A MzScheme Implementation of Transition P Systems

Delia Balbontín Noval, Mario J. Pérez Jiménez,
and Fernando Sancho Caparrini

Abstract. The main goal of this paper is to present the design of an MzScheme program that allows us to simulate the behavior of transition P systems. For that, a library of procedures have been developed that work in two stages. In the first one, the *parsing/compiling* stage, the input P system is checked, and if it is well defined, then it is represented by means of an internal grammar. In a second stage, the *simulation*, the computation tree associated to the P system is generated until a prefixed level.

1 Introduction

In October 1998, Gheorghe Păun [1] introduced a new computability model of a non-deterministic and highly parallel type, the *membrane systems*. They are based on the synchronized work of several units, called *membranes*, structured in a dynamic hierarchy (understood as vesicles in a space) embedded in a *skin membrane* that separates the system from the environment. When a membrane has no membrane inside, it is called *elementary*. Each membrane encloses a space between it and the membranes directly included in it (if any). This space (the *region* of the membrane) can contain a multiset (a set where the elements can be repeated) of objects (represented by symbols of a given alphabet) and a set of (evolution) rules for them. Each membrane defines an unique region.

This model, called *transition P systems*, is inspired from the observation that the processes which take place in the complex structure of a living cell can be viewed as computation–like processes.

We present here a library of MzScheme procedures [5], that allows us both to input easily a transition P system and to simulate its non-deterministic and highly parallel behavior. It reads, analyzes and compiles the input data defining a P system; then, it generates the subsequent computations.

Our implementation is based on the formalization given in [3].

The program runs in two independent stages: *parsing/compiling* and *simulation/running.*

$$\Pi - input \quad \xrightarrow{parser/compiler} \quad \Pi \quad \xrightarrow{simulator} \quad \mathbf{Comp}(\Pi)$$

At stage one, *parsing/compiling*, the input data are read and the respective P system is rewritten as an element of the language generated by a proposed *internal grammar*. To get it, the input data have to be *syntactically correct* according to the *input grammar*. Moreover, they have to define a *well defined* P system (according to the formalization above mentioned).

Stage two, *simulation/running*, starts when the *parsing/compiling* is finished. Starting from the P system *initial configuration*, the associated *computation tree* is generated. The expansion of that *computation tree* is made in a progressive way, level by level (*breadth expansion*), until to a given depth level. To get it we follow a *breadth-expansion-tree* scheme based on the definitions and steps proposed in [3]:

applicable-rules →
applicability-vectors →
applicability-matrices → *configurations*

This paper is organized as follows: Section 2 briefly presents some basic concepts about a formalization of transition P systems, following [3]. Section 3 describes briefly the whole *simulator* scheme. Section 4 is about the way to input a P system, showing the proposed *input grammar*. Section 5 presents the *internal grammar* and describes the *parser/compiler* performance. Section 6 describes the *simulator* behavior properly. Finally, in Section 7 we present a complete example to illustrate the way of working of the program.

2 Preliminaries about a Formalization of Transition P Systems

Following [3], we recall here the basic concepts and definitions about P systems.

2.1 Membrane Structure and Cells

A *membrane structure* is a rooted tree, where the nodes are called *membranes*, the root is called *skin*, and the leaves are called *elementary membranes*.

A *cell* over an alphabet, A, is a pair (μ, M), where $\mu = (V(\mu), E(\mu))$ is a membrane structure, and M is an application, $M : V(\mu) \longrightarrow \mathbf{M}(A)$ (the set of multisets over A).

2.2 Evolution Rules

Let $C = (\mu, M)$ be a cell over an alphabet A. Let $x \in V(\mu)$. An *evolution rule* associated to x is a 3-tuple $r = (d_r, v_r, \delta_r)$ where d_r is the left-side of the rule, v_r is the right-side of the rule, and $\delta_r \in \{\neg\delta, \delta\}$ indicates if the application of the rule dissolves the membrane.

A *collection R of evolution rules* associated to C is a function with domain $V(\mu)$ such that for every membrane $x \in V(\mu)$, $R_x = \{r_{x,1}, \ldots, r_{x,s_x}\}$ is a finite set (possibly empty) of (evolution) rules associated to x. A *priority relation over* R is a function, ρ, with domain $V(\mu)$ such that for every membrane $x \in V(\mu)$, ρ_x is a strict partial order over R_x (possibly empty).

2.3 Transition P Systems

A *transition P system* is a 4-tuple $\Pi = (A, C_0, \mathcal{R}, i_0)$, where:

- A is a non-empty finite set (usually called base alphabet).
- $C_0 = (\mu_0, M_0)$ is a cell over A.
- $\mathcal{R}$ is an ordered pair (R, ρ) where R is a collection of (evolution) rules associated to C_0, and ρ is a priority relation over R.
- i_0 is a node of μ_0, which specifies the output membrane of Π.

The number $|V(\mu_0)|$ is called the *degree* of Π.

2.4 Configurations

A *configuration*, C, of a P system, $\Pi = (A, C_0, \mathcal{R}, i_0)$ with $C_0 = (\mu_0, M_0)$, is a cell $C = (\mu, M)$ over A, where $V(\mu) \subseteq V(\mu_0)$, and μ has the same root as μ_0. The configuration C_0 will be called the *initial configuration of* Π.

2.5 Applicability

Let $x \in V(\mu_0)$. We say that the (evolution) rule $r \in R_x$ is *semi-applicable* to C if the membrane associated to node x exists in C (dissolution is not allowed in the root node), the membrane associated to x has all the necessary objects to apply the rule, and nodes where the rule tries to send objects (by means of in_y) are children of x.

We say that the rule $r \in R_x$ is *applicable* to C, if it is semi-applicable to C and there is no semi-applicable rules in R_x with higher priority.

We say that $\boldsymbol{p} \in \mathbf{N}^{\mathbf{N}}$ is an *applicability vector* over $x \in V(\mu)$ for C, and we will denote it as $\boldsymbol{p} \in \mathbf{Ap}(x, C)$, if it has correct size (that is, for all j greater the number of rules associated to x we have $p(j) = \emptyset$), every rule can be applied as many times as the vector $\boldsymbol{p}$ indicates, all the rules can be applied simultaneously, and it is maximal.

We will say that $P : V(\mu_0) \longrightarrow \mathbf{N}^{\mathbf{N}}$ is an *applicability matrix* over C, denoted $P \in \mathbf{M}_{\mathbf{Ap}}(C)$, if for every $x \in V(\mu_0)$ we have that $P(x) \in \mathbf{Ap}(x, C)$.

2.6 Transitions

The *execution* of P over $C = (\mu, M)$, denoted $P(C)$, returns a new configuration $C' = (\mu', M')$ of Π, that can be considered acting in two stages: $(\mu, M) \rightarrow (\mu, M'') \rightarrow (\mu', M')$.

In the first stage we suppose that the rules are applied without attending dissolving actions, and in the second one dissolution and distribution of contents are carried out.

We will say that a configuration C_1 of a P system Π yields a configuration C_2 by a *transition in one step* of Π, denoted $C_1 \Rightarrow_\Pi C_2$, if there exists a non–zero applicability matrix over C_1, P, such that $P(C_1) = C_2$.

2.7 Computation Tree

The *computation tree of a P system Π*, denoted $\mathbf{Comp}(\Pi)$, is a rooted labeled maximal tree defined as follows: the root of the tree is the initial configuration, C_0, of Π; the children of a node are the configurations that follow in one step of transition; nodes and edges are labeled by configurations and applicability matrices, respectively, in such way that two labeled nodes C, C' are adjacent in $\mathbf{Comp}(\Pi)$, by means of an edge labeled with P, if and only if $P \in \mathbf{M_{Ap}}(C) - \{\mathbf{0}\}$ and $C' = P(C)$. The maximal branches of $\mathbf{Comp}(\Pi)$ will be called *computations* of Π. We will say that a computation of Π *halts* if it is a finite branch. The configurations verifying $\mathbf{M_{Ap}}(C) = \{\mathbf{0}\}$ will be called *halting configurations*.

3 Preliminaries about the P Systems Simulator

We consider that the basic features of a computing program able to simulate transition P systems should be the following:

1. To have a formal definition of transition P systems to be based on.
2. To choose a suitable programming language to implement the simulation.
3. To have an easy way to input the data describing the P system.
4. To choose an efficient internal representation of P systems.
5. To design a parser/compiler to analyze the input data and to obtain the P system internal representation.
6. To design a P system simulator of computations to generate the respective computation tree.

As we said previously, the implementation we present here has been developed on MzScheme (a functional language from Lisp family), and it is based on the formalization given in the above section, but slightly modified. This modification arises from the convenience to identify the *applicable* rules to a given configuration.

The rules of a P system are static elements. Nevertheless, to determine if a rule $r = (d_r, v_r, \delta_r)$ is *applicable* to an arbitrary configuration C, a new component $\alpha_r \in \{\#\mathtt{t}, \#\mathtt{f}\}$ has been added, getting $r^* = (d_r, v_r, \delta_r, \alpha_r)$. Initially, α_r will be set to $\#\mathtt{f}$; it will be modified to $\#\mathtt{t}$ if (and only if) the rule r is applicable to C. Consequently, if we denoted for every $x \in V(\mu_0)$, $R_x = \{r_{x,1}, \dots, r_{x,s_x}\}$, then we have $R^*_x = \{r^*_{x,1}, \dots, r^*_{x,s_x}\}$, with $r^*_{x,j} = (d_{x,j}, v_{x,j}, \delta_{x,j}, \alpha_{x,j})$ and $\alpha_{x,j} = \#\mathtt{f}$; then, $R^* = \bigcup_{x \in V(\mu_0)} R^*_x$, and $\mathcal{R} = (R^*, \rho)$.

Moreover, for every configuration, $C = (\mu, M)$ and every $x \in V(\mu_0)$, we will denote by $R^C_x = \{r^C_{x,1}, \dots, r^C_{x,s_x}\}$, with $r^C_{x,j} = (d_{x,j}, v_{x,j}, \delta_{x,j}, \alpha_{x,j})$ and $\alpha_{x,j} = \#\mathtt{t}$ if and only if the rule $r_{x,j}$ is applicable to C, the *tagged-rules* of x to C. Finally, we will note $R^C = \bigcup_{x \in V(\mu_0)} R^C_x$.

4 The Input of a Transition P System

To define a P system we need to input its membrane structure and describe the content of every membrane. Each membrane has symbols from a given alphabet, transition rules and priority relations over them. The membrane structure has to be a rooted tree and the priority between rules must be a strict partial order.

4.1 Default Settings

In order to introduce easily any P system we have considered, by default, that:

1. Only finite alphabets A will be used, and the elements of A are *symbols*.
2. A *word* $\in A^*$ is a string of *symbols* of A. We will represent the empty word by ().
3. The membranes will be labeled with the *the first N natural numbers*, where N is the degree of the P system.
4. The skin membrane is labeled with 1.
5. A distinguished membrane is considered as the *output membrane*.
6. We will input the membrane structure of a P system as a list of *contain-pairs* $(i\ j)$, representing the relation "*membrane i contains membrane j*".
7. Every rule has a *word* as its antecedent, and a set of *actions* as its consequent. Only the last action could be "**delete**". The other ones have the form (*word target*).
8. A *target* could be "**here**", "**out**" or a membrane label.
9. If a membrane has $k > 0$ rules, then their labels go from 1 to k.
10. We represent the relation "*rule r runs before rule s*" by the *preference-pair* $(r\ s)$.
11. Every membrane contains a *word*, a list of rules, and a list of *preference-pairs*.

4.2 The Input Grammar

With the default settings provided above, any P system of degree N, over an alphabet A, is recognized by the *input grammar* defined as follows:

$$
\begin{array}{ll}
< input-ps > & ::= (A\ N < struct >< objects >< rules >< orders >< output >) \\
< struct > & ::= (< arc >< arc > \cdots < arc >) \\
< arc > & ::= (< memb-ref >< memb-ref >) \\
< memb-ref > & ::= 1 \mid 2 \mid 3 \mid \ldots \mid N \\
 & \\
< objects > & ::= [< word >< word > \stackrel{N}{\ldots} < word >] \\
< word > & ::= \forall\, w \in A^* \\
 & \\
< rules > & ::= [< memb-rules > \stackrel{N}{\ldots} < memb-rules >] \\
< memb-rules > & ::= (< rule >< rule > \cdots < rule >) \\
< rule > & ::= (< word > \to\ (< action > \cdots < action > \mathbf{delete})) \mid \\
 & \quad\ (< word > \to\ (< action > \cdots < action >))
\end{array}
$$

$$
\begin{array}{ll}
< action > & ::= (< word >< target >) \\
< target > & ::= \textbf{here} \;\; | \;\; \textbf{out} \;\; | \;\; < memb - ref > \\
\\
< orders > & ::= [< memb - or >< memb - or > .\overset{N}{.}. < memb - or >] \\
< memb - or > & ::= (< pref - pair >< pref - pair > \cdots < pref - pair >) \\
< pref - pair > & ::= (< rule - ref >< rule - ref >) \\
< rule - ref > & ::= 1 \mid 2 \mid 3 \ldots \\
< output > & ::= < memb - ref >
\end{array}
$$

Here, $(a\; b \ldots z)$ stands for a *list* (standard MzScheme *list*), and $[a\; b \overset{N)}{\ldots}\; z]$ stands for a *vector* of N elements (standard MzScheme *vector*).

5 The Parser/Compiler

The parser/compiler reads the input data describing a P system and analyzes: if they are *syntactically* correct according to the *input grammar*, if they define a *well defined* P system according to the chosen formalization, and, if no error appears, it returns the P system according to the proposed *internal grammar*.

Even if the input system is syntactically correct, we cannot conclude that any input data recognized by the *input grammar*, define a *well-defined* P system. In fact, it could happen that the structure $< struct >$ defined as a list of arcs $(< arc >^*)$ were not a *rooted tree* with root at membrane label 1; or, that there exists a membrane, such that the order relation $(< mem - or >)$ defined as a list of preference pairs $(< pref - pair >^*)$ were not a strict partial order.

The MzScheme sentence to execute the parser/compiler is:

```
(parser-ps N A <struct> <objects> <rules> <orders> <output>)
```

This process of parsing/compiling works as follows:

- The alphabet A is checked.
- The *rooted tree* μ, associated to the membrane structure, is created.
- For every membrane x, its objects are encoded as a *multiset* M_x, getting $M : V(\mu) \longrightarrow \mathbf{M}(A)$.
- Then, the *initial configuration*, $C_0 = (\mu, M)$, is built.
- Every rule, r, from the input data is encoded by $r^* = (d_r,\ v_r,\ \delta_r,\ \alpha_r)$, where α_r is set initially to `#f`. Then, one gets R^*.
- For every membrane x a strict partial order $\rho_x : R_x \times R_x \longrightarrow \{\#\mathtt{t}, \#\mathtt{f}\}$ is returned, with: $\rho_x(r,t) = \#\mathtt{t} \Leftrightarrow$ " r runs before s" at x. So, we obtain ρ.
- From R^* and ρ we have $\mathcal{R} = (R^*, \rho)$.
- The *output* membrane is checked to be in $V(\mu)$, getting i_0.

If no error occurs, the `parser-ps` procedure returns a *well-defined* P system $\Pi = (A, C_0, \mathcal{R}, i_0)$ as an element recognized by the *internal grammar* below.

5.1 Internal Grammar

The grammar to represent internally and to deal with P systems of degree N is the following:

$$\begin{array}{ll}
< ps > & ::= [< alph > ; < conf > ; < Rules > ; < orders > ; < output >]
\end{array}$$

$$\begin{array}{ll}
< alph > & ::= [\mathbf{a_1}, \mathbf{a_2}, \ldots, \mathbf{a_K}]
\end{array}$$

$$\begin{array}{ll}
< conf > & ::= [< tree > ; < multisets >]
\end{array}$$

$$\begin{array}{ll}
< tree > & ::= [< vertices > ; < arcs > ; < root >] \\
< vertices > & ::= \{< x > , \ \ldots \ , < x >\} \\
< arcs > & ::= \{< arc > , < arc > , \ \ldots \ , < arc >\} \\
< arc > & ::= [< x > ; < x >] \\
< x > & ::= \forall\, n \in \mathbf{N}^+ \mid n \leq N \\
< root > & ::= 1
\end{array}$$

$$\begin{array}{ll}
< multisets > & ::= [< multiset >< multiset > \ \stackrel{N)}{\ldots} \ < multiset >] \\
< multiset > & ::= [< nat >< nat > \ \stackrel{\mathbf{K})}{\ldots} \ < nat >] \\
< nat > & ::= \forall\, n \in \mathbf{N}
\end{array}$$

$$\begin{array}{ll}
< Rules > & ::= [< rules >< rules > \ \stackrel{N)}{\ldots} \ < rules >] \\
< rules > & ::= [< rule >< rule > \ldots < rule >] \\
< rule > & ::= [< anteced > ; < actions > ; < dissol > ; < app-tag >] \\
< anteced > & ::= < multiset > \\
< actions > & ::= (< action >< action > \cdots < action >) \\
< action > & ::= [< multiset > ; < target >] \\
< target > & ::= \mathbf{here} \quad | \quad \mathbf{out} \quad | \quad < x > \\
< dissol > & ::= \#\mathtt{t} \quad | \quad \#\mathtt{f} \\
< app-tag > & ::= \#\mathtt{t} \quad | \quad \#\mathtt{f}
\end{array}$$

$$\begin{array}{ll}
< orders > & ::= [< test > < test > \ \stackrel{N)}{\ldots} \ < test >] \\
< test > & ::= \lambda : rules \times rules \longrightarrow \{\#\mathtt{t}, \#\mathtt{f}\}
\end{array}$$

$$\begin{array}{ll}
< output > & ::= < x >
\end{array}$$

6 The Simulator

Once the *parsing/compiling* task is finished, we have a *well-defined* P system, namely $\Pi = (A, C_0, \mathcal{R}, i_0)$, and we have to generate the computation tree $\mathbf{Comp}(\Pi)$. To do that we use the procedure `configurations`:

$$\Pi \quad \xrightarrow{\texttt{configurations}} \quad \mathbf{Comp}(\Pi)$$

We get the computation tree $\mathbf{Comp}(\Pi)$ through the MzScheme sentence `(configurations` Π *level*`)`.

The procedure `configurations` is based on the `breadth-expansion-tree` procedure that, starting from the initial configuration C_0, generates level by level the *computation tree*. It uses the auxiliary procedures `applicability-vectors`, `tag-rules` and `apply-matrix`. Here we present a brief outline. We will give in the next sections a detailed description of every one.

The operators to compute the *successor configurations* of a given configuration, C, are the *applicability matrices*. The process to generate the elements of $\mathbf{M_{Ap}}(C)$ works as follows:

- R^C (that is, the *tagged-rules* of x to C) is obtained by the `tag-rules` procedure. For every rule $r^* = (d_r, v_r, \delta_r, \alpha_r) \in R^*$, it sets $\alpha_r = \#\texttt{t}$ iff r is applicable to C.
- Every R^C_x, for every membrane x in C, is easily obtained from R^C.
- Every $\mathbf{Ap}(x, C)$ (that is, the *applicability vectors* of membrane x in C) is generated from R^C_x, by means of the `applicability-vectors` procedure.
- Finally, $\mathbf{M_{Ap}}(C)$ is constructed as a cartesian product from the set of *applicability vectors* $\mathbf{Ap}(x, C)$, of every membrane x in C.

$$\begin{array}{ccccccc}
 & & \overset{x_1}{\nearrow} & R^C_{x_1} & \xrightarrow{\texttt{applicability-vectors}} & \mathbf{Ap}(x_1, C) \searrow & \\
C \xrightarrow{\texttt{tag-rules}} & R^C & \xrightarrow{\texttt{x}_2} & R^C_{x_2} & \xrightarrow{\texttt{applicability-vectors}} & \mathbf{Ap}(x_2, C) \rightarrow & \mathbf{M_{Ap}}(C) \\
 & & \overset{x_3}{\searrow} & R^C_{x_3} & \xrightarrow{\texttt{applicability-vectors}} & \mathbf{Ap}(x_3, C) \nearrow &
\end{array}$$

Then, every $P \in \mathbf{M_{Ap}}(C)$ is applied to C to obtain the *successor configuration* $P(C)$. To do that the `apply-matrix` procedure is used.

$$\begin{array}{cccc}
 & \nearrow P_1 \in \mathbf{M_{Ap}}(C) & \xrightarrow{\texttt{apply-matrix}} & C_1 = P_1(C) \\
\mathbf{M_{Ap}}(C) \rightarrow & P_2 \in \mathbf{M_{Ap}}(C) & \xrightarrow{\texttt{apply-matrix}} & C_2 = P_2(C) \\
 & \searrow P_3 \in \mathbf{M_{Ap}}(C) & \xrightarrow{\texttt{apply-matrix}} & C_3 = P_3(C)
\end{array}$$

6.1 The Breadth-Expansion-Tree Procedure

This procedure is based on a *dynamic* breadth-search scheme; this means that for every node of the tree to be built, the applicable operators are generated *dynamically*.

To start, the `breadth-expansion-tree` procedure needs: (1) an *initial node*, n_0, (2) a test `final-node?` to check if a node n is or not a *final node*, (3) a function `generate-op`, that, taking a node n, returns the set of operators Op_n to be applied to n and, finally, (4) another function `apply-op` that, taking a node n and an operator $op \in Op_n$, returns the *successor node* of n by this operator op.

The `breadth-expansion-tree` procedure expands the tree and returns the set of *final nodes*.

```
Procedure breadth-expansion-tree (n_0 final-node? generate-op
apply-op)
final-nodes ← {}
open-nodes ← {n_0}
Repeat until open-nodes = ∅ do
n ← the first node in  open-nodes
succ_n ← {}
If (final-node?  n) = #t
   then final-nodes ← {n}∪ final-nodes
   else
         Op_n ← (generate-op n)
         For every op ∈ Op_n do
             suc ← (apply-op op n)
             If suc ≠ #f ∧ suc ∉ open-nodes then
               succ_n ← succ_n ∪ {suc}
open-nodes ← (open-nodes−{n}) ∪ succ_n
Return final-nodes
```

The procedures `configurations` and `applicability-vectors`, to generate configurations and applicability vectors, respectively, are based on this procedure.

6.2 The Configurations Procedure

For a given P system $\Pi = (A, C_0, \mathcal{R}, i_0)$, we generate $\mathbf{Comp}(\Pi)$ (until a level given by the user), through the MzScheme sentence (`configurations` Π *level*). This procedure works as follows:

1. It starts defining locally:
   - The *node-structure* as $<$ `node` $>$ $::= [C; R^C; path_C]$, where C ia a configuration; R^C, the *tagged-rules* for C; and $path_C$, the list of operators applied to reach the actual node from the initial one.
   - The `final-node?` test. A node $n = [C; R^C, path]$ is a *final node* if either it is a *halting node*, or the *path* length has reached the value of *level*.
   - The `generate-op` function. It takes a node $n = [C; \mathbf{R}^C; path]$ and returns the *applicability matrices* $\mathbf{M_{Ap}}(C)$. It needs the procedure `applicability-vectors`.
   - Finally, the procedure `apply-op`, which, taking a node $n = [C; R^C, path]$ and an applicability matrix $P \in \mathbf{M_{Ap}}(C)$, returns the *successor node* $n' = [C'; R^{C'}; P \cup path]$. It needs the procedures `apply-matrix` and `tag-rules`.
2. Then, it builds the *init-node*: $n_0 = [C_0; R^{C_0}; ()]$, making use of the procedure `tag-rules` to get R^{C_0}.

3. It expands the tree through the sentence:
   (breadth-expansion-tree
   $init-node$ final-node? generate-op apply-op)
4. Finally, it returns the list of *final-nodes* $[C;R^C;path_C]$.

Procedure configurations (*Π level*)

1. *Local definitions*
   $< \texttt{node} > \ ::= [C;R^C;path_C]$
   $\texttt{final}-\texttt{node?} ::= \lambda_1 : < \texttt{node} > \longrightarrow \{\#\texttt{t}, \#\texttt{f}\}$
   $\texttt{generate}-\texttt{op} ::= \lambda_2 : < \texttt{node} > \longrightarrow \mathbf{M_{A_p}}(C) = (P_1, P_2, \dots)$
   $\texttt{apply}-\texttt{op} ::= \lambda_3 : k \times < \texttt{node} > \longrightarrow [C';R^{C'};path_{C'}]$
   with $C' = P_k(C)$
   and, $path_{C'} = P_k \cup path_C$
2. *The initial node*
   $R^{C_0} \leftarrow (\texttt{tag}-\texttt{rules}\ \texttt{C}_0\ \texttt{R}^*\ \rho)$
   $path_{C_0} \leftarrow ()$
   $n_0 \leftarrow [C_0;R^{C_0};path_{C_0}]$
3. *The final-nodes*
   final-nodes $\leftarrow$ (breadth-expansion-tree
   n_0 final-node? generate-op apply-op)
4. Return *final-nodes*

Notes:

- $\lambda_1([C;R^C;path_C]) = \#\texttt{t} \leftrightarrow (\alpha_r = \#\texttt{f}\ \forall r \in R^C) \vee |path_C| = level$
- λ_2 uses applicability-vectors procedure to get $\mathbf{M_{A_p}}(C)$.
- λ_3 uses apply-matrix procedure to get $C' = P(C)$ and then, tag-rules to get $R^{C'}$.
- Every node $[C;R^C;path_C] \in$ *final-nodes*, contains all the information we need about the *computation tree*. Particularly,
  - If for every $r \in R^C$ is $\alpha_r = \#\texttt{f}$, then C is a *halting configuration*, and $path_C$ is a *halting computation* of *Π*.
  - Otherwise, C is a *non-halting configuration*, and the branch $path_C$ could be extended further than the prefixed *level*.

6.3 The Applicability-Vectors Procedure

To generate the *applicability vectors* for a membrane x in C, we only need M_x and $D = [d_1, d_2, \dots, d_{s_x}]$, where M_x is the *multiset* of x, and d_r $(r = 1, 2, \dots, s_x)$ is the antecedent of the *tagged*-rule r in R^C_x, provided that $\alpha_r = \#\texttt{t}$. (*Note*: if $\alpha_r = \#\texttt{f}$, then we take $d_r = \#\texttt{f}$.) We generate $\mathbf{Ap}(x, C)$ through the MzScheme sentence: (applicability-vectors M_x D). The procedure works as follows:

1. It starts defining locally:
   - The *node-structure* as $< \texttt{node} > ::= [m; V]$, where m is a multiset, and $V = [v_1, v_2, \ldots, v_{s_x}]$.
   - The `final-node?` test. A node $n = [m; V]$ is a final node if $\forall\, d_r \in D$ $(d_r = \#\texttt{f} \vee m < d_r)$.
   - The `generate-op` function. It returns the operators list $(d_1, d_2, \ldots, d_{s_x})$.
   - The `apply-op` procedure. From a node $n = [m; V]$, and an operator $d_r \neq \#\texttt{f}$, it returns the *successor node* $n' = [m'; V']$, with $m' = m - d_r$, $v'_r = v_r + 1$, and $v'_j = v_j$, $\forall\ j \neq r$. If $d_r = \#\texttt{f}$, then it returns `#f`.
2. Then, it builds the *init-node*: $n_0 = [M_x; [0, 0, \overset{s_x}{\ldots}, 0]]$.
3. It expands the tree through the sentence:
   `(breadth-expansion-tree`
   $init - node$ `final-node? generate-op apply-op)`y
4. Finally, it returns the *applicability vector* V of every final node $[m; V]$.

`Procedure applicability-vectors` $(M_x\ D)$

1. *Local definitions*
   $< \texttt{node} > \quad ::= [m; V] \quad$;whith $V = [v_1, v_2, \ldots, v_{s_x}]$
   $\texttt{final} - \texttt{node}? ::= \lambda_1 : < \texttt{node} > \longrightarrow \{\#\texttt{t}, \#\texttt{f}\}$
   $\texttt{generate} - \texttt{op} ::= \lambda_2 : < \texttt{node} > \longrightarrow (d_1, d_2, \ldots, d_{s_x})$
   $\texttt{apply} - \texttt{op} \quad ::= \lambda_3 : r \times < \texttt{node} > \longrightarrow [m'; V']$
   with, $m' = m - d_r$, $V' = [v'_1, v'_2, \ldots, v'_{s_x}]$,
   being, $v'_r = v_r + 1$ but, $v'_j = v_j\ \forall j\ \neq r$
2. *The initial node*
   $m_0 \leftarrow M_x$
   $V_0 \leftarrow [0, 0, \overset{s_x}{\ldots}, 0]$
   $n_0 \leftarrow [m_0; V_0]$
3. *The final-nodes*
   final-nodes $\leftarrow$ (`breadth-expansion-tree`
   n_0 `final-node? generate-op apply-op`)
4. `Returns` *the vector* V *of every node* $[m; V]$ *of final-nodes*

Notes:

- Every v_r *counts* the times the rule r could be applied.
- $\lambda_1([m; V]) = \#\texttt{t} \leftrightarrow \forall\, d_r \in D(d_r = \#\texttt{f} \vee m < d_r)$
- $\lambda_3(r, [m; V]) = \#\texttt{f}$ if $d_r = \#\texttt{f}$

6.4 The Tag-Rules Procedure

The `tag-rules` procedure *updates* the *app-tag* α_r of those rules r of R^* that are applicable to a given configuration $C = (\mu, M)$. The MzSheme sentence (`tag-rules` $C\ R^*\ \rho$) returns R^C. The procedure works as follows:

1. It starts getting the degree, N, of Π.
2. Then, its work is based on an *external* and an *internal loop*, to go through the membranes and through the rules of every membrane, respectively.
   – The *external loop* generates R_x^C, for every $x = 1, 2, \ldots, N$, and, once it is finished, it builds $R^C = (R_1^C, R_2^C, \ldots, R_N^C)$. If $x \notin V(\mu) \vee M_x = \emptyset$, then $R_x^C = R_x^*$, otherwise, R_x^C has to be generated by the *internal loop*.
   – The *internal loop* generates R_x^C for a given $x \in V(\mu)$. It checks the applicability of every rule $r_{x,j} \in R_x^*$ to C, it changes $\alpha_{x,j}$ from $\#\mathtt{f}$ to $\#\mathtt{t}$ if so, and it obtains the *tagged-rule* $r_{x,j}^C$; finally, it builds and returns to the *external loop*, $R_x^C = (r_{x,1}^C, r_{x,2}^C, \ldots, r_{x,s_x}^C)$.
3. It returns R^C.

`Procedure tag-rules` $(C\ R^*\ \rho)$
$N \leftarrow$ *length of* ρ
`For every` $x = 1, 2, \ldots, N$ `do`
`If` $x \notin V(\mu) \vee M_x = 0$ `then` $R_x^C \leftarrow R_x^*$
`else`
`For every` $j = 1, 2, \ldots, s_x$ `do`
    `If` $r_{x,j}$ *is not semi-applicable* `to` C `then` $r_{x,j}^C \leftarrow r_{x,j}^*$
    `else`
    `If` $\exists k < j \mid \alpha_{x,k}^C = \#\mathtt{t}\ \wedge\ \rho(k,j) = \#\mathtt{t}$ `then` $r_{x,j}^C \leftarrow r_{x,j}^*$
        `else` $r_{x,j}^C \leftarrow (d_{x,j}, v_{x,j}, \delta_{x,j}, \#\mathtt{t})$
    `If` $\alpha_{x,j} = \#\mathtt{t}$ `then`
        `For every` $k < j \mid \alpha_{x,k}^C = \#\mathtt{t} \wedge \rho(j,k) = \#\mathtt{t}$ `do`
            $\alpha_{x,k}^C \leftarrow \#\mathtt{f}$
$R_x^C \leftarrow (r_{x,1}^C, r_{x,2}^C, \ldots, r_{x,s_x}^C)$
$R^C \leftarrow (R_1^C, R_2^C, \ldots, R_N^C)$
`Return` R^C

6.5 The Apply-Matrix Procedure

The `apply-matrix` procedure computes one *transition step*, $C' = P(C)$, from a configuration, $C = (\mu, M)$, and an *applicability matrix*, $P \in \mathbf{M_{Ap}}(C)$. The MzSheme sentence is `(apply-matrix` $C\ \ P\ \ R^C$`)`. It works in two steps:

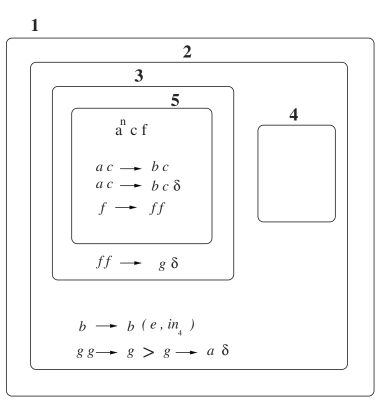
1. For every membrane x in C and every rule $r_{x,j}$ in R_x^C, provided $P_{x,j} \neq 0$:
   – $r_{x,j}$ is applied $P_{x,j}$ times *without dissolution*. So, some objects of membrane x are consumed, and maybe itself and/or, its *father* and *children* receive some objects. A more internal loop identifies the *target* where every *action* of the rule sends its objects,
   – then, if $r_{x,j}$ is a dissolution rule, x is stored in Δ as a node to be dissolved.

2. Then, we visit the nodes of μ in a *bottom-up ordered* way, the nodes kept on Δ are dissolved. Every dissolved node sends its objects (and *children*) to its *father* and *disappears* from μ.

Procedure apply-matrix $(C\ \ P\ \ R^C)$
$M' \leftarrow M$
$\mu' \leftarrow \mu$
$\Delta \leftarrow \{\}$
For every $x \in V(\mu')$ do
If $x \neq root(\mu')$ then $f_x \leftarrow$ *the father of* x *in* μ' else $f_x \leftarrow$ #f
For every $r_{x,j} = (d_{x,j}, v_{x,j}, \delta_{x,j}, \alpha_{x,j}) \in R^C_x$ do
    If $P_{x,j} \neq 0$ then
     $M'_j \leftarrow M'_j - P_{x,j} \otimes d_{x,j}$
     For every $action = (m, tar) \in v_{x,j}$ do
         If $tar \notin V(\mu')$ then
           If $tar = here$ then $tar \leftarrow x$ else $tar \leftarrow f_x$
         If $tar \neq$ #f then $M'_{tar} \leftarrow M'_{tar} + P_{x,j} \otimes m$
     If $\delta_{x,j} =$ #t then $\Delta \leftarrow \Delta \cup \{x\}$
$nodes \leftarrow$ *the bottom-up ordered* $V(\mu')$
For every $x \in nodes$ do
If $x \in \Delta$ then
   $M'_{f_x} \leftarrow M'_{f_x} + M'_x$
   $M'_x \leftarrow \emptyset$
   $\mu' \leftarrow$ delete-node (μ', x)
Return $C' = (\mu', M')$

7 A Complete Example: Generating Squares $1^2, 2^2, \ldots, n^2$

Finally we present here a complete example to illustrate the way our simulator should be used. The P system to be considered is the following one:



7.1 The Input Data

First of all, we have to input the data describing the P system. We do that defining the different elements: A, N, *struct*, *output*, *objects*, *rules*, and *orders*. As we need the symbol $a^n cf$, for the given n, this one is generated by the auxiliar procedure `generate-symbol`. The MzScheme sentence (`sq1` n), assigns the respective value to every compound, and invokes the *parser/compiler*.

```
> (define sq1
   (lambda (n)
    (let ((N 5)
          (A '(a b c e f g))
          (o_m 4)
          (struct '((1 2) (2 3) (2 4) (3 5)))
          (objects
           (vector () () () () (generate-symbol
                   (list 'a n) '(c 1) '(f 1))))
          (rules
           (vector
            '()
            '((b -> ((b here) (e 4)))
              (gg -> ((g here)))
              (g -> ((a here) delete)))
            '((ff -> ((g here) delete)))
            '()
            '((ac -> ((bc here)))
              (ac -> ((bc here) delete))
              (f -> ((ff here))))))
          (orders
           (vector '() '((2 3)) '() '() '())))
      (parser-ps N A struct objects rules orders o_m))))
```

7.2 The Parser-Compiler

The *parser/compiler* returns the internal representation of the P system, and displays it in a readable way. So, if $n = 4$ the sentence

```
(define ps (sq1 4))
```

defines, if no error occurs, `ps` as the representation to be used together with the `configurations` procedure.

7.3 Configurations

Finally, using the procedure `configurations` to expand the *computation tree*, we obtain *all configurations* until the given level. In particular, with an appropriate level we get all the *final configurations*. In the previous example it is enough to use 9 as depth level.

```
> (configurations ps 9)
TREE:      ((1 2 3 4 5) ((1 2) (2 3) (2 4) (3 5)) 1) ;a non-halting
CONTENTS:                                            ;configuration
  Membrane 1 and Membrane 4:
    Multiset:  #(0 0 0 0 0 0)
    Applic-Rules: #()
  Membrane 2:
    Multiset:  #(0 0 0 0 0 0)
    Applic-Rules: #(#f #f #f)
  Membrane 3:
    Multiset:  #(0 0 0 0 0 0)
    Applic-Rules: #(#f)
  Membrane 5:
    Multiset:  #(0 4 1 0 512 0)
    Applic-Rules: #(#f #f #t)   ;the third rule could be applied
TREE:      ((1 4) ((1 4)) 1)    ;a halting configuration
CONTENTS:
  Membrane 1:
    Multiset:  #(1 4 1 0 0 0)
    Applic-Rules: #()
  Membrane 4:
    Multiset:  #(0 0 0 16 0 0)
    Applic-Rules: #()
TREE:      ((1 4) ((1 4)) 1)   ;a halting configuration
CONTENTS:
  Membrane 1:
    Multiset:  #(2 3 1 0 0 0)
    Applic-Rules: #()
  Membrane 4:
    Multiset:  #(0 0 0 9 0 0)
    Applic-Rules: #()
TREE:      ((1 4) ((1 4)) 1)   ;a halting configuration
CONTENTS:
  Membrane 1:
    Multiset:  #(3 2 1 0 0 0)
    Applic-Rules: #()
  Membrane 4:
    Multiset:  #(0 0 0 4 0 0)
    Applic-Rules: #()
TREE:      ((1 4) ((1 4)) 1)   ;a halting configuration
CONTENTS:
  Membrane 1:
    Multiset:  #(4 1 1 0 0 0)
    Applic-Rules: #()
  Membrane 4:
    Multiset:  #(0 0 0 1 0 0)
    Applic-Rules: #()

Output Membranes:   (() eeeeeeeeeeeeeeee eeeeeeeee eeee e)
```

8 Conclusions

Up to now there is no implementation of P systems with a *practical usefulness* that allows the researchers to test and improve the abstract designs they make. The simulation of P systems by conventional programming languages can be considered not only as a practical approach to this computing model, but also as an useful way to understand and improved the P systems designed to solve real problems. We think that, because of the *standard* grammar it uses, the program presented here can be used both as a research tool and a teaching tool, allowing to see the way the P system evolves along its running. The program has been developed in such a way that it could be improved to simulate different variants of P systems. In a future work a graphical interface will be added, to make easier the interaction with the user.

References

1. Gh. Păun, Computing with membranes, *Journal of Computer and System Sciences*, 61, 1 (2000), 108–143, and *Turku Center for Computer Science-TUCS Report* No 208, 1998 (www.tucs.fi).
2. Gh. Păun, G. Rozenberg, A guide to membrane computing, *Theoretical Computer Science*, 287, 1 (2002), 73–100.
3. M.J. Pérez–Jiménez, F. Sancho-Caparrini. A formalization of transition P systems. *Fundamenta Informaticae*, 49, 1-3 (2002), 261–272.
4. M.J. Pérez–Jiménez, F. Sancho-Caparrini. Verifying a *P* system generating squares. *Romanian Journal of Information Science and Technology*, 5, 2–3 (2002), 181–191.
5. MzScheme Home Page. `http://www.cs.rice.edu/CS/PLT/packages/mzscheme/`