P-Lingua in two steps: flexibility and efficiency

Ignacio Pérez-Hurtado[1] · David Orellana-Martín[1] · Gexiang Zhang[2] · Mario J. Pérez-Jiménez[1]

Abstract
Membrane computing is a bio-inspired computing paradigm that lacks in vivo implementation. That is why software or hardware implementations have to be used to validate models. Several tools have been created for this purpose; some of them are created for specific purposes, such as solving a computationally hard problem; and others are more generic, to cover a broad spectrum of possible models. The former have the advantage of being very efficient, crucial for solving large instances of certain problems; however, this efficiency leads to a loss of generality, since algorithms are usually hard-coded and they do not allow other models. On the contrary, the latter are perfect tools for researchers, given that new models can be checked without much effort by defining them in the framework; since these algorithms have to simulate as many models as possible, they lack specificities to improve the performance. P-Lingua has been widely used to simulate membrane systems, having integrated both a language and a simulator. To obtain better results in terms of time used to simulate models defined in this language, a new perspective is studied. The model defined in P-Lingua will be compiled into C++ source code that will implement an ad hoc simulator. This code will consider specifications about how rules have to be executed, that is, some simple specifications of the semantics. To show how it works, some examples of specifications of models will be presented, which can be simulated using the new-developed GNU GPLv3 command-line tool *pcc*.

Keywords P-Lingua · Membrane computing · Simulators

1 Introduction

Membrane computing is an unconventional model of computation within natural computing that was introduced in 1998 by Păun [16]. The computational devices in membrane computing, also known as membrane systems or P systems, are non-deterministic theoretical machines inspired on the biochemical processes that take place inside the compartments of living cells.

Among the different types of membrane systems, two main families are studied: cell-like membrane systems, characterized by their rooted tree structure, where membranes act as filters that let certain elements to pass through them [16], and membrane systems structured as directed graphs, representing the communication between cells within a tissue of a living being, called tissue-like P systems [9] or between neurons in a brain, called spiking neural P systems [7]. The evolution of these systems is directed by a set of predefined *rewriting rules*, in such a way that given a multiset of objects present in a given *compartment*, they can evolve and/or be transported to another compartment. The applicability of the rules is given by the corresponding semantics of the system.

A *configuration* of a P system is defined by the structure of the compartments at a certain moment, and the elements (being usually objects, although other kinds of elements can be considered, as strings, catalysts [16] and anti-matter [14], among others) contained in each compartment, as well as other characteristics from specific types of P systems, providing a *snapshot* of the system at an instant t.

✉ Ignacio Pérez-Hurtado
perezh@us.es

David Orellana-Martín
dorellana@us.es

Gexiang Zhang
zhgxdylan@126.com

Mario J. Pérez-Jiménez
marper@us.es

1 Research Group on Natural Computing, Department of Computer Science and Artificial Intelligence, University of Seville, Seville, Spain

2 School of Electrical Engineering, Southwest Jiaotong University, Chengdu, Sichuan, China

On the one hand, in P systems with active membranes [18], both objects and membranes change through the application of evolution, communication, division, separation, creation and dissolution. In this framework, membranes can have a polarization associated with each membrane. On the other hand, in tissue P systems [9], symport/antiport rules are devoted to make objects move from a cell to another cell or to the *environment* (a special compartment where there exist an arbitrary number of objects of an alphabet defined a priori), while division, separation and creation rules allow an exponential growth in linear time.

We say that a configuration C_t yields to a configuration C_{t+1} if, by applying the rules specified in the model according to its semantics, we can obtain C_{t+1} from C_t. A *computation* of a P system is a (finite or infinite) sequence of instantaneous configurations.

We consider a *family* (or *model*) of P systems as the definition of a type of P systems, that is, its syntax and semantics. According to the specification of a particular family of P systems, we consider the definition of an individual P system, that is, its working alphabet, initial membrane structure with initial multisets of objects and the set of rewriting rules with other characteristics, such as special alphabets or the number of environments, of the corresponding family. By the definition of the family, we can interpret the structure and behavior of a specific model within that family.

Membrane computing is a very flexible framework where different types of devices can be outlined. In fact, the intersection between membrane computing and other fields, such as engineering [19], biology [22] and ecology [2], as well as a long list of other scientific lines [5, 13, 23], has generated necessities that could only be filled by the creation of new kinds of P systems, expanding the scope of researchers in this area. For an exhaustive explanation of the different types of P systems, we refer the reader to [15, 17].

Software and hardware simulators have been implemented from the beginnings of membrane computing. As generic simulators are necessary to simulate several types of P systems, *ad hoc* simulators are crucial as they can be optimized for the input design and the hardware to be used. But the hard-coding process requires an excellent knowledge of the hardware architecture, as well as the design to be implemented. Debugging should be always critical and the results are not very reusable.

In this work, we have extended the P-Lingua framework [6, 24] to include semantic features specific to the models. For this purpose, we have implemented a GNU GPLv3 command-line tool to compile P-Lingua input files to ad hoc source code in C++. The output files are optimized for the input designs and the entire process can be automated using *makefiles*, i.e.,files which specify how to derive the target program.

The paper is structured as follows: In the next section, some preliminary concepts about P-Lingua are introduced. In Sect. 3, we propose an extension for the P-Lingua language to directly define model constraints in P-Lingua files, providing a more flexible and experimental framework. The next Section is devoted to the new GNU GPLv3 software tool to compile the input P-Lingua files and generate source code in C++, as well as JSON code codifying the input designs for third-party applications. Section 5 introduces the simulation algorithm used in the generated simulators. In Sect. 6 some examples of the new P-Lingua extension are introduced. Finally, some conclusions and future work are drawn.

2 Preliminaries

P-Lingua [6, 24] is a software framework that includes a definition language for P systems (also called P-Lingua) and a GNU GPLv3 Java library (pLinguaCore) that is able to parse P-Lingua files and simulate computations. The library contains three main components:

- A parser for reading input files in P-Lingua format and checking syntactic and semantic constraints related to predefined models. To achieve this, the first line of a P-Lingua file should include a P system model declaration using an unique identifier. There are several P system models that can be used, each one with its own identifier, such as `transition`, `membrane_division`, `tissue_psystems`, and `probabilistic`. The analysis of semantic ingredients, such as rule patterns, is hard-coded for each model. Several versions of pLinguaCore [6, 8, 10, 20] have been launched to cover different types of models.
- For each type of model, the pLinguaCore library includes one or more built-in simulators, each one implementing a different simulation algorithm. For instance, Population Dynamic P systems [1] (`probabilistic` identifier in P-Lingua) can be simulated within the library by applying three different algorithms: `BBB`, `DNDP`, and `DCBA` [3, 11]. Software projects such as MeCoSim (Membrane Computing Simulator) [21, 26] use the simulators integrated into the library to perform P system computations and generate relevant information as a result for custom applications.
- Alternatively, the pLinguaCore library is able to transform the input P-Lingua files to other formats such as XML or binary format to feed external simulators. The generated files for the given P systems are free of syntactic/semantic errors since the transformation is done after the parser analysis. Several external simulators use this feature, for example, the PMCGPU project (Parallel

simulators for membrane computing on GPU) [12, 25] uses definitions generated by pLinguaCore to provide the input of CUDA GPU simulators.

The P-Lingua language is currently a standard widely used for the scientific community since the syntax is modular, parametric and close to the common scientific notation. The description of the language can be found in [6, 8, 10, 20, 24]. For example, the definition of a basic transition P system follows:

```
@model<transition>
def main()
{
  @mu = [[[]’3 []’4]’2]’1;
  @ms(3) = a,f;

  [a --> a,bp]’3;
  [a --> bp,@d]’3;
  [f --> f*2]’3;

  [bp --> b]’2;
  [b []’4 --> b [c]’4]’2;
  (1) [f*2 --> f ]’2;
  (2) [f --> a,@d]’2;
}
```

In the example, a module `main` is defined including an initial membrane structure $[\,[\,[\ ]_3\ [\ ]_4\,]_2\,]_1$, an initial multiset for the membrane labeled 3, and a set of seven multiset rewriting rules. The special symbol `@d` is used to specify dissolution. The last two rules include priorities as integer numbers in parenthesis at the beginning of the left-hand side of the rules. More complex examples can be found in the P-Lingua web [24].

3 An extension of P-Lingua for semantic features

As explained above, the analysis of semantic ingredients belonging to P systems is hard-coded in the pLinguaCore library, i.e., the only way to define new types of models is by implementing code inside the library. In this section, we propose an extension to the P-Lingua language to directly define model constraints in their own P-Lingua files, providing a more flexible and experimental framework. Two types of semantic constraints can be defined with this extension: *rule patterns* and *derivation modes*.

3.1 Rule patterns

The P-Lingua parser is able to recognize rules with the next general syntax:

$$\overset{p}{u[v_1[v_{1,1}]^{\alpha_{1,1}}_{h_{1,1}}\dots[v_{1,m_1}]^{\alpha_{1,m_1}}_{h_{1,m_1}}]^{\alpha_1}_{h_1}\dots[v_n[v_{n,1}]^{\alpha_{n,1}}_{h_{n,1}}\dots[v_{n,m_n}]^{\alpha_{n,m_n}}_{h_{n,m_n}}]^{\alpha_n}_{h_n}}$$
$$\xrightarrow{q} \text{ or } \overset{q}{\longleftrightarrow}$$
$$w_0[w_1[w_{1,1}]^{\beta_{1,1}}_{g_{1,1}}\dots[w_{1,r_1}]^{\beta_{1,r_1}}_{g_{1,r_1}}]^{\beta_1}_{g_1}\dots[w_s[w_{s,1}]^{\beta_{s,1}}_{g_{s,1}}\dots[w_{s,r_s}]^{\beta_{s,r_s}}_{g_{s,r_s}}]^{\beta_s}_{g_s},$$

where

- p is a priority related to the rule given by a natural number, where a lower number means a higher rule priority.
- q is a probability related to the rule given by a real number in [0, 1].
- $\alpha_i, \alpha_{i,j}, 1 \le i \le n, 1 \le j \le m_i$ and $\beta_i, \beta_{i,j}, 1 \le i \le s, 1 \le j \le r_i$ are electrical charges.
- $h_i, h_{i,j}, 1 \le i \le n, 1 \le j \le m_i$ and $g_i, g_{i,j}, 1 \le i \le s, 1 \le j \le r_i$ are membrane labels.
- $u, v_i, v_{i,j}, 1 \le i \le n, 1 \le j \le m_i$ and $w_i, w_{i,j}, 1 \le i \le s, 1 \le j \le r_i$ are multisets of objects.

Next, there is a list of P-Lingua rule examples matching the general rule syntax:

- `a,b [ d,e*2 ]’h --> [f,g]’h : : q;` where `q` is the probability of the rule.
- `(p) [a]’h --> [b]’h;` where `p` is the priority of the rule.
- `[a --> b]’h;`, the left-hand side and right-hand side of evolution rules can be collapsed.
- `+[a]’h --> +[b]’h -[c]’h;` a division rule using electrical charges.
- `[a]’h --> ;` a dissolution rule.
- `a[ ]’h --> [b]’h;` a send-in rule.
- `[a]’h --> b[ ]’h;` a send-out rule.
- `[a --> #]’h;` the symbol # is usually used as the empty multiset.
- `[a]’1 <--> [b]’0;` a symport/antiport rule in the tissue-like framework.

The syntax of the general rule is very permissive, and so different parsers for different models have been developed to restrict the rules used in each one. To provide the researcher a more flexible framework, not having to depend on the implementation itself but acquiring the capacity of restricting the model by himself, we introduce the next syntax in P-Lingua for rule pattern matching:

```
!rule-type-id
{
pattern1
pattern2
...
patternN
}
```

where `rule-type-identifier` is an unique name for the type of rule that is going to be defined and `pattern1`, `pattern2`, ..., `patternN` are rule patterns following the same syntax as common rules in P-Lingua, where anonymous variables beginning with `?` can be optionally used instead of probabilities, charges and priorities. In the patterns, the symbols beginning with `a`, `b` or `c` always mean single objects and symbols beginning with `u`, `v` and `w` always mean multisets of objects. In Sect. 6, several examples of rule patterns in P-Lingua for different types of cell-like and tissue-like models are given.

3.2 Derivation modes

From an informal point of view, we can see a derivation mode as the way a step of a P system is performed. Semantics complement syntax in such a way that they provide the functioning of the system. Derivation modes control the number of rules of each type that can be executed in each transition step. An extensive study of derivation modes can be found in [4]. To make this work self-contained, we give a minimal definition of a derivation mode.

A derivation mode ϑ is defined as a function that selects different multisets of rules "really applicable" to a configuration C_t of a P system depending on a specification. For this purpose, let Π be a P system with $\mathcal{R}$ as its set of rules, R a multiset of compatible rules applicable to a P system at configuration C_t, and let $\mathbf{R} = P(\mathcal{R})$ be the set of all multisets applicable to a P system at configuration C_t.

In this extension of P-Lingua, we provide two main derivation modes:

- *Maximally parallel derivation mode* (max) It is the default mode for P systems. In this mode, we only take multisets from R that are not extensible, that is:

  $$\mathbf{R}' = \{R \mid R \in \mathbf{R} \wedge \nexists R' \in \mathbf{R} : R \subsetneq R'\}.$$

  The multiset of rules finally applied to C_t is selected non-deterministically from $\mathbf{R}'$.
- *Bounded-by-rule maximally parallel derivation mode* ($bound_{B_1,\dots,B_r}$) Let $\{a, b, \dots\}$ be the set of different types of rules present in a P system, $\mathcal{R}_j$ be the set of rules applicable to the j-th compartment of the system [1] in the configuration C_t, where there are m membranes, $\mathcal{R}^{(k)}$ be the set of rules of the type k, being $k \in \{a, b, \dots\}$ and $\mathcal{R}_j^{(k)}$ the set of rules of type k applicable to the j-th compartment. B_i can be of the following forms:

  - $B_i = j, j \in \{a, b, \dots\}$;
  - $B_i = \beta_n(B_{1_i}, \dots, B_{r_i})$, being $n \in \mathbb{N}$, and for each $B_j = \beta_{m_j}(B_{1_j}, \dots, B_{r_j})$, $j \in \{1_i, \dots r_i\}$, $m_j \leq n$;
  - As a restriction, a type of rule cannot appear more than once in the whole definition of the derivation mode.

  We say that n is the *bound* of $B_i = \beta_n$. We say that a type of rule (j) is in the *context* of B_i if:

  - There exists $B_i = \beta_n(j)$ (we call B_i its *immediate context*); and
  - There exists $B_i = \beta_n(B_{1_i}, \dots, B_{r_i})$ such that B_j is a context of the type of rule (j).

  This mode is defined *recursively*, and we can understand the *applicability* of the rules in a defined bounded-by-rule parallel derivation mode in the following sense:

  - The total number of rules within a *context* $\beta_n(B_1, \dots, B_r)$, that can be applied in parallel in a P system in a configuration C_t is n; and
  - In a bounded-by-rule parallel derivation mode $bound_{B_1,\dots,B_r}$, if $B_i = j (j \in \{a, b, \dots\})$, being $1 \leq i \leq r$, then rules of type j can be applied in a maximal parallel way.

  With this mode, we can define the classical mode used in P systems with active membranes, that is, evolution rules (a) can be applied in a maximal parallel mode, while the other types of rules (send-in communication rules (b), send-out communication rules (c), dissolution rules (d), division rules for elementary (e) and non-elementary (f) membranes) can be applied at most once per membrane at each computation step. It would be defined as $bound_{a,\beta_1(b,c,d,e,f)}$. Then, the formal definition of the bounded-by-rule maximally parallel mode is the following:

$$\begin{aligned}\mathbf{R}' = \{\ & R \mid R \in \mathbf{R} \wedge \forall j \in \{1, \dots, m\} \\ & |\{r \mid r \in R, r \in \mathcal{R}_j^{(k)}, \text{ for all } k \text{ in the context of } B_i = \beta_n\}| \leq n \\ & \wedge \nexists R' \in \mathbf{R} : R \subsetneq R'\}\end{aligned}$$

Thus, a model type can be defined in P-Lingua by aggregating the allowed rule patterns and its corresponding derivation modes; the syntax is as follows:

[1] It is important to remark that some membrane systems have a dynamic structure, so j does not have to match with the label of the membrane.

```
@model(id) = rule-type-id1,..., rule-type-idN;
```

where `id` is an unique identifier for the model and `rule-type-id1` ,...,
`rule-type-idN` are unique identifiers for the corresponding allowed rule patterns. By default, all rules behave in maximally parallel derivation mode, but rules can be grouped in sets to behave in bounded parallel derivation mode as follows:

```
@model(id) = @bound{rule-type-id1,..., rule-type-idN};
```

where `bound` is a natural number defining the maximum number of rules in the group that can be applied to a given configuration. In Sect. 6, several examples of model definitions in P-Lingua are given.

4 A command-line tool for generating ad hoc simulators

A GNU GPLv3 command-line tool called **pcc** has been implemented in C++ language with Flex [27] and Bison [28]. The source code including examples and instructions can be downloaded from https://github.com/RGNC/plingua.

The tool provides three main functionalities:

- *Parsing P-Lingua files* While printing the syntactic and semantic errors to the standard error output. In this sense, the tool acts as a conventional compiler, showing the name of the file, as well as the number of the line and column for each error with a short description. The analysis of semantic errors is done using the rule patterns and derivation modes defined in the own P-Lingua files. Several files can be compiled together like conventional programs; furthermore, standard *makefiles* can be also used. The developer can decide to write the rule patterns and derivation modes in a set of files and reuse them in several projects. More explanations can be found in the website.
- *Generating JSON files* The tool is able to translate the definitions contained in P-Lingua files to JSON format [29] for compatibility with third-party simulators. The translation is done after parsing the input files; thus, the JSON files are free of syntactic/semantic errors and the third-party applications do not have to check them. Several P-Lingua files can be combined together in one JSON file, including also the selected derivation modes.
- *Generating source code* The tool can generate all the source files for a command-line executable in C++ which is a complete ad hoc simulator optimized for the design given by the input files. The generated program is able to simulate computations for the defined P system following the specified derivation modes. It interacts with the user by the command-line as common Linux console applications. Generic front-ends could be easily implemented because the command-line options are common to all the simulators. The simulations could be interrupted and resumed, since intermediate configurations can be saved in JSON files. Initial multisets can also be defined before the simulation, as well as setting different halting conditions, such as simulating a fixed number of computation steps or running until the execution of a rule marked in the P-Lingua file as halting rule.

The *pcc* tool performs several analyses over the input files to optimize the memory and time that is going to be used for the simulator. The C++ structures used to represent the membrane tree are selected depending on the type of rules that can be used. For instance, if there are no send-in/send-out rules, then C++ pointers to parent/child membranes are not necessary. The generated code can be compiled with the GNU g++ tool [30] *makefiles* can also be used to automate all the process from the P-Lingua files to the Linux executable. Instructions and examples can be found in the web page.

5 The simulation algorithm

The compiler presented in Sect. 4 generates the source code in C++ for an ad hoc simulator which is able to reproduce computations for the input design written in P-Lingua. The generated code follows the scheme shown in Fig. 1. The simulation is provided by a sequential loop where each iteration simulates one step of computation. For each iteration, the simulator determines the multiset of rules which is going to be applied and then, it applies it to the current configuration C_t obtaining the next configuration C_{t+1}. The halting condition is checked after each iteration.

The algorithm used to select rules is described in Pseudocode 1. It returns a multiset B of pairs (m, r) and a configuration C'_t. One pair (m, r) means that rule r has been selected once to be applied over membrane m in C_t. The configuration C'_t contains a copy of C_t after applying the left-hand side of the selected rules, i.e, after removing from C_t the multisets of objects specified by the left-hand side of the selected rules. On the other hand, the applicability function determines the maximum number of possible applications for a rule r over a membrane m in configuration C'_t. It

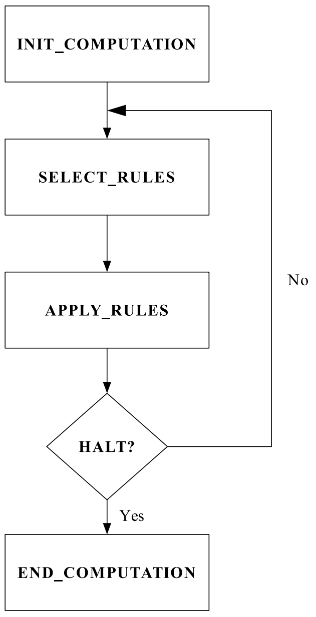

Fig. 1 The main simulation loop

considers the left-hand side, the charges in the right-hand side, as well as the derivation mode of r. A membrane m in C'_t is marked as *fixed* if at least one pair (m, r) is contained in B or *unfixed* otherwise. A rule r cannot be selected if it would change the electrical charge of a *fixed* membrane.

Finally, Algorithm 2 receives the partial configuration C'_t and generates the next configuration C_{t+1} by applying the right-hand side of the selected rules.

6 Examples

6.1 Transition P systems

```
!transition_evolution /* Limited to rules with 3 inner membranes */
{
        [a -> v]'h;
        [a -> v, @d]'h;
    (?) [a -> v]'h;
    (?) [a -> v, @d]'h;
        [a [ ]'h1 --> v [w]'h1]'h;
        [a [ ]'h1 --> v [w]'h1]'h;
    (?) [a [ ]'h1 --> v [w]'h1]'h;
    (?) [a [ ]'h1 --> v [w]'h1]'h;
        [a [ ]'h1 [ ]'h2 --> v [w1]'h1 [w2]'h2]'h;
        [a [ ]'h1 [ ]'h2 --> v [w1]'h1 [w2]'h2]'h;
    (?) [a [ ]'h1 [ ]'h2 --> v [w1]'h1 [w2]'h2]'h;
    (?) [a [ ]'h1 [ ]'h2 --> v [w1]'h1 [w2]'h2]'h;
        [a [ ]'h1 [ ]'h2 [ ]'h3 --> v [w1]'h1 [w2]'h2 [w3]'h3]'h;
        [a [ ]'h1 [ ]'h2 [ ]'h3 --> v [w1]'h1 [w2]'h2 [w3]'h3]'h;
    (?) [a [ ]'h1 [ ]'h2 [ ]'h3 --> v [w1]'h1 [w2]'h2 [w3]'h3]'h;
    (?) [a [ ]'h1 [ ]'h2 [ ]'h3 --> v [w1]'h1 [w2]'h2 [w3]'h3]'h;
}

@model(transition) = transition_evolution;
```

Algorithm 1 SELECT_RULES

```
Require: Current configuration $C_t$; Set of rules $R$;
  $M \leftarrow \emptyset; B \leftarrow \emptyset; C'_t \leftarrow C_t$;
  for each membrane $m$ in $C'_t$ do
    $A_m \leftarrow R$; // Copy the set of rules
    $M \leftarrow M \cup m$
    Mark m as $unfixed$
  end for
  while $|M| > 0$ do
    $m \leftarrow$ Randomly select one membrane in $M$
    $r \leftarrow$ Randomly select one rule in $A_m$
    $k \leftarrow applicability(C'_t, r, m)$
    if $k = 0$ then
      Remove $r$ from $A_m$
      if $|A_m| = 0$ then
        Remove $m$ from $M$
      end if
    else
      $n \leftarrow$ A random natural number in $[1, k]$
      $C'_t \leftarrow apply\_left\_hand\_side(C'_t, r, m, n)$
      $B \leftarrow B \cup \{(m, r)^n\}$
      Mark m as $fixed$
    end if
  end while
  return $(C'_t, B)$
```

Algorithm 2 APPLY_RULES

```
Require: Partial configuration $C'_t$; Multiset of selected rules $B$;
  $C_{t+1} \leftarrow C'_t$;
  for each pair $(m, r)$ in $B$ do
    $C_{t+1} \leftarrow apply\_right\_hand\_side(C_{t+1}, r, m)$
  end for
  return $C_{t+1}$
```

6.2 Active membranes with division rules

```
!dam_evolution
{
   ?[a -> v]'h;
   ?[a ->  ]'h;
}

!dam_send_in
{
   a ?[ ]'h -> ?[b]'h;
}

!dam_send_out
{
   ?[a]'h -> b ?[ ]'h;
}

!dam_dissolution
{
   ?[a]'h -> b;
   ?[a]'h ->  ;
}
!dam_division
{
   ?[a]'h -> ?[ ]'h ?[ ]'h;
   ?[a]'h -> ?[b]'h ?[ ]'h;
   ?[a]'h -> ?[ ]'h ?[b]'h;
   ?[a]'h -> ?[b]'h ?[c]'h;
}

@model(membrane_division) =
       dam_evolution, @1{dam_send_in, dam_send_out, dam_dissolution, dam_division};
```

6.3 Tissue-like P systems with communication and cell division

```
!tissue_communication
{
  [u]'h1 <--> [v]'h2;
}

!tissue_division
{
   [a]'h -> [ ]'h [ ]'h;
   [a]'h -> [b]'h [ ]'h;
   [a]'h -> [ ]'h [b]'h;
   [a]'h -> [b]'h [c]'h;
}

@model(tissue_division) =
       tissue_communication, @1{tissue_division};
```

6.4 Population dynamics P systems

```
!pdp_evolution
{
   u1 ?[v1]'h -> u2 ?[v2]'h :: ?;
}

!pdp_environment_communication
{
   [[a]'e1 [ ]'e2]'h -> [[ ]'e1 [b]'e2]'h :: ?;
}

@model(probabilistic) =
       pdp_evolution, pdp_environment_communication;
```

7 Conclusions and future work

This paper presents for the first time a compiler for membrane computing which is able to generate C++ source code for optimized ad hoc simulators. The input P systems are written in P-Lingua, a common language to define membrane computing designs. In this paper, we have extended the language to include semantics ingredients, such as rule patterns and derivation modes. The compiler can recognize the rule patterns and show syntactic/semantic errors during the parsing process. The generated simulators are able to simulate computations given by the derivation modes, even if the derivation modes are experimental. Thus, the goal of this tool is twofold: On the one hand, it purports to be a good assistant for researchers while verifying their designs, even working with experimental models. On the other hand, it provides optimized simulators for real applications, such as robotics or simulation of biological phenomena.

Several lines are open for future work. From the point of view of the language, the semantic ingredients that can be written in P-Lingua should be studied to cover more types of models; for instance, defining bounds for the multiplicities of objects in different compartments, such as the environment in tissue-like P systems, where the multiplicity of objects can be infinite. On the other hand, custom directives could be included in P-Lingua files and translated to C preprocessor directives for the simulator. For example, if the design is confluent, a directive could be written to optimize the simulation time, since it is not necessary to simulate the non-determinism using random numbers.

From the point of view of the generated simulators, it would be very interesting to produce optimized code for different parallel hardware architectures such as multi-core processors, GPUs or FPGAs. Until now, the faster simulators for parallel architectures are relatively ad hoc, since several optimizations should be done by analyzing the input design. A tool able to automatize this process for a wide variety of input designs could approximate the membrane computing paradigm to other disciplines where efficient solutions to hard problems are needed. In particular, it could be applied to anytime algorithms for robotics, such as social navigation in crowdy environments or automatic driving, where the robot should have a fast response in real time, but the solution could be improved using more computational time.

Acknowledgements This work is supported by the research project TIN2017-89842-P, cofinanced by Ministerio de Economía, Industria y Competitividad (MINECO) of Spain, through the Agencia Estatal de Investigación (AEI), and by Fondo Europeo de Desarrollo Regional (FEDER) of the European Union. The authors also acknowledge the Grants No 61320106005 of the National Natural Science Foundation of China.

References

1. Colomer, M., Margalida, A., & Pérez-Jiménez, M. J. (2013). Population dynamics P system (PDP) models: A standardized protocol for describing and applying novel bio-inspired computing tools. *PLoS One*, *8*(14), 1–13.
2. Cardona, M., Colomer, M. A., Pérez-Jiménez, M. J., Sanuy, D., & Margalida, A. (2008). Modeling ecosystems using P systems: The bearded vulture, a case study. In *Membrane computing, 9th international workshop, WMC*. Edinburgh, UK, July 28–31, 2008, Revised selected and invited papers. Lecture notes in computer science (2009) (Vol. 5391, pp. 137–156).
3. Colomer, M., Pérez-Hurtado, I., Pérez Jiménez, M. J., & Riscos-Núñez, A. (2012). Comparing simulation algorithms for multienvironment probabilistic P system over a standard virtual ecosystem. *Natural Computing*, *11*, 369–379.
4. Freund, R., & Verlan, S. (2007). A formal framework for static (tissue) P systems. In G. Eleftherakis, P. Kefalas, G. Păun, G. Rozenberg, & A. Salomaa (Eds.), *Membrane Computing. WMC 2007. Lecture Notes in Computer Science,* (Vol. 4860, pp. 271–284). Berlin, Heidelberg: Springer. https://doi.org/10.1007/978-3-540-77312-2_17.
5. Frisco, P., Gheorghe, M., & Pérez-Jiménez, M. J. (2014). Applications of membrane computing in systems and synthetic biology. In *Emergence, complexity and computation* (series ISSN 2194-7287), Vol. 7. Berlin: Springer International Publishing. eBook ISBN 978-3-319-03191-0, Hardcover ISBN 978-3-319-03190-3. https://doi.org/10.1007/978-3-319-03191-0.
6. García-Quismondo, M., Gutiérrez-Escudero, R., Pérez-Hurtado, I., Pérez-Jiménez, M. J., & Riscos-Núñez, A. (2010). An overview of P-Lingua 2.0. In G. Păun, M. J. Pérez-Jiménez, A. Riscos-Núñez, G. Rozenberg, & A. Salomaa (Eds.) *Membrane Computing. WMC 2009. Lecture Notes in Computer Science*, (Vol. 5957, pp. 264–288). Berlin, Heidelberg: Springer. https://doi.org/10.1007/978-3-642-11467-0_20
7. Ionescu, M., Păun, Gh, & Yokomori, T. (2006). Spiking neural P systems. *Fundamenta Informaticae*, *71*(2–3), 279–308.
8. Macías, L. F., Pérez-Hurtado, I., García-Quismondo, M., Valencia, L., Pérez-Jiménez, M. J., & Riscos-Núñez, A. (2012). A P-Lingua based simulator for spiking neural P systems. In M. Gheorghe, G. Păun, G. Rozenberg, A. Salomaa, & S. Verlan (Eds.) *Membrane Computing Lecture notes in computer science, CMC 2011* (Vol. 7184, pp. 257–281). Berlin, Heidelberg: Springer. https://doi.org/10.1007/978-3-642-28024-5_18.
9. Martín-Vide, C., Păun, Gh, Pazos, J., & Rodríghez-Patón, A. (2003). Tissue P systems. *Theoretical Computer Science*, *296*(2), 295–326.
10. Martínez-del-Amor, M. A., Pérez-Hurtado, I., Pérez-Jiménez, M. J., & Riscos-Núñez, A. (2010). A P-Lingua based simulator for tissue P systems. *Journal of Logic and Algebraic Programming*, *79*(6), 374–382. https://doi.org/10.1016/j.jlap.2010.03.009
11. Martínez-del-Amor, M. A., Pérez-Hurtado, I., García-Quismondo, M., et al. (2013). DCBA: Simulating population dynamics P systems with proportional objects distribution. Lecture notes in computer science, Vol. 7762, pp. 257–276.
12. Martínez-del-Amor, M. A., García-Quismondo, M., Macías-Ramos, L. F., Valencia-Cabrera, L., Riscos-Núñez, A., & Pérez-Jiménez, M. J. (2015). Simulating P systems on GPU devices: A survey. *Fundamenta Informaticae*, *136*(3), 269–284.
13. Pan, L., Paun, Gh., Pérez-Jiménez, M. J., & Song, T. Bio-inspired computing: Theories and applications. Communications in computer and information science (series ISSN 1865-0929), Vol. 472. Berlin: Springer. Print ISBN 978-3-662-45048-2, Online ISBN 978-3-662-45049-9, 2014. https://doi.org/10.1007/978-3-662-45049-9.
14. Pan, L., & Păun, Gh. (2009). Spiking neural P systems with antimatter. *International Journal of Computers Communications & Control*, *4*(3), 273–282.
15. Păun, Gh, Rozenberg, G., & Salomaa, A. (Eds.). (2010). *The Oxford handbook of membrane computing*. Oxford: Oxford University Press.

16. Păun, Gh. (2000). Computing with membranes. *Journal of Computer and System Sciences*, *61*(1), 108–143 and Turku Center for CS-TUCS Report No. 208, 1998.
17. Păun, Gh. (2002). *Membrane computing. An introduction*. Berlin: Springer.
18. Păun, Gh. (2001). P systems with active membranes: Attacking NP-complete problems. *Journal of Automata, Languages and Combinatorics*, *6*, 75–90.
19. Peng, H., Wang, J., Ming, J., Shi, P., Pérez-Jiménez, M. J., Yu, W., & Tao, Ch. (2017). Fault diagnosis of power systems using intuitionistic fuzzy spiking neural P systems. *IEEE Transactions on Smart Grid*. https://doi.org/10.1109/TSG.2017.2670602**(in press)**.
20. Pérez-Hurtado, I., Valencia-Cabrera, L., Chacón, J. M., Riscos-Núñez, A., & Pérez-Jiménez, M. J. (2014). A P-Lingua based simulator for tissue P systems with cell separation. *Romanian Journal of Information Science and Technology*, *17*(1), 89–102.
21. Pérez-Hurtado, I., Valencia-Cabrera, L., Pérez-Jiménez, M. J., Colomer, M., & Riscos-Núñez, A. (2010). MeCoSim: A general purpose software tool for simulating biological phenomena by means of P Systems. In *IEEE fifth international conference on bio-inpired computing: Theories and applications (BIC-TA 2010)*, pp. 637–643.
22. Romero-Campero, F. J., & Pérez-Jiménez, M. J. (2008). A model of the quorum sensing system in Vibrio Fischeri using P systems. *Artificial Life*, *14*(1), 95–109. https://doi.org/10.1162/artl.2008.14.1.95.
23. Zhang, G., Pérez-Jiménez, M. J., & Gheorghe, M. (2017). Real-life applications with membrane computing. In Emergence, complexity and computation (series ISSN 2194-7287), Vol. 25. Berlin: Springer International Publishing. Online ISBN 978-3-319-55989-6, Print ISBN 978-3-319-55987-2. https://doi.org/10.1007/978-3-319-55989-6.
24. The P-Lingua web page: http://www.p-lingua.org. Accessed 15 Dec 2018.
25. The PMCGPU web page: https://sourceforge.net/projects/pmcgpu/. Accessed 15 Dec 2018.
26. The MeCoSim web page: http://www.p-lingua.org/mecosim/. Accessed 15 Dec 2018.
27. The Flex web page: https://github.com/westes/flexl. Accessed 15 Dec 2018.
28. The Bison web page: https://www.gnu.org/software/bison/. Accessed 15 Dec 2018.
29. The JSON web page: https://www.json.org/. Accessed 15 Dec 2018.
30. The GNU g++ compiler: https://gcc.gnu.org/. Accessed 15 Dec 2018.