

P Systems Generating Trees

Rudolf FREUND¹ Marion OSWALD¹ Andrei PĂUN²

¹ Faculty of Informatics
Vienna University of Technology
Favoritenstr. 9, A-1040 Wien, Austria
E-mail: {rudi,marion}@emcc.at

² Department of Computer Science
Louisiana Tech University
Ruston, Louisiana
E-mail: apaun@latech.edu

Abstract

We consider P systems with active membranes, but without polarizations, yet with using membrane division and membrane generation, but as the result of a halting computation we do not take the terminal string generated in a designated output membrane, instead we consider the resulting tree representing the membrane structure of the final configuration as its result. We show that each recursively enumerable tree language can be obtained in that way generated by P systems with active membranes working on strings.

1 Introduction

In [11] membrane systems (then called P systems) were introduced as bio-inspired computing devices that work in a parallel and distributed way (see [13] for a comprehensive overview and [9] for current developments in the area).

One main feature of membrane systems (P systems) is their membrane structure. So far, P systems have usually been considered as devices for generating or accepting multisets of symbol objects or string objects to be found in a designated output membrane in the final configuration of a halting configuration. In this paper we now consider the membrane structure itself as the result of a successful computation, i.e., we obtain a set of trees being computed as the membrane structures in the final configurations of halting computations.

P systems with active membranes were introduced (e.g., see [13]; for variants solving NP complete problems, e.g., see [7], [10], [14]) with rules for

- (a) rewriting multisets;
- (b) introducing objects into membranes;
- (c) sending objects out of membranes;

- (d) dissolving membranes;
- (e) dividing elementary membranes;
- (f) dividing non-elementary membranes.

All these rules were associated with membranes not only having a specific label, but also having assigned an electrical charge (also called polarization), which could be $+$, 0 , and $-$. The rules of the forms (b) and (c) involving membranes could change the polarization(s) of the involved membrane(s), but never changed the label(s) of the membranes.

For generating tree languages, in this paper we use a special variant of P systems with active membranes with rules especially including division of elementary membranes, but without changing polarizations, which in fact means that we simply may forget the polarizations of membranes; moreover, we shall also use membrane generation rules as introduced in [4]. In contrast to the original model, where P systems with active membranes were working on symbol objects, in this paper (as in [4]) we consider P systems with active membranes working on string objects.

The rest of the paper is organized as follows: In the following section, we first recall some basic definitions from the theory of formal languages, give the definitions of deterministic register machines and then recall the most important results concerning the (universal) computational power of these devices. In the third section, we specify P systems with active membranes working on string objects; as we shall show as the main result of this paper, the model of P systems we chose to use allows us to generate each recursively enumerable tree language by interpreting the membrane structure of final configurations in halting computations as tree objects. A short summary and an outlook to future research conclude the paper.

2 Preliminaries

Before proceeding to a formal description of P systems, we first fix some basic notations in this section. For more notions as well as basic results from the theory of formal languages, the reader is referred to [1], [6], and [15].

For an alphabet V , by V^* we denote the free monoid generated by V under the operation of concatenation; the *empty string* is denoted by λ , and $V^* \setminus \{\lambda\}$ is denoted by V^+ . Any subset of V^+ is called a λ -free (*string*) *language*. Moreover, by \mathbf{N} we denote the set of positive integers (or natural numbers). The family of λ -free recursively enumerable languages is denoted by RE , the family of sets of λ -free recursively enumerable languages over a one-letter alphabet by NRE (in fact, this family corresponds to the set of recursively enumerable sets of natural numbers $PsRE$, the family of Parikh sets of the languages in NRE).

A (*string*) *grammar* is a quadruple $G = (V_N, V_T, P, A)$, where V_N and V_T are finite sets of *nonterminal* and *terminal symbols*, and $V_N \cap V_T = \emptyset$, P is a finite set of *productions* $\alpha \rightarrow \beta$ with $\alpha \in V^+$ and $\beta \in V^*$, where $V = V^+ \cup V^*$, and $A \in V_N$

is the *axiom*. For $x, y \in V^*$ we say that y is *directly derivable* from x in G , denoted by $x \Rightarrow_G y$, if and only if for some $\alpha \rightarrow \beta$ in P and $u, v \in V^*$ we get $x = u\alpha v$ and $y = u\beta v$. Denoting the reflexive and transitive closure of the derivation relation \Rightarrow_G by \Rightarrow_G^* , the *language generated by G* is $L(G) = \{w \in V_T^* \mid A \Rightarrow_G^* w\}$. A production $\alpha \rightarrow \beta$ is called *context-free* if $\alpha \in V_N$. If G contains only context-free rules it is called a *context-free grammar*. A context-free grammar is said to be in *Chomsky normal form*, if it contains only rules of the form $A \rightarrow BC$ and $A \rightarrow a$ where $A, B, C \in V_N$ and $a \in V_T$ (a context-free grammar in Chomsky normal form usually is also assumed to be *reduced*, which means that every non-terminal symbol $A \in V_N$ can be reached from the start symbol S , i.e., $S \Rightarrow_G^* uAv$ for some $u, v \in (V_N \cup V_T)^*$, and that that from every non-terminal symbol $A \in V_N$ we can derive a terminal word, i.e., $A \Rightarrow_G^* w$ for some $w \in V_T^*$).

A *deterministic register machine* is a construct $M = (n, R, l_0, l_h)$, where n is the number of registers, R is a finite set of instructions injectively labelled with elements from a given set $lab(M)$, l_0 is the initial/start label, and l_h is the final label.

The instructions are of the following forms:

- $l_1 : (add(r), l_2)$
Add 1 to the contents of register r and proceed to the instruction (labelled with) l_2 . (We say that we have an ADD instruction.)
- $l_1 : (sub(r), l_2, l_3)$
If register r is not empty, then subtract 1 from its contents and go to instruction l_2 , otherwise proceed to instruction l_3 . (We say that we have a SUB instruction.)
- $l_h : halt$
Stop the machine. The final label l_h is only assigned to this instruction.

A register machine M is said to accept a natural number n if and only if, starting with the instruction with label l_0 and with register one containing the number n and all other registers containing the number 0, the machine stops (it reaches the instruction $l_h : halt$) with all registers containing the number 0.

The register machines are known to be computationally universal, equal in power to deterministic Turing machines (e.g., see [8]): they accept exactly the sets of natural numbers which can be accepted by Turing machines, that is, the family *PsRE*. Even more specifically, register machines can accept string languages $L \subseteq T^*$ accepted by Turing machines (i.e., the family *RE*) in the following way: For each $w \in T^*$ the register machine accepts the number $2^{g_{z+1}(w)}$ if and only if the string w is accepted by the Turing machine, where $g_{z+1}(w)$ is the numerical z -ary encoding of w at base $z + 1$, z being the number of symbols in T (e.g., see [2], [3], [5]).

Without loss of generality, in the proofs of the following section we will assume that in each SUB instruction $l_1 : (sub(r), l_2, l_3)$ the labels l_1, l_2, l_3 are mutually distinct: For instance, to achieve this goal, we replace each SUB instruction $l_1 : (sub(r), l_2, l_3)$ by the instruction $l_1 : (sub(r), l'_2, l''_3)$ and add the instructions $l'_2 : (add(n + 1), l'''_2)$, $l'''_2 : (sub(n + 1), l_2, l'_2)$, $l''_3 : (add(n + 1), l'_3, l''_3)$, $l'_3 : (sub(n + 1), l_3, l''_3)$,

where $n + 1$ is a new register (this can be the same for all SUB instructions we start from), and all primed labels are distinct and different from the initial labels.

3 P Systems with Active Membranes and String Objects as Tree Generating Devices

A *P system with active membranes and string objects* (for sake of simplicity, we often will refer to such a device as a *P system* in the following) is a construct

$$\Pi = (V, T, H, \mu, w_1, \dots, w_m, R),$$

where: $m \geq 1$; V is an alphabet (the *total alphabet* of the system); $T \subseteq V$ (the *terminal* alphabet); H is a finite set of *labels* for membranes (in the following we will always label the skin membrane by 0); μ is a *membrane structure*, consisting of m membranes (represented by matching pairs of brackets), labelled with elements of H ; w_1, \dots, w_m are finite multisets of words over V (describing the *initial objects in the membranes* placed in the m regions of μ); R is a finite set of *rules*, of the following forms:

- (sa) $[A]_h \longrightarrow [v]_{h'}$, where $A \in V$, $v \in V^*$, $h, h' \in H$ (a letter A is rewritten into v in a membrane labelled with h thereby changing its label to h');
- (sb) $A[]_h \longrightarrow [v]_{h'}$, where $A \in V$, $v \in V^*$, $h, h' \in H$ (a letter A is rewritten into v and then the resulting word is sent into a membrane labelled with h thereby changing its label to h'),
- (sc) $[A]_h \rightarrow v[]_{h'}$, where $A \in V$, $v \in V^*$, $h, h' \in H$ (an object A is rewritten into v and then the resulting string is sent out of the membrane labelled with h thereby changing its label to h'),
- (sd) $[A]_h \rightarrow v$, where $A \in V$, $v \in V^*$, $h \in H$ (an object A is rewritten into v at the same time dissolving the surrounding membrane labelled with h),
- (se) $[A]_h \rightarrow [v]_{h'}[v']_{h''}$, where $A \in V$, $v, v' \in V^*$, $h, h', h'' \in H$
 (2-division rule for elementary membranes; in reaction with an object, the membrane is divided into two membranes with, possibly, different labels; the word containing the letter A specified in the rule is replaced in the corresponding words in the two new membranes by possibly new substrings v, v' ; at the same time, all the objects except for the word containing the letter A that started the membrane division are duplicated into the two new membranes),
- (sg) $[A]_h \longrightarrow [[v]_{h''}]_{h'}$, where $A \in V$, $v \in V^*$, $h, h', h'' \in H$ (a letter A in a membrane labelled with h , by changing its label to h' , is rewritten into v and then the resulting word is sent into a newly generated inner membrane labelled with h'').

We refer to [12], [13], and [14] for a more precise definition of the way originally P systems with active membranes were supposed to work; here we only informally

describe the way of passing from one configuration of the system to the next one. The difference between the original model and our proposed model mainly lies in the fact that we work on strings rather than with symbols (therefore, the types of rules carry the additional marking s); moreover, in describing the membrane structure as well as the rules we only label the right-hand brackets of the matching pairs.

The rules are applied in a maximally parallel manner, yet with the following restrictions: The rules are applied “from bottom up”, in one step, starting with the rules of the innermost region and, then, going up level by level until the region of the skin membrane is reached. In each region, all strings which can evolve using a rule of the forms (sa) , (sb) , (sc) , (sg) can and have to evolve if they do not change the label of the surrounding/involved membrane, yet (at most) only one string (afterwards!) may take a rule changing the label of the surrounding/involved membrane or dissolve it or divide it, i.e., rules of type (sa) , (sb) , (sc) , (sg) can be used in parallel as long as they do not change the label of the membrane they affect. At most one rule of type (se) can be applied to one selected string after the evolution of all other strings in the membrane region not changing the membrane label or dissolving the membrane; if a membrane with label h is divided by a rule of type (se) , which involves a word containing letter A , then all other words in membrane h are copied into each of the resulting membranes.

The rules associated with a membrane labelled with h are used for all copies of this membrane; it does not matter whether the membrane is an initial one or it was obtained by membrane division or membrane generation. The skin membrane can never divide (nor dissolve). By (sx_0) , $x \in \{a, b, c, d, e, g\}$, we denote the types of rules corresponding to (sx) , if the rules do not change the labels of the involved membranes (i.e., in any case we have $h = h'$).

In this paper, as the result of a halting computation (no rule can be used in the last configuration) we consider the tree represented by the membrane structure of the final configuration: the skin membrane represents the root of the tree and is always labelled by 0 (remember that the skin membrane cannot be divided); if a membrane labelled by i is enclosed by a membrane labelled by j , in the tree the edge from the corresponding node representing this membrane labelled by j to the corresponding node representing this membrane labelled by i gets the label i . In what concerns the labels of nodes, we might consider several variants: First, we could neglect the contents of the membrane regions and thus consider trees without node labels. Yet in the following, we shall consider trees having labels at the nodes, too; hence, in the final configuration the contents of each membrane region has to be a singleton $a \in T$ which then is taken as the label of the corresponding node representing this membrane (if this condition is not fulfilled, this final configuration does not contribute to the tree language).

By $L(\Pi)$ we denote the tree language generated as described above by a P system Π . If a computation goes forever, then it does not contribute to the set $L(\Pi)$.

Now let D be a non-empty subset of $\{sx, sx_0 \mid x \in \{a, b, c, d, e, g\}\}$. Then, by $LPT(D)$ we denote the family of tree languages generated by P systems with active membranes and string objects as defined above using only rules of types from D .

Example 1. Consider a context-free grammar in Chomsky normal form $G = (V_N, V_T, P, A)$. Then we construct a P system

$$\Pi = (V_N \cup V_T, V_T, H, []_0, S, R)$$

with the following rules in R :

- $[S]_0 \rightarrow [a]_0$
for $S \rightarrow a \in P, a \in V_T$;
- $[S]_0 \rightarrow [[S']_1]_0, [S']_1 \rightarrow [B]_1 [C]_2$
for $S \rightarrow BC \in P, B, C \in V_N$;
- $[A]_h \rightarrow [[A']_1]_h, [A']_1 \rightarrow [B]_1 [C]_2$
for $h \in \{1, 2\}, A \rightarrow BC \in P, A, B, C \in V_N$;
- $[A]_h \rightarrow [a]_h$
for $h \in \{1, 2\}, A \rightarrow a \in P, A \in V_N, a \in V_T$.

As we can assume G to be reduced, termination of a computation in Π already means that we obtain the resulting membrane structure in the final configuration as our terminal tree result. Obviously, the trees we obtain are the derivation trees of G .

Now let TRE denote the family of recursively enumerable tree languages over finite alphabets of labels for nodes and edges. Then, as the main result of this paper, we can show the following theorem:

Theorem 2. Each recursively enumerable tree language over finite alphabets of labels for nodes and edges can be generated by a P system; more specifically,

$$LPT(\{sb_0, sc_0, sd_0, se_0, sg_0\}) = TRE.$$

Proof (Sketch). We only sketch the proof for the inclusion

$$LPT(\{sb_0, sc_0, sd_0, se_0, sg_0\}) \supseteq TRE,$$

i.e., given a tree language L_t (over the alphabet T for the labels of the nodes and the alphabet $\{j \mid 1 \leq j \leq k\}$ for the labels of the edges) we describe the main ingredients of a P system

$$\Pi = (V, T, H, []_0, X_0, R)$$

such that $L_t = L(\Pi)$. Let N denote the set of non-terminal symbols in Π , i.e., $N = V - T$. The skin membrane is always labelled by 0, hence $H \supseteq \{j \mid 0 \leq j \leq k\}$. The additional labels as well as non-terminal symbols will become obvious from the description of the rules in R given below.

The main idea of the proof now is to generate the membrane structure corresponding to an arbitrary tree over T and $\{j \mid 1 \leq j \leq k\}$ and in parallel to encode

the tree in a single word just by encoding the corresponding rules taken in the P system for the generation of the membrane (=tree) structure, which can be done in the following way: For each rule in Π relevant for the generation of the membrane structure we take a specific symbol (only finitely many symbols are needed as we shall see later in the construction given below); now let all these symbols form the set E with cardinality z , then each element from E can be identified by a number between 1 and z . Hence, we can encode a sequence of these rules not only by a word w , but also by the number $g_{z+1}(w)$ at base $z+1$, i.e., applying the next rule can be codified by multiplying the current number by $z+1$ and then adding the value for the applied rule.

For the given tree language L_t , by definition, there exists a deterministic Turing machine $M_T(L_t)$ accepting the code $c(s)$ of a tree s if and only if $s \in L_t$. Obviously, from this Turing machine $M_T(L_t)$ one can construct a Turing machine $M'_T(L_t)$ which accepts the code w (which may be different from $c(s)$) of a tree s as described above representing the generation of the membrane structure corresponding to s if and only if $s \in L_t$. As already mentioned in the preceding section, for the Turing machine $M'_T(L_t)$ there exists a register machine $M_R(L_t)$ that for each w accepts the number $2^{g_{z+1}(w)}$ if and only if the string w is accepted by the Turing machine $M'_T(L_t)$. Here and in the register machine programs described below we take advantage of the constructions elaborated, e.g., in [2], [3], [5], and therefore omit the annoying details of these constructions; in contrast, we only show how (sequences of) ADD instructions and SUB instructions of a register machine can be simulated in the P system Π .

We now start the description of the rules needed in the generation phase:

- $[X_0]_0 \rightarrow [[X'_0]_{C_a}]_0$ for $a \in T$;
- $[X'_0]_{C_a} \rightarrow [X''_0]_{C_a} [X_A E_1^{g_{z+1}(a)}]_C$ for $a \in T$;

E_1 is the symbol used for representing the contents of register one of the register machine $M_R(L_t)$, i.e., the number of symbols E_1 in the current string corresponds with the value stored in register one in that moment of the simulation; $X_A E_1^{g_{z+1}(a)}$ now starts the acceptance check for a tree consisting only of the root labelled with a (the acceptance phase will be described in more detail below).

- $[X''_0]_{C_a} \rightarrow a$ for $a \in T$;
- $[X'_0]_{C_a} \rightarrow [X''_0]_{C_a} [X_G E_1^{g_{z+1}(a)}]_h$ for $a \in T$, $1 \leq h \leq k$;

$[X_G]_h \rightarrow [aD]_h [(X'_G, h, a, h')]_{h'}$ for $a \in T$, $1 \leq h \leq k$, $2 \leq h' \leq k+1$, $h < h'$;

At a specific level of the tree, the children of a node are generated by the rules given above in an ordered manner (according to the number assigned to the edges).

$\langle X'_G, h, a, h' \rangle$ stands for a “subroutine” which simply multiplies the current number of symbols E_1 by $z + 1$ and adds the number that encodes the generation of a new membrane labelled by h' from the membrane labelled by h (which now will get the node label a); it ends up with the control symbol X_G ; in fact, as already mentioned above, this is a simple register machine program we are not going to describe in more detail, as we will show later how (sequences of) ADD instructions and SUB instructions of a register machine can be simulated.

- $[X_G]_{k+1} \rightarrow \tilde{X}_A$;

at some point we may end the generation phase and go to start the acceptance check; for this purpose we move the single word containing all information and now carrying the control symbol \tilde{X}_A to the skin membrane by using the following rules:

- $[\tilde{X}_A]_h \rightarrow \tilde{X}_A []_h$ for $1 \leq h \leq k$;
- $[\tilde{X}_A]_0 \rightarrow [[X_A]_C]_0$;

with X_A we now start the checking phase which first from n symbols E_1 computes 2^n symbols E_1 (which again is a simple register machine program) and then simulates the actions of the Turing machine $M'_T(L_t)$ by simulating the register machine $M_R(L_t)$. This procedure will end up with the final control symbol X_{halt} (corresponding to the final label of the register machine) if and only if the generated membrane structure corresponds with a tree in L_t ; in the positive case, we halt after having applied the rule

- $[X_{halt}]_C \rightarrow \lambda$,

otherwise we end up in an infinite loop.

- $[X_G]_{k+1} \rightarrow \tilde{X}_G$;

the control symbol \tilde{X}_G allows us to move the string containing all necessary information to other membranes already generated and to produce new membranes in them.

- $[\tilde{X}_G]_h \rightarrow \langle \tilde{X}_G, h, out \rangle []_h$,
- $\tilde{X}_G []_h \rightarrow [\langle \tilde{X}_G, h, in \rangle]_h$ for $1 \leq h \leq k$;

the “subroutines” $\langle \tilde{X}_G, h, out \rangle$ and $\langle \tilde{X}_G, h, in \rangle$, respectively, multiply the current number of symbols E_1 by $z + 1$ and add the number that encode these movements (of the string carrying all information encoded in the number of symbols E_1) out of/ into a membrane labelled by h ; they both end up with the control symbol \tilde{X}_G again.

The control symbol X_G can only be regained to generate new children:

- $\tilde{X}_G []_h \rightarrow [\tilde{X}'_G]_h$ for $1 \leq h \leq k$;
 $[\tilde{X}'_G]_h \rightarrow [[\langle \tilde{X}'_G, h, h' \rangle]_{h'}]_h$ for $1 \leq h, h' \leq k$;

the “subroutine” $\langle \tilde{X}'_G, h, h' \rangle$ multiplies the current number of symbols E_1 by $z + 1$ and adds the number that encodes this movement into a membrane labelled by h and the generation of a new membrane labelled with h' inside; it ends up with the control symbol X_G , which then allows for the generation of other children by using the rules

$$[X_G]_h \rightarrow [aD]_h [\langle X'_G, h, a, h' \rangle]_{h'} \text{ for } a \in T, 1 \leq h \leq k, 2 \leq h' \leq k + 1, h < h',$$

already listed above. In contrast to the first use of these rules for the children of the root, the string with aD now also contains a lot of non-terminal symbols E_1 . In general, for removing all non-terminal symbols from a string carrying the control symbol D we use the following rules:

- $[D]_h \rightarrow [[D]_D]_h$ for $h \in \{j \mid 1 \leq j \leq k\} \cup \{C\}$,
 $[A]_D \rightarrow \lambda$ for $A \in N$;

in that way, all non-terminal symbols (including D) can be eliminated; if D is eliminated before all other non-terminal symbols have been eliminated from the string, we are forced to enter an infinite loop according to the following rules:

- $[A]_h \rightarrow [[\#]_D]_h$ for $A \in N, h \in \{j \mid 1 \leq j \leq k + 1\} \cup \{C\}$,
 $[\#]_D \rightarrow \#$,
 $[\#]_h \rightarrow [[\#]_D]_h$ for $h \in \{j \mid 1 \leq j \leq k + 1\} \cup \{C\}$.

The occurrence of the trap symbol $\#$ prohibits halting.

It now only remains to show how ADD instructions and SUB instructions can be simulated:

- The ADD instruction $X_1 : (add(r_i), X_2)$ incrementing register i of $M_R(L_t)$ can be simulated by the rules

$$[X_1]_h \rightarrow [[X'_1 E_i]_I]_h \text{ for } h \in \{j \mid 1 \leq j \leq k + 1\} \cup \{C\},$$

$$[X'_1]_I \rightarrow X_2,$$

where the number of symbols E_i corresponds with the value stored in register i .

- Decrementing register i of $M_R(L_t)$ by the SUB instruction $X_1 : (sub(r_i), X_2, X_3)$ is accomplished by the rules

$$[X_1]_h \rightarrow [[X_2]_{E_i}]_h \text{ for } h \in \{j \mid 1 \leq j \leq k + 1\} \cup \{C\},$$

$$[E_i]_{E_i} \rightarrow \lambda,$$

$$[X_2]_{E_i} \rightarrow \#.$$

The rule $[X_2]_{E_i} \rightarrow \#$ introduces the trap symbol $\#$ (which will lead to a non-halting computation) only if decrementing is not possible, i.e., there is

no symbol E_i in the current string, otherwise we proceed correctly with the control symbol X_2 .

The other case, where we assume that register i contains zero, i.e., no symbol E_i occurs in the current string, is settled by the following rules:

$$[X_1]_h \rightarrow [X_3']_{E'_i} \text{ for } h \in \{j \mid 1 \leq j \leq k+1\} \cup \{C\};$$

$$[X_3']_{E'_i} \rightarrow [X_3]_{E'_i} [E'_i]_{E'_i},$$

$$[X_3]_{E'_i} \rightarrow X_3;$$

the first copy now containing X_3 assumes the choice was correct, whereas the second copy of the string carrying the symbol E'_i in membrane E'_i now checks for the occurrence of E_i ; in the positive case the introduction of the trap symbol $\#$ leads to an infinite computation, otherwise the second copy is completely erased:

$$[B]_{E'_i} \rightarrow \lambda []_{E'_i} \text{ for } B \in N - \{E_i\},$$

$$E'_i []_{E'_i} \rightarrow [E'_i]_{E'_i},$$

$$[E'_i]_{E'_i} \rightarrow \lambda.$$

If the string after the application of the rule $[E'_i]_{E'_i} \rightarrow \lambda$ is not yet empty (which for sure is the case if at least one symbol E_i originally was present), then the trap symbol $\#$ will be introduced.

In total, we have shown how we can generate an arbitrary membrane structure and in parallel compute its code for the acceptance check carried out by simulating the register machine representing the given tree language. We get a halting computation if and only if the register machine halts, i.e., if and only if it accepts the code of the tree represented by the generated membrane structure. This observation completes the proof. \square

As can be seen from the construction of the proof given above, a halting computation fulfills all conditions for interpreting the final configuration as a tree over the node alphabet T and the edge alphabet $\{1, \dots, k\}$, which provides an even stronger result than that stated in the theorem, because the additional condition for being able to interpret the membrane structure of the final configuration as a (node-labelled) tree, i.e., that the contents of each membrane region has to be a singleton $a \in T$, need not be taken into account.

4 Summary and Future Research

We have shown that P systems can also be used for generating (representations of) any recursively enumerating tree language by taking the tree representing the membrane structure of the final configuration as the result of a halting computation. In this paper we took the model of P systems with active membranes (but without using polarizations) working on string objects for obtaining this result for tree languages. By allowing for membrane deletion we could avoid changing labels.

Future investigations will focus on variants of P systems with active membranes using other restricted types of rules, even with using polarizations or changing labels; moreover, we shall consider various models of P systems working with symbol objects and investigate their computational power with respect to generating or accepting tree languages.

Acknowledgements

This paper was partly initiated in the friendly and inspiring atmosphere of the Brainstorming Week on Membrane Computing taking place at Seville in the first week of February 2004, where the first author could take advantage of the fruitful discussions with several participants.

References

- [1] J. Dassow, Gh. Păun: *Regulated Rewriting in Formal Language Theory*. Springer-Verlag, Berlin (1989)
- [2] R. Freund, C. Martin-Vide, Gh. Păun: From regulated rewriting to computing with membranes: collapsing hierarchies. *Theoretical Computer Science* **312** (2004) 143–188
- [3] R. Freund, M. Oswald: GP systems with forbidding context. *Fundamenta Informaticae* **49**, 1-3 (2002) 81–102
- [4] R. Freund, A. Păun: P systems with active membranes and without polarizations. To appear in *Soft Computing*
- [5] R. Freund, Gh. Păun: On the number of non-terminal symbols in graph-controlled, programmed and matrix grammars. In: M. Margenstern, Yu. Rogozhin (eds.): *Proc. MCU 2001*, Chişinău, 2001, LNCS 2055, Springer-Verlag, Berlin (2001) 214–225
- [6] J. Hopcroft, J. Ullmann: *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley (1979)
- [7] S. N. Krishna, R. Rama: A variant of P systems with active membranes: solving NP-complete problems. *Romanian J. of Information Science and Technology* **2**, 4 (1999) 357–367
- [8] M.L. Minsky: *Computation: Finite and Infinite Machines*. Prentice Hall, Englewood Cliffs, New Jersey, USA (1967)
- [9] The P Systems Web Page: <http://psystems.disco.unimib.it/>
- [10] A. Păun: On P systems with active membranes. Proc. of the First International Conference on Unconventional Models of Computation (UMC2K). In: I. Antoniou, C.S. Calude, M.J. Dinneen (eds.): *Discrete Mathematics and Theoretical Computer Science*, Springer-Verlag, Brussels, Belgium (2000) 187–201

- [11] Gh. Păun: Computing with membranes. *Journal of Computer and System Sciences* **61**, 1 (2000) 108–143, and TUCS Research Report 208 (1998) (<http://www.tucs.fi>)
- [12] Gh. Păun: P systems with active membranes: attacking NP complete problems. *Journal of Automata, Languages and Combinatorics* **6**, 1 (2001) 75–90
- [13] Gh. Păun: *Membrane Computing: An Introduction*. Springer-Verlag, Berlin (2002)
- [14] Gh. Păun, Y. Suzuki, H. Tanaka, T. Yokomori: On the power of membrane division in P systems. *Proc. Conf. on Words, Languages, and Combinatorics*, Kyoto (2000)
- [15] A. Salomaa, G. Rozenberg (eds.): *Handbook of Formal Languages*. Springer-Verlag, Berlin (1997)