing, World Scientific, Ch. 2 (2019) 33–73, https://doi.org/10.1142/9789813143180_0002.
[6] G. Zhang, M. Pérez-Jiménez, M. Gheorghe, Real-life Applications with Membrane Computing, Vol. 25, Springer, 2017. doi:10.1007/978-3-319-55989-6..
[7] D. Orellana-Martín, L. Valencia-Cabrera, A. Riscos-Núñez, M.J. Pérez-Jiménez, A path to computational efficiency through membrane computing, Theoretical Computer Science 777 (2019) 443–453, https://doi.org/10.1016/j.tcs.2018.12.024.
[8] A. Leporati, L. Manzoni, G. Mauri, A.E. Porreca, C. Zandron, A survey on space complexity of P systems with active membranes, International Journal of Advances in Engineering Sciences and Applied Mathematics 10 (3) (2018) 221–229, https://doi.org/10.1007/s12572-018-0227-8.
[9] L. Valencia-Cabrera, D. Orellana-Martín, M.A. Martínez-del-Amor, I. Pérez-Hurtado, M. Pérez-Jiménez, From NP-completeness to DP-completeness: A Membrane Computing perspective, Complexity 2020 (2020) 1–10, https://doi.org/10.1155/2020/6765097.
[10] Gh. Păun, F.J. Romero-Campero, Membrane Computing as a modeling framework. Cellular systems case studies, in: M. Bernardo, P. Degano, G. Zavattaro (Eds.), Formal Methods for Computational Systems Biology, Lecture Notes in Computer Science, Vol. 5016, Springer, Berlin Heidelberg, 2008, pp. 168–214, https://doi.org/10.1007/978-3-540-68894-5_6.
[11] M. Gheorghe, N. Krasnogor, M. Camara, P systems applications to systems biology, Biosystems 91 (3) (2008) 435–437, https://doi.org/10.1016/j.biosystems.2007.07.002.
[12] R. Barbuti, P. Bove, P. Milazzo, G. Pardini, Minimal probabilistic P systems for modelling ecological systems, Theoretical Computer Science 608 (2015) 36–56, https://doi.org/10.1016/j.tcs.2015.07.035.
[13] M.A. Colomer, A. Margalida, M.J. Pérez-Jiménez, Population Dynamics P System (PDP) Models: A Standardized Protocol for Describing and Applying Novel Bio-Inspired Computing Tools, PLOS ONE 8 (5) (2013), https://doi.org/10.1371/journal.pone.0060698 e60698.
[14] Gh. Păun, R. Păun, Membrane Computing and Economics: Numerical P Systems, Fundamenta Informaticae 73 (1,2) (2006) 213–227.
[15] H. Peng, B. Li, J. Wang, X. Song, T. Wang, L. Valencia-Cabrera, I. Pérez-Hurtado, A. Riscos-Núñez, M.J. Pérez-Jiménez, Spiking neural P systems with inhibitory rules, Knowledge-Based Systems 188 (2020), https://doi.org/10.1016/j.knosys.2019.105064 105064.
[16] D. Orellana-Martín, M.A. Martínez-del-Amor, L. Valencia-Cabrera, I. Pérez-Hurtado, A. Riscos-Núñez, M.J. Pérez-Jiménez, Dendrite P systems toolbox: Representation, algorithms and simulators, International Journal of Neural Systems 31 (01) (2021) 2050071, https://doi.org/10.1142/S0129065720500719.
[17] T. Wang, G. Zhang, J. Zhao, Z. He, J. Wang, M.J. Pérez-Jiménez, Fault diagnosis of electric power systems based on fuzzy reasoning spiking neural P systems, IEEE Transactions on Power Systems 30 (3) (2015) 1182–1194.
[18] A. Pavel, O. Arsene, C. Buiu, Enzymatic numerical P systems - a new class of membrane computing systems, in: 2010 IEEE Fifth International Conference on Bio-Inspired Computing: Theories and Applications (BIC-TA), 2010, pp. 1331–1336.
[19] A. Pavel, C. Buiu, Using enzymatic numerical P systems for modeling mobile robot controllers, Natural Computing 11 (3) (2012) 387–393, https://doi.org/10.1007/s11047-011-9286-5.

[20] A. Florea, C. Buiu, Membrane computing for distributed control of robotic swarms: Emerging research and opportunities, IGI Global (2017), https://doi.org/10.4018/978-1-5225-2280-5.
[21] I. Pérez-Hurtado, M.J. Pérez-Jiménez, G. Zhang, D. Orellana-Martín, Simulation of Rapidly-Exploring Random Trees in Membrane Computing with P-Lingua and Automatic Programming, International Journal of Computers, Communications and Control 13 (6) (2019) 1007–1031.
[22] I. Pérez-Hurtado, M. Martínez-del-Amor, G. Zhang, F. Neri, M. Pérez-Jiménez, A membrane parallel rapidly-exploring random tree algorithm for robotic motion planning, Integrated Computer-Aided Engineering 27 (2020) 1–18, https://doi.org/10.3233/ICA-190616.
[23] G. Zhang, Z. Shang, S. Verlan, M.A. Martínez-del Amor, C. Yuan, L. Valencia-Cabrera, M.J. Pérez-Jiménez, An overview of hardware implementation of membrane computing models, ACM Comput. Surv. 53 (4). doi:10.1145/3402456..
[24] L. Valencia-Cabrera, I. Pérez-Hurtado, M.A. Martínez-del-Amor, Simulation challenges in membrane computing, Journal of Membrane Computing 2 (2020) 1–11, https://doi.org/10.1007/s41965-020-00056-w.
[25] L. Valencia-Cabrera, D. Orellana-Martín, M.A. Martínez-del-Amor, M.J. Pérez-Jiménez, An interactive timeline of simulators in membrane computing, Journal of Membrane Computing 1 (3) (2019) 209–222, https://doi.org/10.1007/s41965-019-00016-z.
[26] D. Díaz-Pernil, C. Graciani-Díaz, M.A. Gutiérrez-Naranjo, I. Pérez-Hurtado, M.J. Pérez-Jiménez, Software for P systems, Ch. 17, Oxford University Press, 2010, pp. 437–454.
[27] D. Cascado-Caballero, F. Díaz-del-Rio, D. Cagigas-Muñiz, A. Rios-Navarro, J. Guisado-Lizar, I. Pérez-Hurtado, A. Riscos-Núñez, MAREX: A generic hardware architecture for membrane computing, Information Sciences 584 (2022) 360–386.
[28] M.A. Martínez-del-Amor, M. García-Quismondo, L.F. Macías-Ramos, L. Valencia-Cabrera, A. Riscos-Núñez, M.J. Pérez-Jiménez, Simulating P systems on GPU devices: a survey, Fundamenta Informaticae 136 (3) (2015) 269–284, https://doi.org/10.3233/FI-2015-1157.
[29] M.Á. Martínez-del-Amor, D. Orellana-Martín, I. Pérez-Hurtado, L. Valencia-Cabrera, A. Riscos-Núñez, M.J. Pérez-Jiménez, Design of Specific P Systems Simulators on GPUs, in: T. Hinze, G. Rozenberg, A. Salomaa, C. Zandron (Eds.), Membrane Computing, Lecture Notes in Computer Science, Vol. 11399, Springer International Publishing, 2019, pp. 202–207, https://doi.org/10.1007/978-3-030-12797-8_14.
[30] M.A. Martínez-del-Amor, I. Pérez-Hurtado, D. Orellana-Martín, M.J. Pérez-Jiménez, Adaptative parallel simulators for bioinspired computing models, Future Generation Computer Systems 107 (2020) 469–484, https://doi.org/10.1016/j.future.2020.02.012.
[31] J.M. Cecilia, J.M. García, G.D. Guerrero, M.A. Martínez-del-Amor, I. Pérez-Hurtado, M.J. Pérez-Jiménez, Simulation of P systems with active membranes on CUDA, Briefings in Bioinformatics 11 (3) (2010) 313–322, https://doi.org/10.1093/bib/bbp064.
[32] I. Pérez-Hurtado, D. Orellana-Martín, M.A. Martínez-del-Amor, L. Valencia-Cabrera, A. Riscos-Núñez, M.J. Pérez-Jiménez, 11 years of P-Lingua: A backward glance, in: Pre-proceedings of the 20th International Conference on Membrane Computing (CMC 2019), Curtea del Arges, Romania, 2019, pp. 451–462..
[33] I. Pérez-Hurtado, D. Orellana-Martín, G. Zhang, M.J. Pérez-Jiménez, P-Lingua in two steps: flexibility and efficiency, Journal of Membrane Computing 1 (2) (2019) 93–102, https://doi.org/10.1007/s41965-019-00014-1.
[34] I. Pérez-Hurtado, L. Valencia-Cabrera, J.M. Chacón, A. Riscos-Núñez, M.J. Pérez-Jiménez, A P-Lingua based simulator for tissue P systems with cell separation, Romanian, Journal of Information Science and Technology 17 (2014) 89–102.
[35] M.A. Martínez-del-Amor, I. Pérez-Hurtado, M.J. Pérez-Jiménez, A. Riscos-Núñez, A P-Lingua based simulator for tissue P systems, Journal of Logic and Algebraic Programming 79 (2010) 374–382. doi:dx.doi.org/10.1016/j.jlap.2010.03.009..
[36] M. García-Quismondo, R. Gutiérrez-Escudero, M.A. Martínez-del-Amor, E.F. Orejuela-Pinedo, I. Pérez-Hurtado, P-Lingua 2.0: A software framework for cell-like P systems, International Journal of Computers, Communications and Control IV (2009) 234–243. url:http://www.journal.univagora.ro/?page=article_details&id=368.
[37] D. Díaz-Pernil, I. Pérez-Hurtado, M.J. Pérez-Jiménez, A. Riscos-Núñez, A P-Lingua programming environment for membrane computing, Lecture Notes in Computer Science 5391 (2009) 187–203, https://doi.org/10.1007/978-3-540-95885-7_14.
[38] P-Lingua website, url:https://www.p-lingua.org/, accessed: 2021-02-12..
[39] I. Pérez-Hurtado, L. Valencia-Cabrera, M.J. Pérez-Jiménez, M.A. Colomer, A. Riscos-Núñez, MeCoSim: A general purpose software tool for simulating biological phenomena by means of P systems, IEEE Fifth International Conference on Bio-inpired Computing: Theories and Applications (BIC-TA 2010) I (2010) 637–643. doi:https://doi.org/10.1109/BICTA.2010.5645199..
[40] MeCoSim project website, url:http://www.p-lingua.org/mecosim/, accessed: 2021-02-12..
[41] PMCGPU project website, url:https://sourceforge.net/projects/pmcgpu/, accessed: 2020-09-29..
[42] R. Freund, S. Verlan, A Formal Framework for Static (Tissue) P Systems, in: G. Eleftherakis, P. Kefalas, G. Păun, G. Rozenberg, A. Salomaa (Eds.), Membrane Computing, Springer, Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 271–284.
[43] R. Freund, I. Pérez-Hurtado, A. Riscos-Núñez, S. Verlan, A formalization of membrane systems with dynamically evolving structures, Int. J. Comput. Math. 90 (4) (2013) 801–815, https://doi.org/10.1080/00207160.2012.748899.
[44] M.A. Martínez-del-Amor, I. Pérez-Hurtado, M. García-Quismondo, L.F. Macías-Ramos, L. Valencia-Cabrera, Á. Romero-Jiménez, C. Graciani-Díaz, A. Riscos-Núñez, M. À. Colomer, M.J. Pérez-Jiménez, DCBA: Simulating population dynamics P systems with proportional object distribution, Vol. 2 of Proceedings of the Tenth Brainstorming Week on Membrane Computing, Sevilla, Spain, 2012, pp. 27–56. url:http://hdl.handle.net/11441/34064..
[45] P-Lingua 5 website, url:https://github.com/RGNC/plingua/, accessed: 2021-09-13..
[46] P. Guo, C. Quan, L. Ye, UPSimulator: A general P system simulator, Knowledge-Based Systems 170 (2019) 20–25, https://doi.org/10.1016/j.knosys.2019.01.013.
[47] G. Zhang, H. Rong, P. Paul, Y. He, F. Neri, M.J. Pérez-Jiménez, A complete arithmetic calculator constructed from spiking neural P systems and its application to information fusion, International Journal of Neural Systems 31 (01) (2020) 2050055, https://doi.org/10.1142/s0129065720500550.
[48] M. Zhu, Q. Yang, J. Dong, G. Zhang, X. Gou, H. Rong, P. Paul, F. Neri, An adaptive optimization spiking neural P system for binary problems, International Journal of Neural Systems 31 (01) (2020) 2050054, https://doi.org/10.1142/s0129065720500549.
[49] L. Pan, G. Păun, G. Zhang, F. Neri, Spiking neural P systems with communication on request, International Journal of Neural Systems 27 (08) (2017) 1750042, https://doi.org/10.1142/s0129065717500423.
[50] T. Wu, F.-D. Bılbıe, A. Păun, L. Pan, F. Neri, Simplified and yet turing universal spiking neural P systems with communication on request, International Journal of Neural Systems 28 (08) (2018) 1850013, https://doi.org/10.1142/s0129065718500132.