P-Lingua: A Programming Language for Membrane Computing

Daniel Díaz–Pernil, Ignacio Pérez–Hurtado,
Mario J. Pérez–Jiménez, Agustín Riscos–Núñez

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

Summary. Software development for cellular computing has already been addressed, yielding a first generation of applications. In this paper, we develop a new programming language: P-Lingua. Furthermore, we present a simulator for the class of recognizing P systems with active membranes. We illustrate it by giving a solution to the SAT problem as an example.

1 Introduction

Membrane computing (or cellular computing) is an emerging branch within natural computing that was introduced by Gh. Păun [4]. The main idea is to consider biochemical processes taking place inside living cells from a computational point of view, in a way that gives us a new nondeterministic model of computation by using cellular machines.

Since the model was presented, many software applications have been produced (see [2], [10]). The common purpose of all of these software applications is to simulate P systems devices (cellular machines), and hence the designers have faced similar difficulties. However, these systems were usually focused on, and adapted for, particular cases, making it difficult to work on generalizations.

In order to give the first steps towards a next generation of applications, it is convenient to agree on some standards (specifications that regulate the performance of specific processes in order to guarantee their interoperability) and to implement the necessary tools and libraries.

When designing software for membrane computing, one has to describe precisely the P systems specification that is to be used. This task is hard if we need to handle families of P systems where the set of rules, the alphabet, the initial contents and even the membrane structure depend on the value assigned to some initial parameters. In existing software, several options have been implemented:

plain text files with a determined format, XML documents, graphical user interfaces, etc. As mentioned above, most of these solutions are adapted to specific models or to the specific purpose of the software.

In this paper we propose a programming language, called P-Lingua, whose programs define families of P systems in a parametric and modular way. After assigning values to the initial parameters, the compilation tool generates an XML document associated with the corresponding P system from the family, and furthermore it checks possible programming errors (both lexical/syntactical and semantical). Such documents can be integrated into other applications, thus guaranteeing interoperability. More precisely, in the simulators framework, the XML specification of a P system can be translated into an executable representation.

We present a practical application of P-Lingua in this paper. We give a simulator for recognizing P systems with active membranes that accepts as input an XML document generated by the compiler and that allows us to simulate a computation of the P system, obtaining the answer that the system outputs to its environment, plus a text file with a detailed step-by-step report of the computation.

The paper is structured as follows. In Section 2 several definitions and concepts are given for the sake of completeness of the paper. Section 3 introduces the P-Lingua programming language, and the syntax for P systems with active membranes is specified. In Section 4 we implement a solution to the SAT problem using P-Lingua. In Section 5 the compilation tool for the language is presented. Finally, Section 6 presents a simulator for recognizing P systems with active membranes. The paper ends with some conclusions and ideas for future work in Section 7.

2 Preliminaries

Polynomial time solutions to **NP**-complete problems in membrane computing are produced by trading time for space. This is inspired by the capability of cells to produce an exponential number of new membranes (new workspace) in polynomial time. Basically, there are two ways of producing new membranes in living cells: *mitosis* (membrane division) and *autopoiesis* (membrane creation). Both ways of generating new membranes have given rise to different variants of P systems: *P systems with active membranes*, where the new workspace is generated by membrane division, and *P systems with membrane creation*, where the new membranes are created from objects. Both models were proved to be computationally universal.

In this paper, we use the first variant mentioned above. Recall that a P system with active membranes is a construct of the form $\Pi = (O, H, \mu, \omega_1, \ldots, \omega_m, R)$, where $m \geq 1$ is the initial degree of the system; O is the alphabet of *objects*, and H is a finite set of *labels* for membranes; μ is a membrane structure, consisting of m membranes injectively labelled with elements of H, and $\omega_1, \ldots, \omega_m$ are strings over O, describing the *multisets of objects* placed in the m regions of μ; R is a finite set of *rules*, where each rule is of one of the following forms:

(a) $[a \rightarrow v]_h^\alpha$ where $h \in H$, $\alpha \in \{+, -, 0\}$ (electrical charges), $a \in O$ and v is a string over O describing a multiset of objects associated with membranes and depending on the label and the charge of the membranes (*object evolution rules*).

(b) $a\,[\,]_h^\alpha \rightarrow [b]_h^\beta$ where $h \in H$, $\alpha, \beta \in \{+, -, 0\}$, $a, b \in O$ (*send-in communication rules*). An object is introduced in the membrane, possibly modified, and the initial charge α is changed to β.

(c) $[a]_h^\alpha \rightarrow [\,]_h^\beta b$ where $h \in H$, $\alpha, \beta \in \{+, -, 0\}$, $a, b \in O$ (*send-out communication rules*). An object is sent out of the membrane, possibly modified, and the initial charge α is changed to β.

(d) $[a]_h^\alpha \rightarrow b$ where $h \in H$, $\alpha \in \{+, -, 0\}$, $a, b \in O$ (*dissolution rules*). A membrane with a specific charge is dissolved in reaction with a (possibly modified) object.

(e) $[a]_h^\alpha \rightarrow [b]_h^\beta\,[c]_h^\gamma$ where $h \in H$, $\alpha, \beta, \gamma \in \{+, -, 0\}$, $a, b, c \in O$ (*division rules*). A membrane is divided into two membranes. The objects inside the membrane are replicated, except for a, that may be modified in each membrane.

Rules are applied according to the following principles:

- Rules from (a) to (e) are used as is usual in the framework of membrane computing, i.e. in a maximal parallel way. In one step, each object in a membrane can only be used for one rule (non-deterministically chosen), but any object which can evolve by a rule must do it (with the restrictions indicated below).
- If a membrane is divided each object a in a membrane labelled with h and with charge α is divided into two membranes with label h, and one membrane has charge β and the second membrane has charge γ. The objects are replicated, but a can be modified in each membrane.
- If a membrane is dissolved, its content (multiset and interior membranes) becomes part of the immediately external one. The skin is never dissolved.
- All the elements which are not involved in any of the operations to be applied remain unchanged.
- Rules associated with label h are used for all membranes with this label, irrespective of whether the membrane is an initial one or whether it was created.
- Rules (b) to (e) can not be applied simultaneously in a membrane in one computation step.

Recognizing P systems were introduced in [5], and are the natural framework to study and solve decision problems, since deciding whether an instance has an affirmative or negative answer is equivalent to deciding if a string belongs or not to the language associated with the problem.

In the literature, recognizing P systems are associated in a natural way with P systems with *input*. The data related to an instance of the decision problem has to be provided to the P system in order for it to compute the appropriate answer. This is done by codifying each instance as a multiset placed in an *input membrane*. The output of the computation, *yes* or *no*, is sent to the environment.

A *P system with input* is a tuple (Π, Σ, i_Π), where: (a) Π is a P system, with working alphabet Γ, with p membranes labelled by $1, \ldots, p$, and initial multisets $\omega_1, \ldots, \omega_p$ associated with them; (b) Σ is an (input) alphabet strictly contained in Γ; the initial multisets are over $\Gamma \setminus \Sigma$; and (c) i_Π is the label of a distinguished (input) membrane.

Let m be a multiset over Σ. The *initial configuration* of (Π, Σ, i_Π) with input m is $(\mu, \omega_1, \ldots, \omega_{i_\Pi} + m, \ldots, \omega_p)$.

A *recognizing P system* is a P system with input, (Π, Σ, i_Π), and with external output such that:

(a) The working alphabet contains two distinguished elements, *yes* and *no*.
(b) The system always halts.
(c) If C is a computation of Π, then either some object *yes* or some object *no* (but no both) must be released into the environment, and only in the last step of the computation.

We say that C is an accepting computation (respectively, rejecting computation) if the object *yes* (respectively, *no*) appears in the external environment associated with the corresponding halting configuration of C.

In this paper, we present a programming language to define P systems with active membranes. A programming language is an artificial language that can be used to control the behavior of a machine, particularly a computer, but it can be used also to define a model of a machine that can be translated into an executable representation by a simulation tool. The act of simulating something generally entails representing certain key characteristics or behaviours of some physical, or abstract, system. Do not confuse a simulation tool with an emulation tool: the second one duplicates the functions of one system by using a different system, so that the second system behaves like (and appears to be) the first system. With the actual technology, we can not emulate the functionality of a cellular machine by using a conventional computer to resolve **NP** problems in polynomial time, but we can simulate these cellular machines, not necessarily in polynomial time, in order to aid researchers.

Programming languages, like natural languages, are defined by syntactic and semantic rules which describe their structure and meaning respectively. Usually, they are asociated with compilation tools that are computer programs that translates text written in a programming language into another language. The original sequence is usually called the source code whereas the output called the object code. Commonly the output has a form suitable for being processed by other programs or for being executed by the computer, but it may be a human-readable text file. In this paper, we use an XML language-like object code. The Extensible Markup Language (XML) is a general-purpose specification for creating custom markup languages. It is classified as an extensible metalanguage because it allows its users to define their own elements. Its primary purpose is to facilitate the sharing of structured data across different information systems. The files written by using a specific XML language are called XML documents.

The P system computations are massively parallel. One of the most common programming methods to simulate real parallelism in a conventional computer with a single processor is to use multithreading. A thread in this sense is a thread of execution. Threads are a way for a program to fork (or split) itself into two or more simultaneously (or pseudo-simultaneously) running tasks. Multiple threads can be executed in parallel on a single computer. This multithreading generally occurs by time-division multiplexing where the processor switches between different threads. This context switching can happen so fast as to give the illusion of parallelism to an end-user. On a multiprocessor or multi-core system, threading can be achieved via multiprocessing, wherein different threads can literally run simultaneously on different processors or cores.

3 The P-Lingua programming language

3.1 Syntax for P systems with active membranes

What follows is the syntax of the language for P systems with active membranes (whose description can be found in [6] and [1] among others.)

Valid identifiers

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

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

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

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

Identifiers for electrical charges

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

Data types

Two data types exist in P-Lingua:

- `Integer numbers:` We use 32 bits (signed) to store integer values, this allows a range from -2,147,483,648 to 2,147,483,647 for indexes and parameters.
- `Text strings:` These are valid identifiers that are used to define the alphabet objects and the membrane labels of a P system.

Variables

Two kind of variables are permitted in P-Lingua:

- `indexes`
- `Parameters`

Variables are used to store numeric values and their names are valid identifiers.

Numeric expressions

Numeric expressions can be written by using the `*` (multiplication), `/` (division), `%` (module), `+` (addition), `-` (subtraction) operators with integer numbers or variables, along with the use of parentheses.

Objects

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

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

Modules definition

Similarities between various solutions to **NP**-complete numerical problems by using families of recognizing P systems are discussed in [3]. Also, a cellular programming language is proposed based on libraries of subroutines. Using these ideas, a P-Lingua program consists of a set of programming modules that can be used more times by the same, or other, programs.

The syntax to define a module is the following.

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

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

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

In P-Lingua there are sentences to define the membranes configuration of a P system, to specify multisets, to define rules and to make calls to other modules. Next, let us see how such sentences are written.

Module calls

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

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

where `value`i is an integer number or a variable.

Definition of the initial membrane structure of a P system

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

```
@mu = expr;
```

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

Examples:

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

Definition of multisets

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

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

where `label` is a valid identifier or a natural number that represents a label of the structure of membranes and `list_of_objects` is a comma-separated list of objects. The character `#` is used to represent the empty multiset.

Union of multisets

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

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

Definition of rules

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

   α`[a --> v]'h`

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

   `a`α`[]'h -->`β`[b]`

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

   α`[a]'h -->` β`[]b`

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

   α`[a]'h -->`β`[b]`γ`[c]`

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

   α`[a]'h --> b`

where:

- α, β and γ are identifiers for electrical charges.
- `a`, `b` and `c` are objects of the alphabet.
- `v` is a comma-separated list of objects that represents a multiset.
- `h` is a label.

Some examples:

- $[x_{i,1} \rightarrow r_{i,1}^4]_2^+ \equiv$ `+[x{i,1} --> r{i,1}*4]'2`
- $d_k[\,]_2^0 \rightarrow [d_{k+1}]_2^0 \equiv$ `d{k}[]'2 --> [d{k+1}]`
- $[d_k]_2^+ \rightarrow []_2^0 d_k \equiv$ `+[d{k}]'2 --> []d{k}`
- $[d_k]_2^0 \rightarrow [d_k]_2^+[d_k]_2^- \equiv$ `[d{k}]'2 --> +[d{k}]-[d{k}]`
- $[a]_2^- \rightarrow b \equiv$ `-[a]'2 --> b`

Parametric sentences

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

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

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

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

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

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

Some examples of parametric sentences:

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

Inclusion of comments

The programs in P-Lingua can be commented by writing phrases into the text strings `/*` and `*/`.

4 Implementation of a solution to SAT problem

SAT problem is the following: *Given a boolean formula in conjunctive normal form (CNF), to determine whether or not it is satisfiable, that is, whether there exists an assignment to its variables on which it evaluates to true.*

4.1 A solution to SAT

In this section, we present a solution to the **SAT** problem using recognizing P systems with active membranes, given by M.J. Pérez–Jiménez et al. [6].

For each $(m, n) \in \mathbb{N}^2$, we consider the P system

$$(\Pi(\langle m, n\rangle), \Sigma(m, n), i(m, n))$$

where

- $\Sigma(m, n) = \{x_{i,j}, \bar{x}_{i,j} : 1 \leq i \leq m, 1 \leq j \leq n\}$
- $i(m, n) = 2$
- $\Pi(\langle m, n\rangle) = (\Gamma(m, n), \{1, 2\}, [[\,]_2]_1, w_1, w_2, R)$, is defined as follows:
  - $\Gamma(m, n) = \Sigma(m, n) \cup \{c_k : 1 \leq k \leq m + 2\} \cup$
    $\{d_k : 1 \leq k \leq 3n + 2m + 3\} \cup$
    $\{r_{i,k} : 0 \leq i \leq m, 1 \leq k \leq m + 2\} \cup \{e, t\} \cup \{Yes, No\}$
  - $w_1 = \emptyset$
  - $w_2 = \{d_1\}$

– The set of rules, R, is given by:

$\{[d_k]_2^0 \rightarrow [d_k]_2^+[d_k]_2^- : 1 \leq k \leq n\}$

$\{[x_{i,1} \rightarrow r_{i,1}]_2^+, [\bar{x}_{i,1} \rightarrow r_{i,1}]_2^- : 1 \leq i \leq m\}$

$\{[x_{i,1} \rightarrow \lambda]_2^-, [\bar{x}_{i,1} \rightarrow \lambda]_2^+ : 1 \leq i \leq m\}$

$\{[x_{i,j} \rightarrow x_{i,j-1}]_2^+, [x_{i,j} \rightarrow x_{i,j-1}]_2^- : 1 \leq i \leq m, 2 \leq j \leq n\}$

$\{[\bar{x}_{i,j} \rightarrow \bar{x}_{i,j-1}]_2^+, [\bar{x}_{i,j} \rightarrow \bar{x}_{i,j-1}]_2^- : 1 \leq i \leq m, 2 \leq j \leq n\}$

$\{[d_k]_2^+ \rightarrow []_2^0 d_k, [d_k]_2^- \rightarrow []_2^0 d_k : 1 \leq k \leq n\}$

$\{d_k[]_2^0 \rightarrow [d_{k+1}]_2^0 : 1 \leq k \leq n-1\}$

$\{[r_{i,k} \rightarrow r_{i,k+1}]_2^0 : 1 \leq i \leq m, 1 \leq k \leq 2n-1\}$

$\{[d_k \rightarrow d_{k+1}]_1^0 : n \leq k \leq 3n-3\}; [d_{3n-2} \rightarrow d_{3n-1}e]_1^0$

$e[]_2^0 \rightarrow [c_1]_2^+; [d_{3n-1} \rightarrow d_{3n}]_1^0$

$\{[d_k \rightarrow d_{k+1}]_1^0 : 3n \leq k \leq 3n+2m+2\}$

$[r_{1,2n}]_2^+ \rightarrow []_2^- r_{1,2n} \;;\; \{[r_{i,2n} \rightarrow r_{i-1,2n}]_2^- : 1 \leq i \leq m\}$

$r_{1,2n}[]_2^- \rightarrow [r_{0,2n}]_2^+$

$\{[c_k \rightarrow c_{k+1}]_2^- : 1 \leq k \leq m\}$

$[c_{m+1}]_2^+ \rightarrow []_2^+ c_{m+1} \;;\; [c_{m+1} \rightarrow c_{m+2}t]_1^0$

$[t]_1^0 \rightarrow []_1^+ t \;;\; [c_{m+2}]_1^+ \rightarrow []_1^- Yes \;;\; [d_{3n+2m+3}]_1^0 \rightarrow []_1^+ No$

4.2 Implementation

The following is the code of the program written in P-Lingua that encodes a solution to the SAT problem.

Objects of the form $\bar{x}_{i,j}$ are written as `nx{i,j}`.

```
/* Module that defines a family of recognizing P systems
   to solve the SAT problem */
def Sat(m,n)
{
 /* Initial configuration */
 @mu = [[]'2]'1;

 /* Initial multisets */
 @ms(2) = d{1};

 /* Set of rules */
 [d{k}]'2 --> +[d{k}]-[d{k}] : 1 <= k <= n;
```

```
 {
  +[x{i,1} --> r{i,1}]'2;
  -[nx{i,1} --> r{i,1}]'2;
  -[x{i,1} --> #]'2;
  +[nx{i,1} --> #]'2;
 } : 1 <= i <= m;

 {
  +[x{i,j} --> x{i,j-1}]'2;
  -[x{i,j} --> x{i,j-1}]'2;
  +[nx{i,j} --> nx{i,j-1}]'2;
  -[nx{i,j} --> nx{i,j-1}]'2;
 } : 1<=i<=m, 2<=j<=n;

 {
  +[d{k}]'2 --> []d{k};
  -[d{k}]'2 --> []d{k};
 } : 1<=k<=n;

 d{k}[]'2 --> [d{k+1}] : 1<=k<=n-1;
 [r{i,k} --> r{i,k+1}]'2 : 1<=i<=m, 1<=k<=2*n-1;
 [d{k} --> d{k+1}]'1 : n <= k<= 3*n-3;
 [d{3*n-2} --> d{3*n-1},e]'1;
 e[]'2 --> +[c{1}];
 [d{3*n-1} --> d{3*n}]'1;
 [d{k} --> d{k+1}]'1 : 3*n <= k <= 3*n+2*m+2;
 +[r{1,2*n}]'2 --> -[]r{1,2*n};
 -[r{i,2*n} --> r{i-1,2*n}]'2 : 1<= i <= m;
 r{1,2*n}-[]'2 --> +[r{0,2*n}];
 -[c{k} --> c{k+1}]'2 : 1<=k<=m;
 +[c{m+1}]'2 --> +[]c{m+1};
 [c{m+1} --> c{m+2},t]'1;
 [t]'1 --> +[]t;
 +[c{m+2}]'1 --> -[]Yes;
 [d{3*n+2*m+3}]'1 --> +[]No;

} /* End of Sat module */

/* Main module */
def main()
{
 /* Call to Sat module for m=4 and n=6 */
 call Sat(4,6);
 /* Expansion of the input multiset */
```

```
 @ms(2) += x{1,1}, nx{1,2}, nx{2,2}, x{2,3},
           nx{2,4}, x{3,5}, nx{4,6};
} /* End of main module */
```

The module `main` is instantiated with the formula

$$\varphi \equiv (x_1 + \overline{x}_2)(\overline{x}_2 + x_3 + \overline{x}_4)\, x_5\, \overline{x}_6$$

where $n = 6$, $m = 4$ and the input multiset: $x_{1,1}, \overline{x}_{1,2}, \overline{x}_{2,2}, x_{2,3}, \overline{x}_{2,4}, x_{3,5}, \overline{x}_{4,6}$.

5 The P-Lingua compiler

A compiler is a program that translates code written in some computer language to another language. We have developed a compiler that is able to translate programs written in P-Lingua into XML documents, after having assigned values to some initial parameters. Recall that a P-Lingua program can, in a flexible way, encode a family of P systems (with the help of some parameters), whereas the XML document generated by the compiler specifies only a single P system of the family. In this way, the applications do not need to process parametric systems, and hence their implementation is much easier.

The choice of the metalanguage XML is due to the fact that it is a broadly known standard, that has the following advantages:

- It is extensible. After having an XML specification designed, one can extend it by adding new labels, allowing in this way compatibility with earlier versions.
- The analyzer is a generic component, it is not necessary to create a new one for each XML specification. This avoids errors and speeds up the development of applications.
- The structure of the language is easy to understand and to process, facilitating compatibility with earlier versions.

5.1 An XML language for P systems with active membranes

The structure of the XML documents generated by the P-Lingua compiler for P systems with active membranes is as follows:

```
<?xml version="1.0"?>
<active_membrane_psystem version="1.0">

  <init_config>
   ...
  </init_config>
```

```
  <multisets>
   ...
  </multisets>

  <rules>
   ...
  </rules>

</active_membrane_psystem>
```

The main element is named `active_membrane_psystem`, and it has an attribute indicating the version of the specification. There are three internal elements:

- `init_config`: defines the membrane structure of the initial configuration.
- `multisets`: defines the initial multisets.
- `rules`: defines the set of rules.

Definition of the membrane structure of the initial configuration

Next, we describe the element `init_config` corresponding to the membrane structure $[[\,]_e^+[\,]_r^-]_s^0$.

```
<init_config>
  <membrane label="s" charge="0">
    <membrane label="e" charge="+1"/>
    <membrane label="r" charge="-1"/>
  </membrane>
</init_config>
```

`init_config` allows a recursive representation of a membrane structure. The element `membrane` has two attributes: `label`, which indicates the label of the membrane, and `charge`, which can take values `0`, `+1` or `-1` and indicates the membrane polarization.

Definition of initial multisets

Multisets of objects present in membranes are defined through the element `multisets`. Let us consider the following example: $w_e = e_0, g_1$, $w_s = z_1^3$ and $w_r = h_0, b_0$.

```
<multisets>
  <multiset label="e">
    <object name="e{0}" multiplicity="1"/>
```

```
    <object name="g{1}" multiplicity="1"/>
  </multiset>
  <multiset label="s">
    <object name="z{1}" multiplicity="3"/>
  </multiset>
  <multiset label="r">
    <object name="h{0}" multiplicity="1"/>
    <object name="b{0}" multiplicity="1"/>
  </multiset>
</multisets>
```

As it can be seen in the example, the element `multisets` is composed of several elements of type `multiset`, each of them having an attribute `label` indicating the label of the membrane where the multiset is contained. The objects present in the multiset are represented by elements of type `object` with two attributes: `name` indicates the symbol naming the object, and `multiplicity` indicates the multiplicity of the object in the multiset.

Definition of the set of rules

Let us consider the following set of rules:

- $[c_9 \rightarrow c_{10}t]_1^0$
- $[r_{1,16}]_2^+ \rightarrow [\,]_2^- r_{1,16}$
- $r_{1,16}[\,]_2^- \rightarrow [r_{0,16}]_2^+$
- $[d_0]_2^0 \rightarrow [d_0]_2^+ [d_0]_2^-$
- $[a]_e^0 \rightarrow b$

The element `rules` is described as follows:

```
<rules>
  <evolution_rule label=1" charge="0">
    <left_hand_rule object="c{9}"/>
    <right_hand_rule object="c{10}" multiplicity="1"/>
    <right_hand_rule object="t" multiplicity="1"/>
  </evolution_rule>
  <send_out_rule label="2" charge="+1">
    <left_hand_rule object="r{1,16}"/>
    <right_hand_rule object="r{1,16}" charge="-1"/>
  </send_out_rule>
  <send_in_rule label="2" charge="-1">
    <left_hand_rule object="r{1,16}"/>
    <right_hand_rule object="r{0,16}" charge="+1"/>
```

```
    </send_in_rule>
    <division_rule label="2" charge="0">
      <left_hand_rule object="d{0}"/>
      <right_hand_rule object="d{0}" charge="+1"/>
      <right_hand_rule object="d{0}" charge="-1"/>
    </division_rule>
    <dissolution_rule label="e" charge="0">
      <left_hand_rule object="a"/>
      <right_hand_rule object="b"/>
    </dissolution_rule>
  </rules>
```

Within the element `rules` we can find five different types of elements: `evolution_rule`, `send_in_rule`, `send_out_rule`, `division_rule` and `dissolution_rule`. All of them contain two attributes: `label` and `charge`, indicating the label and polarization of the membranes to which the rule can be applied.

Besides, there exists an internal element called `left_hand_rule` with an attribute called `object` containing the name of the object that triggers the rule.

For the case of evolution rules, the compiler generates one or more elements of type `right_hand_rule`, each of them having two attributes `object` and `multiplicity` expressing the name of the object produced by the rule, and the number of copies obtained.

Communication rules have only one element `right_hand_rule` with the name of the resulting object and the polarization that the membrane gets after applying the rule.

For division rules, there are two elements `right_hand_rule`, indicating the objects obtained in the two resulting membranes, as well as their respective polarizations.

Finally, for dissolution rules, only one element `right_hand_rule` showing the name of the object that is obtained.

5.2 The compilation tool

The P-Lingua compiler (version 1.0) and its source code can be freely downloaded from the *software* section in the website of the Research Group on Natural Computing [11]. The compiler is under GPL license [7] and is written in Java [8] using the lexical and syntactical analyzers provided by JavaCC [9]. The minimum system requirements are having a Java virtual machine (JVM) version 1.6.0 running in a Pentium III computer.

The compilation tool is a program that may be exectuted from the command line as follows:

```
plingua input_file -xml output_file [-v verbosity_level] [-h]
```

The text file `input_file` contains the program (written in P-Lingua) that we want to be compiled, and `output_file` is the name of the XML file that is generated. Optional arguments are in brackets: `verbosity_level` is a number between 0 and 5 indicating the level of detail of the messages shown during the compilation process, and the option `-h` displays some help information.

6 A simulator for recognizing P systems with active membranes

As a first practical application of the P-Lingua programming language, we have implemented a simulator for recognizing P systems with active membranes that takes as input an XML document generated by the P-Lingua compiler and runs one of the possible computations that the P system may follow, obtaining the answer that the system outputs to its environment, plus a text file with a detailed step-by-step report of the computation.

This simulator is again a Java program under GPL license that can be freely downloaded from the *software* section in the web of the Research Group on Natural Computing [11]. The system requirements are the same as in the case of the P-Lingua compiler.

The simulator is launched from the command line as follows:

```
plingua_sim input_xml [-o output_file]
```

where `input_xml` is an XML document formatted as discussed in this paper, and `output_file` is the name of the file where the report about the simulated computation will be saved.

6.1 Simulation of a solution to SAT problem

We now show an execution of the simulator running on the XML document obtained after compiling the P-Lingua program described in Section 4.2. The results have been obtained on an AMD Sempron machine, at 2.8 Ghz and with 512Mb of RAM memory.

The command used to execute the simulation is:

```
plingua_sim sat.xml -o info.txt
```

The simulation ends when no more rules can be applied, and then the following information is displayed:

```
Environment: t, Yes
Steps: 41
Time: 1.971 s.
Halting configuration (No rule can be selected to be executed
in the next step)
```

Thus, the computation of the P system lasted 41 transition steps, and it took 1,971 seconds to simulate it until reaching a halting configuration (recall that we are simulating a parallel device on a sequential computer).

The file `info.txt` keeps detailed information about each configuration of the simulated computation. More precisely, the multisets and polarizations of all the membranes are listed, as well as the rules selected for execution at each transition step. The configurations are numbered (starting at 0), to keep track of the step of the computation that is being simulated. Some information about the CPU time is shown for each step, and the number of rules of each type that is executed. As an example, we give the information generated for the first two configurations.

```
### MEMBRANE ID: 1, Label: 2, Charge: 0
    Multiset: nx{1, 2}, d{1}, x{3, 5}, nx{2, 4}, nx{2, 2},
              nx{4, 6}, x{2, 3}, x{1, 1}
    Parent Membrane ID: 0
    Rules Selected:
    1*DIVISION RULE: [d{1}]'2 --> +[d{1}] -[d{1}]

@@@ SKIN MEMBRANE ID: 0, Label: 1, Charge: 0
    Multiset: #
    Internal membranes count: 1

Configuration: 0
Time: 0.0 s.
1 division rule(s) selected to be executed in the step 1
***********************************************
### MEMBRANE ID: 1, Label: 2, Charge: +
    Multiset: nx{1, 2}, d{1}, x{3, 5}, nx{2, 4}, nx{2, 2},
              nx{4, 6}, x{2, 3}, x{1, 1}
    Parent Membrane ID: 0
    Rules Selected:
    1*EVOLUTION RULE: +[nx{2, 2} --> nx{2, 1}]'2
    1*EVOLUTION RULE: +[nx{1, 2} --> nx{1, 1}]'2
    1*EVOLUTION RULE: +[x{3, 5} --> x{3, 4}]'2
    1*EVOLUTION RULE: +[x{1, 1} --> r{1, 1}]'2
    1*EVOLUTION RULE: +[nx{2, 4} --> nx{2, 3}]'2
    1*EVOLUTION RULE: +[nx{4, 6} --> nx{4, 5}]'2
    1*EVOLUTION RULE: +[x{2, 3} --> x{2, 2}]'2
    1*SEND-OUT RULE: +[d{1}]'2 --> []d{1}

### MEMBRANE ID: 2, Label: 2, Charge: -
    Multiset: nx{1, 2}, d{1}, nx{2, 4}, x{3, 5}, nx{2, 2},
              x{2, 3}, nx{4, 6}, x{1, 1}
    Parent Membrane ID: 0
```

```
    Rules Selected:
    1*EVOLUTION RULE: -[nx{2, 4} --> nx{2, 3}]'2
    1*EVOLUTION RULE: -[nx{2, 2} --> nx{2, 1}]'2
    1*EVOLUTION RULE: -[nx{4, 6} --> nx{4, 5}]'2
    1*EVOLUTION RULE: -[x{1, 1} --> #]'2
    1*EVOLUTION RULE: -[x{2, 3} --> x{2, 2}]'2
    1*EVOLUTION RULE: -[nx{1, 2} --> nx{1, 1}]'2
    1*EVOLUTION RULE: -[x{3, 5} --> x{3, 4}]'2
    1*SEND-OUT RULE: -[d{1}]'2 --> []d{1}

@@@ SKIN MEMBRANE ID: 0, Label: 1, Charge: 0
    Multiset: #
    Internal membranes count: 2

Configuration: 1
Time: 0.025 s.
14 evolution rule(s) selected to be executed in the step 2
2 send-out rule(s) selected to be executed in the step 2
***********************************************
```

After simulating 41 transition steps, the halting configuration is described as follows:

```
### MEMBRANE ID: 1, Label: 2, Charge: +
    Multiset: r{0, 12}*3, c{4}
    Parent Membrane ID: 0

### MEMBRANE ID: 2, Label: 2, Charge: +
    Multiset: c{1}, r{2, 12}, r{3, 12}
    Parent Membrane ID: 0

### MEMBRANE ID: 3, Label: 2, Charge: +
    Multiset: r{0, 12}*5, c{4}
    Parent Membrane ID: 0

### MEMBRANE ID: 4, Label: 2, Charge: +
    Multiset: r{0, 12}*4, c{4}
    Parent Membrane ID: 0

### MEMBRANE ID: 5, Label: 2, Charge: +
    Multiset: r{0, 12}, r{2, 12}, c{2}
    Parent Membrane ID: 0

### MEMBRANE ID: 6, Label: 2, Charge: +
    Multiset: c{1}, r{3, 12}
```

```
    Parent Membrane ID: 0

### MEMBRANE ID: 7, Label: 2, Charge: +
    Multiset: r{0, 12}*4, c{4}
    Parent Membrane ID: 0
```

⋮

```
@@@ SKIN MEMBRANE ID: 0, Label: 1, Charge: -
    Multiset: t*10, d{29}*64, c{6}*10
    Internal membranes count: 64

~~~ENVIRONMENT: t, Yes

Configuration 41
Time: 1.971 s.
Halting configuration (No rule can be selected to be
executed in the next step)

*********************************************
```

Note that there are 64 different membranes labelled by 2 in this configuration, although for the sake of simplicity we show only seven of them.

7 Conclusions and future work

In this paper we have presented the first programming language for membrane computing, *P-Lingua*, together with a compiler that generates XML documents, and a simulator for a class of P systems called recognizing P systems with active membranes.

Using a programming language to define cellular machines is a new concept in the development of applications for membrane computing that leads to a standardization with the following advantages:

- Users (researchers) can define cellular machines in a modular and parametric way by using an easy-to-learn programming language.
- It is possible to define libraries of modules that can be shared among researchers to facilitate the design of new programs.
- This method to define P systems is decoupled from its applications and the same P-Lingua programs can be used in different software enviroments.
- By using compiling tools, the P-Lingua programs are translated to other file formats that can be interpreted by a large number of different applications.

The first version of P-Lingua is presented for P systems with active membranes. In forthcoming versions we intend to generalize the language so that other types of

cellular devices can be also specified, for instance transition P systems and tissue P systems.

Currently, the compiler is an application that is executed from the command line, but the possibility of a graphical programming environment remains open.

We have chosen an XML language as the output format because of the reasons exposed above. However, we are aware that for some applications it is not the most suitable format, due to the fact that XML does not include any method for compressing data, and therefore the text files can eventually become too large, which is a clear disadvantage for applications running on networks of processors. It would be convenient to modify the compiler so that it generates a larger variety of output formats, of special interest are compressed binary files or executable code (either in C or Java).

It is important to recall that the simulator presented here is designed to run in a conventional computer, having limited resources (RAM, CPU), and this leads to a bound on the size of the instances of **NP**-complete problems whose solutions can be successfully simulated. Moreover, conventional computers are not massively parallel devices, and therefore it seems that the inherent parallelism of P systems must be simulated by means of multithreading techniques.

These shortcomings lead us to the possibility of implementing a distributed simulator running on a network or cluster of processors, where the need of resources arising during the computation could be solved by adding further nodes to the network, thus moving towards massive parallelism.

Acknowledgement

The authors acknowledge the valuable assistance given by Damien Woods who helped us to write this paper.

The authors also wish to acknowledge the support of the project TIN2006-13425 of the Ministerio de Educación y Ciencia of Spain, cofinanced by FEDER funds, as well as the support of the project of excellence TIC-581 of the Junta de Andalucía.

References

1. A. Alhazov, M.J. Pérez–Jiménez. Uniform solution of QSAT using polarizationless active membranes. In J. Durand–Lose and M. Margenstern (eds.) *Machines, Computations, and Universality. Lecture Notes in Computer Science*, 4664 (2007), pp. 122–133.
2. M. Gutiérrez–Naranjo, M.J. Pérez–Jiménez, A. Riscos–Núñez. Available membrane computing software. In G. Ciobau, Gh. Păun, M.J. Péréz–Jiménez (eds.) *Applications of Membrane Computing, Natural Computing Series*, Springer–Verlag, Berlin, 15 (2006), pp. 411–436.
3. M.A. Gutiérrez, M.J. Pérez–Jiménez, A. Riscos–Núñez. Towards a programming language in cellular computing. *Electronic Notes in Theoretical Computer Science*, Elsevier B.V., 123 (2005), pp. 93–110.

4. Gh. Păun. Computing with membranes. *Journal of Computer and System Sciences*, 61, 1 (2000), pp. 108–143.
5. M.J. Pérez-Jiménez, A. Romero–Jiménez, F. Sancho–Caparrini. Complexity classes in cellular computing with membranes. *Natural Computing*, **2**, 3 (2003), 265-285.
6. M.J. Pérez–Jiménez, A. Romero–Jiménez, F. Sancho–Caparrini. A polynomial complexity class in P systems using membrane division. In E. Csuhaj Varjú, C. Kintala, D. Wotschke, G. Vaszil (eds.), *Proceedings of the 5th Workshop on Descriptional Complexity of Formal Systems, DCFS 2003*, Computer and Automation Research Institute of the Hungarian Academy of Sciences, Budapest, (2003), pp. 284–294.
7. The GNU General Public License: `http://www.gnu.org/copyleft/gpl.html`
8. Java web page: `http://www.java.com/`
9. JavaCC web page: `https://javacc.dev.java.net/`
10. P systems web page: `http://ppage.psystems.eu/`
11. Research Group on Natural Computing web page: `http://www.gcn.us.es/`