P-Lingua Compiler: A Tool for Generating Ad-hoc Simulators in Membrane Computing

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

[1] Research Group on Natural Computing
Department of Computer Science and Artificial Intelligence
University of Seville, Seville, Spain
{perezh,dorellana,marper}@us.es
[2] School of Electrical Engineering
Southwest Jiaotong University, Chengdu, Shichuan, China
zhgxdylan@126.com

Abstract. Since the beginnings of membrane computing, software and hardware tools have been implemented for simulating computations of the proposed models. Some of these simulators are relatively generic, providing enough flexibility for a wide variety of models and others are ad-hoc simulators that reproduce computations of a single design that has been hard-coded or computations of a single type of model. On the one hand, generic tools are excellent assistants for the researchers while verifying their designs. On the other hand, the efficiency of specific tools in terms of simulation performance for a given design sacrifices the flexibility of the previous ones. In this paper, it is presented for the first time a tool that breaks this duality, we have implemented a compiler which receives as input the definition of a design in the P-Lingua language and produces as output source code in the C++ language for an ad-hoc simulator that has been optimized for the input design. The objective of this work is twofold: On the one hand, we have extended the P-Lingua framework to include some semantic features concerning to the models, such as rule patterns and derivation modes, that can be written in an explicit manner within their own file. On the other hand, we have developed a GNU GPLv3 command-line tool for Linux which works in the same manner as conventional compilers. Finally, we include in this paper a few examples for different types of cell-like and tissue-like models.

1 Introduction

Membrane computing is an unconventional model of computation within natural computing that was introduced in 1998 by Gh. Păun [17]. The computational devices in membrane computing, also known as P systems, are non-deterministic theoretical machines inspired on the biochemical processes that take place inside the compartments of living cells.

Several kinds of P systems coexist, each of them having different syntactic ingredients, such as different alphabets and structures. The two most studied are

cell-like membrane systems, characterized by their rooted tree structure, where membranes act as filters that let certain elements to pass through them [17], 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 interchange of objects between the different *compartments* is defined by the *rules* of the system, that together with the corresponding semantics, mark the functioning 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 [17] and antimatter [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. By using the rules specified in a model, we can make its objects change, both evolving and moving between the different compartments (membranes in the case of cell-like P systems and cells in the case of tissue-like P systems).

On the one hand, in P systems with active membranes [19], 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 to 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 and separation 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. Semantics rules the behavior of the system, determining which rules can be applied and how they affect the system according to a global clock. 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 system, that is, its syntax and semantics. According to the specification of a particular family of P systems, we consider a (specific) *model* as 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 another characteristics of the correspondent 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 [20], biology [23] and ecology [2], as well as a long list of other scientific lines [5, 13, 24], 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 [18] and [16].

Software and hardware simulators have been implemented from the beginnings of membrane computing. Some of these are very generic and flexible. On the other hand, we define an *ad-hoc* simulator as a simulator for one and only one membrane computing design which has been hard-coded. Such tools are usually the faster simulators since 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, 25] to include semantic features concerning to the models. On the other hand, 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 all the process can be automatized by using *makefiles*, i.e.,files which specify how to derive the target program.

The paper is structured as follows: In the next section, some preliminaries concepts about P-Lingua are introduced. In Section 3, we propose an extension for the P-Lingua language to directly define model constraints in the own 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 Section 6 some examples of the new P-Lingua extension are introduced. Finally, some conclusions and future work are drawn.

2 Preliminaries

P-Lingua [6, 25] 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. In order to achieve this, the first line of a P-Lingua file should include a P system model declaration by 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, 21] 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]. Remarkable software projects such as MeCoSim (Membrane Computing Simulator) [27, 22] use the simulators integrated in the library to perform P system computations and generate relevant information as 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 in order 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, 26] uses definitions generated by pLinguaCore in order 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 the references [6, 8, 10, 21, 25]. As an 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 $[\,[\quad]_3\,[\quad]_4\,]_2\,]_1$, an initial multiset for the membrane labelled 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 [25].

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 for the P-Lingua language to directly define model constraints in the 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:

$$p$$

$$u[v_1[v_{1,1}]^{\alpha_{1,1}}_{h_{1,1}} \dots [v_{1,m_1}]^{\alpha_{1,m_1}}_{h_{1,m_1}}]^{\alpha_1}_{h_1} \dots [v_n[v_{n,1}]^{\alpha_{n,1}}_{h_{n,1}} \dots [v_{n,m_n}]^{\alpha_{n,m_n}}_{h_{n,m_n}}]^{\alpha_n}_{h_n}$$

$$\xrightarrow{q} \text{ or } \xleftrightarrow{q}$$

$$w_0[w_1[w_{1,1}]^{\beta_{1,1}}_{g_{1,1}} \dots [w_{1,r_1}]^{\beta_{1,r_1}}_{g_{1,r_1}}]^{\beta_1}_{g_1} \dots [w_s[w_{s,1}]^{\beta_{s,1}}_{g_{s,1}} \dots [w_{s,r_s}]^{\beta_{s,r_s}}_{g_{s,r_s}}]^{\beta_s}_{g_s}$$

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 $[0, 1]$.
- $\alpha_i, \alpha_{i,j}, 1 \leq i \leq n, 1 \leq j \leq m_i$ and $\beta_i, \beta_{i,j}, 1 \leq i \leq s, 1 \leq j \leq r_i$ are electrical charges.
- $h_i, h_{i,j}, 1 \leq i \leq n, 1 \leq j \leq m_i$ and $g_i, g_{i,j}, 1 \leq i \leq s, 1 \leq j \leq r_i$ are membrane labels.
- $u, v_i, v_{i,j}, 1 \leq i \leq n, 1 \leq j \leq m_i$ and $w_i, w_{i,j}, 1 \leq i \leq s, 1 \leq j \leq r_i$ 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 `#` can be optionally used as 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 in order to restrict the rules used in each one. In order 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 than 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 Section 6, are given several examples of rule patterns in P-Lingua for different types of cell-like and tissue-like models.

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. As a part of semantics, it rules the exact application of rules of the system, deciding when rules can be applied or not when they are applicable. An extensive study of derivation modes can be found in [4]. In order to make the work self-content, 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 $\mathbf{R}$ 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:

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

  The multiset of rules finally applied to C_t is selected non-deterministically from $\mathbf{R}'$.

– **Bounded-by-rule parallel derivation mode** ($bound_{B_1,\ldots,B_r}$): Let $\{a, b, \ldots\}$ be the set of different types of rules present in a P system. B_i can be of the following forms:
  - $B_i = j, j \in \{a, b, \ldots\}$;
  - $B_i = \beta_n(B_{1_i}, \ldots, B_{r_i})$, being $n \in \mathbb{N}$, and for each $B_j = \beta_{m_j}(B_{1_j}, \ldots, B_{r_j})$, $j \in \{1_i, \ldots r_i\}$, $m_j \leq n$;
  - As a restriction, a label for 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 $B_i = \beta_n(B_{1_i}, \ldots, B_{r_i})$ 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:
  - In a *context* $\beta_n(B_1, \ldots, B_r)$, the number of rules 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 $bound_{B_1,\ldots,B_r}$, if $B_i = j (j \in \{a, b, \ldots\})$, being $1 \leq i \leq r$, then rules of type j can be applied in a maximal 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 $bound_{a,\beta_1(b,c,d,e,f)}$. If $\mathcal{R}_j$ is the set of rules from $\mathcal{R}$ of the type j, we formally define the bounded-by-rule maximally parallel mode by

$$\begin{aligned}\mathbf{R}' = \{\ & R \mid R \in \mathbf{R} \\ & \wedge \mid \{r \mid r \in R, r \in \mathcal{R}_j\} \mid \leq n \text{ for all } j \text{ in the context of } B_i = \beta_n \\ & \wedge\ \nexists R' \in \mathbf{R} : R \subsetneqq R'\}\end{aligned}$$

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:

```
@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-id,..., 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 Section 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 [28] and Bison [29]. 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 by 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 [30] 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 in order 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

not 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 [31], *makefiles* can also be used to automatize 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 Section 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.

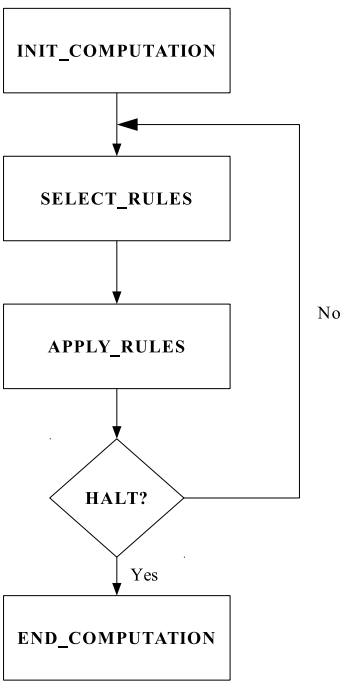

Fig. 1. The main simulation loop

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 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.

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)

6 Examples

6.1 Transition P systems

```
!transition_evolution /* Limited to rules with 3 inner membranes */
{
       [a -> v]'h;
```

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}

```
       [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;
```

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 pretends 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 in order 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 by 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 analysing 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 it is needed efficient solutions to hard problems. 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 by using more computational time.

Acknowledgements

This work was supported by National Natural Science Foundation of China (61672437 and 61702428) and by Sichuan Science and Technology Program (2018GZ0185, 2018GZ0086) and New Generation Artificial Intelligence Science and Technology Major Project of Sichuan Province (2018GZDZX0044).

Authors from the University of Seville also acknowledge the support of the research project TIN2017-89842-P, co-financed 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.

References

1. M. Colomer, A. Margalida, and M.J. Pérez-Jiménez. Population Dynamics P System (PDP) Models: A Standardized Protocol for Describing and Applying Novel Bio-Inspired Computing Tools, *Plos One*, 2013 8 (14), 1–13.
2. M. Cardona, M.A. Colomer, M.J. Prez-Jimnez, D. Sanuy, A. Margalida. Modeling ecosystems using P systems: The bearded vulture, a case study. Membrane Computing, 9th International Workshop, WMC 2008, Edinburgh, UK, July 28-31, 2008, Revised Selected and Invited Papers. *Lecture Notes in Computer Science*, 5391 (2009), 137-156.
3. M. Colomer, I. Pérez-Hurtado, M.J. Pérez Jiménez, and A. Riscos-Núñez. Comparing simulation algorithms for multienvironment probabilistic Psystem over a standard virtual ecosystem, *Natural Computing*, 11 (2012), 369–379.
4. R. Freund, S. Verlan. A Formal Framework for Static (Tissue) P Systems. *Lecture Notes in Computer Science*, 4860 (2007), 271–284.
5. P. Frisco, M. Gheorghe, M. J. Prez-Jimnez. Applications of Membrane Computing in Systems and Synthetic Biology. *Emergence, Complexity and Computation* (Series ISSN 2194-7287), Volume 7. Springer International Publishing, eBook ISBN 978-3-319-03191-0, Hardcover ISBN 978-3-319-03190-3, 2014, XVII + 266 pages (doi: 10.1007/978-3-319-03191-0).

6. M. García-Quismondo, R. Gutiérrez-Escudero, I. Pérez-Hurtado, M.J. Pérez-Jiménez, and A. Riscos-Núñez. An overview of P-Lingua 2.0, *Lecture Notes in Computer Science*, 5957 (2010), 264–288.
7. M. Ionescu, Gh. Păun, T. Yokomori. Spiking Neural P systems. *Fundamenta Informaticae*, **71**, 2-3 (2006), 279-308.
8. L.F. Macías, I. Pérez-Hurtado, M. García-Quismondo, L. Valencia, M.J. Prez-Jimnez, A. Riscos-Nez. A P-Lingua based simulator for Spiking Neural P systems. *Lecture Notes in Computer Science*, 7184 (2012), 257–281.
9. C. Martín-Vide, Gh. Păun, J. Pazos, A. Rodríghez-Patón. Tissue P systems. *Theoretical Computer Science*, **296**, 2 (2003), 295-326.
10. 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, 6 (2010), 374–382.
11. M.A. Martínez-del-Amor, I. Pérez-Hurtado, M. García-Quismondo, et al. DCBA: Simulating population dynamics P systems with proportional objects distribution, *Lecture Notes in Computer Science*, 7762 (2013), 257–276.
12. 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.
13. L. Pan, Gh. Paun, M. J. Prez-Jimnez, T. Song. Bio-inspired Computing: Theories and Applications. *Communications in Computer and Information Science* (Series ISSN 1865-0929), Volume 472, Springer-Verlag Berlin Heidelberg, Print ISBN 978-3-662-45048-2, Online ISBN 978-3-662-45049-9, 2014, XX + 672 pages (`doi: 10.1007/978-3-662-45049-9`).
14. L. Pan, Gh. Păun. Spiking Neural P Systems with Anti-Matter. *International Journal of Computers Communications & Control*, **4**, 3 (2009), 273–282.
15. L. Pan, T.-O. Ishdorj. P Systems with Active Membranes and Separation Rules. *Proceedings of the Second Brainstorming Week on Membrane Computing*, 2-7 February, 2004, Sevilla, Spain, pp. 325–341.
16. Gh. Păun, G. Rozenberg, A. Salomaa (eds.). *The Oxford Handbook of Membrane Computing*, Oxford University Press, Oxford, 2010.
17. Gh. Păun. Computing with membranes. *Journal of Computer and System Sciences*, **61**, 1 (2000), 108–143, and *Turku Center for CS-TUCS Report* No. 208, 1998.
18. Gh. Păun. *Membrane Computing. An introduction.* Springer-Verlag, Berlin, 2002.
19. Gh. Păun. P systems with active membranes: attacking **NP**–complete problems, *Journal of Automata, Languages and Combinatorics*, **6** (2001), 75–90.
20. H. Peng, J. Wang, J. Ming, P. Shi, M.J. Prez-Jimnez, W. Yu, Ch. Tao. Fault diagnosis of power systems using intuitionistic fuzzy spiking neural P systems. *IEEE Transactions on Smart Grid*, in press (2017) (`doi: 10.1109/TSG.2017.2670602`).
21. 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 , 1 (2014), 89–102.
22. I. Pérez-Hurtado, L. Valencia-Cabrera, M.J. Pérez-Jiménez, M. Colomer, and 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), 637–643.
23. F.J. Romero-Campero, M.J. Prez-Jimnez. A model of the Quorum Sensing System in Vibrio Fischeri using P systems. *Artificial Life*, 14, 1 (2008), 95-109 (`doi: 10.1162/artl.2008.14.1.95`).

24. G. Zhang, M. J. Prez-Jimnez, M. Gheorghe. Real-life applications with Membrane Computing. *Emergence, Complexity and Computation* (Series ISSN 2194-7287), Volume 25. Springer International Publishing, Online ISBN 978-3-319-55989-6, Print ISBN 978-3-319-55987-2, 2017, X + 367 pages (doi: 10.1007/978-3-319-55989-6).
25. The P-Lingua web page: http://www.p-lingua.org.
26. The PMCGPU web page: https://sourceforge.net/projects/pmcgpu/
27. The MeCoSim web page: http://www.p-lingua.org/mecosim/.
28. The Flex web page: https://github.com/westes/flexl
29. The Bison web page: https://www.gnu.org/software/bison/
30. The JSON web page: https://www.json.org/
31. The GNU g++ compiler: https://gcc.gnu.org/