Computing with Membranes: Variants with an Enhanced Membrane Handling

Maurice MARGENSTERN[a], Carlos MARTÍN-VIDE[b], Gheorghe PĂUN[c]

[a]Université de Metz, LITA, UFR MIM
Ile du Saulcy, 57045 Metz Cedex, France
E-mail: margens@lita.univ-metz.fr

[b]Research Group in Mathematical Linguistics
Rovira i Virgili University
Pl. Imperial Tàrraco 1, 43005 Tarragona, Spain
E-mail: cmv@astor.urv.es

[c]Institute of Mathematics of the Romanian Academy
PO Box 1-764, 70700 Bucureşti, Romania
E-mail: gpaun@imar.ro, g_paun@hotmail.com

Abstract. Membrane computing is a recently introduced (very general) computing framework which abstracts from the way the living cells process chemical compounds in their compartmental structure. Many variants considered in the literature are computationally universal and/or able to solve NP-complete problems in polynomial (even linear) time – of course, by making use of an exponential working space created in a natural way (for instance, by membrane division).
In the present paper we propose a general class of membrane systems, where besides rules for *objects evolution* (the objects are described by strings over a finite alphabet), there are rules for *moving objects* from a compartment to another one (this is done conditionally, depending on the strings contents), and for *handling membranes*. Especially this latter feature is important (and new in many respects), because it makes possible to interpret several DNA computing experiments as membrane computations. Specifically, rules for *dividing membranes* (with the contents *replicated* or *separated* according to a given property of strings), *creating, merging,* or *dissolving* them are considered. Some of these variants generalize certain previous variants of membrane systems, for the new variants we investigate their power and computational efficiency (as expected, universality results, as well as polynomial solutions of NP-complete problems are found; the latter case is illustrated with the SAT problem).

1 Introduction; The Basic Idea

The membrane systems – they are also called P systems – were introduced (in [11]) as a possible answer to the question whether or not the frequent statements (see, e.g., [3], [10]) that the processes which take place in a living cell are "computations", that "the alive cells are computers", are just metaphors, or a formal computing device can be abstracted from the cell functioning. As we will see below, the answer turned out to be affirmative.

Three are the fundamental features of living cells which are basic to P systems: (1) the complex compartmentation by means of a **membrane structure**, where (2) **sets** (or **multisets**) of chemical compounds (we call them, in general, *objects*) evolve according to prescribed (3) **rules**.

A *membrane structure* is a hierarchical arrangement of membranes, all of them placed in a main membrane, called the *skin* membrane. This one delimits the system from its environment. The membranes should be understood as three-dimensional vesicles, but a suggestive pictorial representation is by means of planar Euler-Venn diagrams (see Figure 1). Each membrane is labeled and it precisely identifies a *region*, the space between it and all the directly inner membranes, if any exists. A membrane without any membrane inside is said to be *elementary*. Mathematically, a membrane structure can be represented by a string of labeled matching parentheses. For instance, the structure from Figure 1 is represented by $[_1[_2\,]_2[_3\,]_3[_4[_5\,]_5[_6[_8\,]_8[_9\,]_9]_6[_7\,]_7]_4]_1$.

In the regions of a membrane structure we place *objects*. In this paper, the objects are described by strings of symbols over a given finite alphabet and we do not count their multiplicities (we work with *sets*, not with *multisets*: as a possible interpretation, think about objects represented by DNA molecules, which can be present in any number of copies we need, produced by amplification).

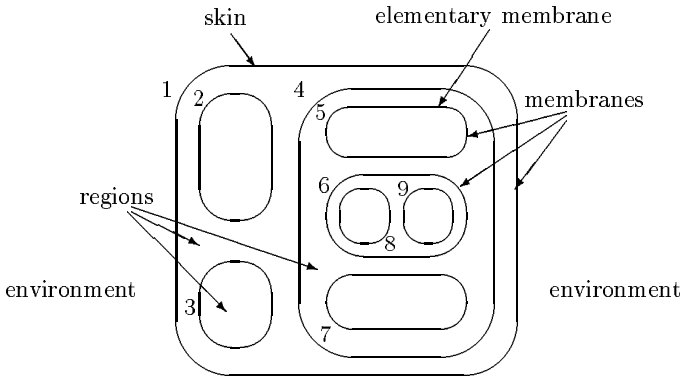

Figure 1: A membrane structure

The objects evolve by means of given *rules*, which are associated with the regions (the intuition is that the rules correspond to chemical reactions and each region has specific conditions, hence the rules from a region cannot necessarily act also elsewhere). These rules specify both object transformation (for

instance, by rewriting symbols from the string-objects, by splicing them, etc) and object transfer from a region to another one. The passing of an object through a membrane is called *communication*. The communication of strings through membranes can also be controlled by the structure of the strings; for instance, a string can exit a given membrane only if it contains a specified substring. Such variants were considered in [2].

The rules are used *in a nondeterministic maximally parallel manner*: the objects to evolve and the rules to be applied to them are chosen in a nondeterministic way, but no object which can evolve at a given step may remain unchanged. Each string-object is processed by only one rule (the strings evolve in parallel, but the rewriting of each of them means using only one rule).

Other features can be introduced, such as the possibility to control the membrane thickness/permeability, or a priority relation among rules, but here we do not consider such features.

The membrane structure together with the objects and the evolution rules present in its regions constitute a *P system*. The membrane structure and the objects define a *configuration* of a given system. By using the rules as suggested above, we can define *transitions* among configurations. A sequence of transitions is called a *computation*. We accept as successful computations only the *halting* ones, those which reach a configuration where no further rule can be applied.

With a successful computation we can associate a *result*, for instance, by considering the objects which leave the system during the computation. More precisely, we can use a P system for solving three types of tasks: as a *generative* device (start from an initial configuration and collect all strings which describe the objects which have left the system during all successful computations), as a *computing* device (start with some input placed in an initial configuration and read the output at the end of a successful computation, by considering the objects which have left the system), and as a *decidability* device (introduce a problem in an initial configuration and wait for the answer in a specified number of steps). In all cases, instead of "reading" the result outside the system we can consider a given *output membrane* and consider as the result its contents at the end of a computation.

Among the numerous variants of P systems already considered in the literature, many are computationally universal (they can compute exactly what Turing machines can compute), while many can solve NP-complete problems in polynomial (often, linear) time, by making use of an exponential workspace.

2 Handling the Membranes

So far, two main ideas were used in order to generate an exponential workspace, both of them of a biochemical inspiration: dividing membranes [12] and replicating string-objects [5]. A combination of them is based on creating membranes from symbol-objects, as in [7], and providing rules only for increasing the number of objects.

Sugestively, but not completely formalized, a rule for dividing membranes is of the form $[_i\ ]_i \to [_i\ ]_i[_i\ ]_i$, with the meaning that the membrane with the label i is divided into two copies, each one inheriting the contents of the former membrane (both the objects and the membranes placed in the initial copy of membrane i are replicated in the two new copies of membrane i).

Similarly, a rule for creating a membrane [7] is of the form $a \to [_i b]_i$, with the meaning that the object a creates a membrane with label i, which contains the object b (because we know the label of the membrane, we also know the evolution rules which are associated with it).

A powerful, related, idea was recently considered in [6], where rules of the form $[_i a_1 a_2 \dots a_n]_i \to [_i a_1]_i [_i a_2]_i \dots [_i a_n]_i$ were used: membrane i is replicated in as many copies as many symbols exist in the string $a_1 a_2 \dots a_n$.

Actually, Head's paper [6] is one of the main incentives for the present work: with the goal of formulating in terms of membrane computations several actual DNA computing experiments, Head has considered a variant of P systems with an enhanced membrane handling, including the possibility of creating, dividing, dissolving, and (this is an entirely new idea) *merging* membranes.

3 A General Class of P Systems

We consider here a general class of P systems, where rules of three types are used: for *evolving* the string-objects, for *communicating* objects through membranes, and for *handling membranes*. At each step of a computation (at each transition from a configuration to another one), we first use the object evolution rules, in the nondeterministic, maximally parallel manner usual in membrane systems, then we use the communication rules (the objects which can be moved from a region to another one are moved), and then we process the membranes. The cycle is iterated.

The evolution rules will be context-free rewriting rules of the form $a \to u$: the symbol a is replaced by the string u. Because this might be of interest (and biologically motivated), we may impose to use a rule as above only for rewriting the symbol a in the leftmost or the rightmost position of a string, which is indicated by writing $a \to_{\rm pref} u, a \to_{\rm suf} u$, respectively. When no indication is given, no restriction on the place of the rewritten a is imposed.

For the communication rules we follow the idea of [2], namely we consider *predicates* associated with each membrane and telling when a string can exit the membrane or can go into an inner membrane. These predicates can be defined according to the structure of a string, taking into account its prefixes, suffixes, subwords, or the symbols appearing in it. We do not enter here into details, because [2] provides a systematic examination of such variants and of their influence on the power of P systems. We only mention the important fact that using rewriting rules without pref/suf restrictions and various combinations of communication predicates, we get computational universality, even for systems with a reduced number of membranes – see [2].

In what concerns the rules for membrane handling, the main topic of our paper, we can consider at least the following possibilities (when specifying the system, besides the alphabet, the initial membrane structure and the objects present in its regions, we also give sets of rules associated with all possible membranes; thus, the labels of new membranes precisely identify the rules corresponding to these membranes):

- *Merge:* the fact that two membranes, with labels i and j, can be merged into a single membrane, with label k, providing that they contain the sets of objects M_i, M_j, respectively, is indicated by $[_i M_i]_i [_j M_j]_j \rightarrow [_k \,]_k$. The contents of the former membranes (objects and inner membranes included) are put together in the new membrane. Of course, in order to perform a merging operation, the two membranes must be adjacent, both of them placed in the same immediately upper membrane.
- *Divide:* a membrane i, containing the set of objects M, can be divided into two new membranes by a rule $[_i M]_i \rightarrow [_j \,]_j [_k \,]_k$; the contents of the former membrane is reproduced in the two new membranes.
- *Separate:* if we want not to replicate the contents of a membrane, but to separate its objects according to a given property π, we can use a rule $[_i \,]_i \rightarrow [_j \pi]_j [_k \neg\pi]_k$. The objects which have the property π are placed in membrane j, the objects which do not have property π are placed in membrane k.
- *Create:* we can also separate the objects of a membrane in such a way that the objects with the property π are put into a new membrane created inside it, by using rules of the form $[_i \,]_i \rightarrow [_i [_j \pi]_j \neg\pi]_i$; the objects which do not have property π remain outside the new membrane.
- *Dissolve:* a membrane can be dissolved when it contains a given set of objects, by a rule of the form $[_i M]_i \rightarrow M$, or when it is empty. In the first case, the contents of the dissolved membrane remains free in the membrane which surrounds it. The skin membrane cannot be dissolved.

The previous rules for handling membranes were formulated for any type of membranes, but they are natural (and mathematically elegant) for elementary membranes. In particular, the skin membrane should be handled with care; for instance, by a division or a separation operation, we can duplicate the system itself. From a computational point of view, this does not make much sense (we have to change the definition of the computation, of its result, etc), but from other points of view, such as the replication of a system, this can be of a central interest.

In this paper, all operations are allowed only for elementary membranes.

In the same way as each string-object is processed at each step by only one rule (the strings which cannot be rewritten are passed unchanged to the next configuration), also the membranes are processed by at most one rule each: if there are some rules which can be applied to a membrane, then one of them is nondeterministically chosen and applied, but if no rule can be applied to a given membrane, then it remains unchanged.

4 The Computing Power

From the above (informal) discussion, the reader can realize that our class of P systems is rather general and powerful. First, it contains the variants considered in [2], where only rewriting and communication rules are used, and several universality results were obtained. Second, most of the operations considered in [6] are captured in our model, so that some of the experiments discussed by Head can also be formalized in the present context. (In this way, the aqueous computations performed by Head and his collaborators can be considered as a sort of membrane computations *avant la lettre*.) Third, even the test tube "programming language" of Lipton-Adleman, [8], [1], is covered by the membrane handling rules considered here.

With respect to the last two statements, it is worth mentioning also the differences between the membrane formalism and the other mentioned ones. A membrane structure is more complex than a "test tube desk", because of the hierarchical structure of membranes, while the versatility of the model is much larger. Consider, for instance, the communication among compartments, which can be done at the level of each object. Then, the membrane formalism is specially designed to define "one macro-pot self-running" computations, in the Turing sense (an input-output function is computed), in a parallel, distributed manner. This makes possible the precise classification of variants of membrane systems and the mathematical comparison of their computing power with Turing machines and their variants.

We will examine here the power of some variants which were not considered before. Specifically, we will not use any communication rules, but only evolution rules and membrane handling rules. Because we cannot send strings out of the system, the result of a computation will be the set of terminal strings (that is, strings over a specified *terminal* alphabet) present at the end of a computation in a specified output membrane. The family of languages computed in this way by systems with at most n membranes present in any configuration, out of m possible types of membranes, using, for instance, the operations of creating and dissolving membranes, is denoted by $LP_{n/m}(create, dissolve, mode)$, where *mode* is *any* when the rewriting can be done in any place, or *pref-suf* when only the first or the last symbol of a string can be rewritten. When m above is not bounded in advance, we replace it with $*$. Of course, the mathematical challenge (also of interest from a practical point of view) is to find combinations of operations as reduced as possible, but such that the generated family is as large as possible.

We mention here three results about such families. RE denotes the family of recursively enumerable languages, those recognized by Turing machines (the reader is assumed familiar with basic elements of formal language theory, as available in many monographs; in particular, the introductory chapters of [4] and [13] can be used).

Theorem 1. $RE = LP_{3/*}(create, dissolve, pref\text{-}suf)$
$= LP_{4/*}(create, dissolve, any)$
$= LP_{3/9}(create, separate, dissolve, any)$.

The proof of the first equality starts from a Chomsky type-0 grammar G in the Kuroda normal form and constructs a P system which simulates the derivations of G by using the "rotate-and-simulate" technique, much used in the H systems area, while the other two equalities are based on the characterization of recursively enumerable languages by means of the so-called matrix grammars with appearance checking (in the binary normal form).

In order to let the reader having an idea about these proofs and also having an example of a P system at work, we give here the proof for the last equality.

Consider a matrix grammar $G = (N, T, S, M, F)$, with $N = N_1 \cup N_2 \cup \{S, \#\}$ and the matrices in M of the form $m_i : (X \to \alpha, A \to u)$, for $X \in N_1, \alpha \in N_1 \cup \{\lambda\}, A \in N_2, u \in (N_2 \cup T)^*$, or of the form $(X \to Y, A \to \#)$, for $X, Y \in N_1, A \in N_2$; also an initial matrix $(S \to XA)$ is used. Assume that we have k matrices of the first type (with rules not used in the appearance checking mode). As recently proved, it is possible to assume that only two symbols, $B^{(1)}, B^{(2)}$, are used in appearance checking rules, $B^{(j)} \to \#, j = 1, 2$.

We construct the P system Π with the total alphabet

$$V = N \cup T \cup \{c_i \mid 1 \leq i \leq k\} \cup \{d_1, d_2, e, f, Z\},$$

the terminal alphabet T, one initial membrane, with label s (from "skin") and containing the strings X, eA, for $(S \to XA)$ being the initial matrix of G, and the following sets of rules (associated with membranes labeled with $s, 0, 1, 2, 3, 4, 5, 31, 32$):

R_s : $[_s\ ]_s \to [_s[_0 \lambda$ is a subword$]_0 \lambda$ is not a subword$]_s$,
$[_s\ ]_s \to [_s[_5 \lambda$ is a subword$]_5 \lambda$ is not a subword$]_s$;

R_0 : $X \to c_i Y$,
$A \to c_i u$, for each matrix $m_i : (X \to Y, A \to u) \in M$,
$X \to c_i f$,
$A \to c_i u$, for each matrix $m_i : (X \to \lambda, A \to u)$ with a terminal u,
$B \to Z$, for all $B \in N_2$,
$[_0\ ]_0 \to [_1 c_i$ is a subword$]_1 [_2 c_i$ is not a subword$]_2$,
$X \to d_j Y$, for $(X \to Y, B^{(j)} \to \#), j = 1, 2$,
$e \to d_j e$, for $j = 1, 2$,
$[_0\ ]_0 \to [_{3j} d_j$ is a subword$]_{3j} [_4 d_j$ is not a subword$]_4, j = 1, 2$,
$[_0 f]_0 \to f$,
$e \to \lambda$;

R_1 : $c_i \to \lambda$, $1 \leq i \leq k$,
$B \to Z$, for all $B \in N_2$,
$[_1 Y]_1 \to Y$, for all $Y \in N_1$;

R_2 : $c_i \to Z$, $1 \leq i \leq k$,
$d_i \to Z$, $j = 1, 2$,

$B \to Z$, for all $B \in N_2$;

R_{3j} : $d_j \to \lambda$,

$[_{3j}\,]_{3j} \to [_{3j}[_3 Y$ is a subword$]_3 Y$ is not a subword$]_{3j}$,

$[_{3j} Y]_{3j} \to Y$,

$B^{(j)} \to Z$, for $j = 1, 2$;

R_3 : $[_3 Y]_3 \to Y$;

R_4 : $c_i \to Z$, $1 \leq i \leq k$,

$d_j \to Z$, $j = 1, 2$,

$B \to Z$, for all $B \in N_2$;

$R_5 = \emptyset$.

Membrane 5 is the output one. At any moment, we have exactly two strings in the system, one derived from the axiom X and one derived from the axiom eA, for $(S \to XA)$ the initial matrix of G. The strings derived from X are used for controlling the rewriting of the "main string", that derived from eA and which will lead to a terminal string of $L(G)$. Membranes 1 and 2 are used for simulating matrices m_i without appearance checking rules, while membranes with labels 31, 32, together with membranes 3 and 4, are used for simulating the matrices with appearance checking rules.

At each step, all strings of the skin membrane can be enclosed in a membrane with label 0. In this membrane, each string can be rewritten. Assume that X is replaced by $c_i Y$ corresponding to a matrix $m_i : (X \to Y, A \to u)$, but the string ew present at the same time in membrane 0 is not rewritten by the corresponding rule $A \to c_i u$. A separation is performed, according to the presence of c_i or d_j; all strings which arrive in membranes 2 or 4 will introduce the trap-symbol Z and remain here forever (these membranes dissolve if they contain no string). If both strings arrive in membrane 1, at the first step the symbols c_i are removed, then the membrane is dissolved, hence we return to the skin membrane with the strings Y, ew', which represent the correct simulation of the matrix m_i.

If we use the rules $X \to d_j Y, e \to d_j e$, then we can simulate the matrix $(X \to Y,\ B^{(j)} \to \#)$: the two strings should be both present in membrane $3j$, where both symbols d_j are removed; because of the string Y, we can create a membrane with label 3, where we send the string Y; at the next step, this membrane is dissolved and at the same time either no rule can be used in membrane $3j$, or the rule $B^{(j)} \to Z$ is used, providing that $B^{(j)}$ is present. At the next step, also membrane $3j$ is dissolved, because of the string Y. We return to the skin membrane with a correct simulation of the matrix.

The process can be iterated. At any moment, in membrane 0 we can also remove the symbol e, which makes impossible the simulation of appearance checking matrices. We can continue with simulating matrices without appearance checking. If the symbol f is produced, then no further simulation is possible. If we create again a membrane 0, it can only be dissolved by the rule $[_0 f]_0 \to f$. At any moment we can also create a membrane with label 5, where

all strings are sent. If such strings are not terminal, then they are not accepted in the generated language. If they are terminal, then they correspond to strings generated by G. Consequently, Π exactly generated the strings in the language $L(G)$.

Because we use membranes of nine types, but at most three are simultaneously present in the system, we have the equality $LP_{3/9}(\textit{create, separate, dissolve, any}) = RE$.

We do not know whether the results in Theorem 1 are optimal in the number of used membranes; in particular, we believe that the total number of membranes used in the first two equalities can be bounded, but we do not have a proof of this assertion. Also, further combinations of allowed rules must be considered.

5 Solving SAT in Linear Time

The computing power is of a clear mathematical interest (for instance, if we want to have *universal* – hence programmable – computing devices), but of a "practical" importance is the efficiency of computing models, in the sense of the time complexity of computations. Making use of the space provided by dividing membranes (and replicating their contents), P systems as above can solve NP-complete problems in a linear time. This is the case with SAT: consider m clauses C_i, involving n variables, and let us construct the P system with the initial membrane structure

$$[_s[_m[_{m-1}\cdots[_1[_0\,]_0]_1\cdots]_{m-1}]_m]_s,$$

with the strings a_1, d_1 placed in membrane 0; consider also the following rules:

$$\begin{aligned}
&[_0 d_i]_0 \to [_{i1}\,]_{i1}[_{i2}\,]_{i2}, 1 \le i \le n,\\
&a_i \to t_i a_{i+1},\\
&d_i \to d_{i+1}, \text{ in membrane } i1, 1 \le i \le n,\\
&a_i \to f_i a_{i+1},\\
&d_i \to d_{i+1}, \text{ in membrane } i2, 1 \le i \le n,\\
&[_{i1} d_{i+1}]_{i1}[_{i2} d_{i+1}]_{i2} \to [_0\,]_0, 1 \le i \le n-1,\\
&[_{n1} d_{n+1}]_{n1} \to d_{n+1},\\
&[_{n2} d_{n+1}]_{n2} \to d_{n+1},\\
&[_i w]_i \to [_i\,]_i w, \text{ if } w \text{ satisfies } C_i, 1 \le i \le m,\\
&[_s w]_s \to [_s\,]_s w, \text{ for all } w.
\end{aligned}$$

By division (and replication) and making use of membranes $i1, i2$, for $1 \le i \le n$, one generates all truth-assignments in membrane 0 (this takes $3n$ steps); when this operation is completed and membranes $n1, n2$ are dissolved, truth-assignments can pass through membranes providing that they satisfy the clauses associated with membranes (m more steps). In this way, in $3n + m + 1$ steps we get a string outside the system if and only if the set of clauses is satisfiable.

The communication rules can be avoided in the following way. Start from only one membrane inside the skin, membrane 0 as above, but with the objects ta_1, d_1 inside; use the previous rules for generating the truth-assignments. After dissolving membranes $n1, n2$ (hence membrane 0 is no longer created), we use the following rules:

$$[_s\ ]_s \to [_s[_{c1} w \text{ satisfies } C_1]_{c1} \text{otherwise}]_s,$$
$$[_{ci}\ ]_{ci} \to [_s[_{c(i+1)} w \text{ satisfies } C_{i+1}]_{c(i+1)} \text{otherwise}]_{ci}, 1 \leq i \leq m-1,$$
$$t_i \to \lambda, \text{ and}$$
$$f_i \to \lambda, 1 \leq i \leq n, \text{ in the membrane with the label } cm,$$
$$[_{ci} t]_{ci} \to t, 1 \leq i \leq m.$$

The truth-assignments which satisfy the clauses are encapsulated in inner membranes which grow iteratively until producing a membrane with the label cm; in this membrane all symbols $t_i, f_i, 1 \leq i \leq n$, are removed, which makes possible to dissolve all membranes $cj, 1 \leq j \leq m$, under the influence of the object t. Thus, we get t in the skin membrane (after $3n + 2m$ steps) if and only if the set of clauses is satisfiable.

Note. Work supported by NATO Project PST.CLG.976912, and, in the case of the third author, also by a grant of NATO Science Committee, Spain, 2000–2001.

References

1. L. M. Adleman, On constructing a molecular computer, in [9], 1–22.
2. P. Bottoni, A. Labella, C. Martin-Vide, Gh. Păun, Rewriting P systems with conditional communication, submitted, 2000.
3. D. Bray, Protein molecules as computational elements in living cells, *Nature*, 376 (1995), 307–312.
4. C. Calude, Gh. Păun, *Computing with Cells and Atoms*, Taylor and Francis, London, 2000.
5. J. Castellanos, A. Rodriguez-Paton, Gh. Păun, Computing with membranes: P systems with worm-objects, *IEEE 7th. Intern. Conf. on String Processing and Information Retrieval, SPIRE 2000*, La Coruna, Spain, 64–74.
6. T. Head, Aqueous simulations of membrane computations, submitted, 2001.
7. M. Ito, C. Martin-Vide, Gh. Păun, A characterization of Parikh sets of ET0L languages in terms of P systems, submitted, 2000.
8. R. J. Lipton, Speeding up computations via molecular biology, in [9], 67–74.
9. R. J. Lipton, E. B. Baum, eds., *DNA Based Computers*, Proc. of a DIMACS Workshop, Princeton, 1995, Amer. Math. Soc., 1996.
10. W. R. Loewenstein, *The Touchstone of Life. Molecular Information, Cell Communication, and the Foundations of Life*, Oxford Univ. Press, New York, 1999.
11. Gh. Păun, Computing with membranes, *Journal of Computer and System Sciences*, 61, 1 (2000), 108–143.
12. Gh. Păun, P systems with active membranes: Attacking NP-complete problems, *J. Automata, Languages and Combinatorics*, 6, 1 (2001).
13. Gh. Păun, G. Rozenberg, A. Salomaa, *DNA Computing. New Computing Paradigms*, Springer-Verlag, Berlin, 1998.

Appendices

1. Formal Language Theory Prerequisites

In this section we introduce the formal language theory notions and notations which are used in this paper; for further details we refer to [1] and [4].

For an alphabet V, by V^* we denote the set of all strings over V, including the empty one, denoted by λ.

In the proofs of the equalities from Theorem 1 we need the notions of a *matrix grammar with appearance checking*, and of *Kuroda normal form*.

A matrix grammar with appearance checking is a construct $G = (N, T, S, M, F)$, where N, T are disjoint alphabets, $S \in N$, M is a finite set of sequences of the form $(A_1 \to x_1, \ldots, A_n \to x_n)$, $n \geq 1$, of context-free rules over $N \cup T$ (with $A_i \in N, x_i \in (N \cup T)^*$, in all cases), and F is a set of occurrences of rules in M (N is the nonterminal alphabet, T is the terminal alphabet, S is the axiom, while the elements of M are called matrices).

For $w, z \in (N \cup T)^*$ we write $w \Longrightarrow z$ if there is a matrix $(A_1 \to x_1, \ldots, A_n \to x_n)$ in M and the strings $w_i \in (N \cup T)^*, 1 \leq i \leq n+1$, such that $w = w_1, z = w_{n+1}$, and, for all $1 \leq i \leq n$, either (1) $w_i = w_i' A_i w_i'', w_{i+1} = w_i' x_i w_i''$, for some $w_i', w_i'' \in (N \cup T)^*$, or (2) $w_i = w_{i+1}$, A_i does not appear in w_i, and the rule $A_i \to x_i$ appears in F. (The rules of a matrix are applied in order, possibly skipping the rules in F if they cannot be applied – therefore we say that these rules are applied in the *appearance checking* mode.)

The language generated by G is defined by $L(G) = \{w \in T^* \mid S \Longrightarrow^* w\}$. The family of languages of this form is denoted by MAT_{ac}. When $F = \emptyset$ (hence we do not use the appearance checking feature), the generated family is denoted by MAT.

It is known that $CF \subset MAT \subset MAT_{ac} = RE$, the inclusions being proper.

A matrix grammar $G = (N, T, S, M, F)$ is said to be in the *binary normal form* if $N = N_1 \cup N_2 \cup \{S, \#\}$, with these three sets mutually disjoint, and the matrices in M are in one of the following forms:

1. $(S \to XA)$, with $X \in N_1, A \in N_2$,
2. $(X \to Y, A \to x)$, with $X, Y \in N_1, A \in N_2, x \in (N_2 \cup T)^*$,
3. $(X \to Y, A \to \#)$, with $X, Y \in N_1, A \in N_2$,
4. $(X \to \lambda, A \to x)$, with $X \in N_1, A \in N_2$, and $x \in T^*$.

Moreover, there is only one matrix of type 1 and F consists exactly of all rules $A \to \#$ appearing in matrices of type 3; $\#$ is called a trap-symbol, because once introduced, it is never removed. A matrix of type 4 is used only once, in the last step of a derivation.

According to [1], for each matrix grammar there is an equivalent matrix grammar in the binary normal form.

For an arbitrary matrix grammar $G = (N, T, S, M, F)$, let us denote by $ac(G)$ the cardinality of the set $\{A \in N \mid A \to \alpha \in F\}$. From the construction in the proof of Lemma 1.3.7 in [1] one can see that if we start from a matrix grammar G and we get the grammar G' in the binary normal form, then $ac(G') = ac(G)$.

Improving the result from [3] (six nonterminals, all of them used in the appearance checking mode, suffice in order to characterize RE with matrix grammars), in [2] it was proved that four nonterminals are sufficient in order to characterize RE by matrix grammars and out of them only three are used in appearance checking rules. Of interest here is another result from [2]: if the total number of nonterminals is not restricted, then each recursively enumerable language can be generated by a matrix grammar G such that $ac(G) \leq 2$.

Consequently, to the properties of a grammar G in the binary normal form we can add the fact that $ac(G) \leq 2$. We will say that this is *the strong binary normal form* for matrix grammars.

A type-0 grammar $G = (N, T, S, P)$ is said to be in the *Kuroda normal form* if the rules from P are of one of the following forms: $A \to BC, A \to a, A \to \lambda$, $AB \to CD$, for $A, B, C, D \in N$ and $a \in T$ (that is, besides context-free rules we have only rules which replace two nonterminals by two nonterminals).

Convention. When comparing two languages, the empty string is ignored, that is, L_1 is considered identical with L_2 as soon as $L_1 - \{\lambda\} = L_2 - \{\lambda\}$.

2. Proofs of Equalities in Theorem 1

Claim 1. $RE = LP_{3/*}(create, dissolve, pref\text{-}suf)$.

Proof. Let us consider a type-0 Chomsky grammar in the Kuroda normal form, $G = (N, T, S, P)$. Consider a new symbol, E, and add all rules of the form $a \to a, a \in N \cup T \cup \{E\}$, to P; clearly, the generated language is not changed. Assume that the non-context-free rules of P are labeled in a one-to-one manner, $r_i : AB \to CD, 1 \leq i \leq k$.

We construct the P system Π with the total alphabet

$$\begin{aligned} V = {} & N \cup T \cup \{E, e, e', e'', X, Z\} \\ & \cup \{a' \mid a \in N \cup T \cup \{E\}\} \\ & \cup \{A_i, B_i' \mid r_i : AB \to CD, 1 \leq i \leq k\}, \end{aligned}$$

the terminal alphabet T, one single initial membrane, with label 0 and containing the strings ES, e, and the following sets of rules:

$R_0 : a \to_{\text{pref}} a'$,
$a' \to_{\text{pref}} Z$, and
$e \to_{\text{pref}} a'e'$, for all $a \in N \cup T \cup \{E\}$,
$e'' \to_{\text{pref}} e$,
$X \to_{\text{suf}} \lambda$,
$e \to_{\text{pref}} Z$,
$[_0\,]_0 \to [_0 [_a$ a' is a prefix$]_a$ a' is not a prefix$]_0$, for all $a \in N \cup T \cup \{E\}$,
$A \to_{\text{pref}} A_i$,
$A_i \to_{\text{pref}} Z$, for $r_i : AB \to CD \in P, 1 \leq i \leq k$,

$[_0\ ]_0 \to [_0[_i A_i$ is a prefix$]_i A_i$ is not a prefix$]_0$,
for $r_i : AB \to CD \in P, 1 \leq i \leq k$,
$B_i' \to_{pref} Z$, for $r_i : AB \to CD \in P, 1 \leq i \leq k$,
$[_0\ ]_0 \to [_0[_f E$ is a prefix$]_f E$ is not a prefix$]_0$;

$R_a : a' \to_{\text{pref}} \lambda$,
$\alpha \to_{\text{suf}} \alpha u X$, for $a \to u$ a rule from P and all $\alpha \in N \cup T \cup \{E\}$,
$e' \to_{\text{pref}} e''$,
$[_a e'']_a \to e''$,
$X \to_{\text{suf}} Z$,
for all $a \in N \cup T \cup \{E\}$;

$R_i : A_i \to_{\text{pref}} \lambda$,
$B \to_{\text{pref}} B_i'$,
$e' \to_{\text{pref}} B_i' e'$,
$[_i\ ]_i \to [_i[_{i'} B_i'$ is a prefix$]_{i'} B_i'$ is not a prefix$]_i$,
$X \to_{\text{suf}} \lambda$,
$e'' \to_{\text{pref}} e$,
$[_i e]_i \to e$,
$a \to_{\text{pref}} Z$, for all $a \in N \cup T \cup \{E\}$;

$R_{i'} : B_i' \to_{\text{pref}} \lambda$,
$e' \to_{\text{pref}} e''$,
$\alpha \to_{\text{suf}} \alpha C D X$, for all $\alpha \in N \cup T \cup \{E\}$,
$[_{i'} e'']_{i'} \to e''$,
for all $r_i : AB \to CD \in P, 1 \leq i \leq k$;

$R_f : E \to_{\text{pref}} \lambda$.

The output membrane is that with the label f.

Let us assume that in the skin membrane we have two strings, $w = w_1 E w_2$ and e; initially, $w_1 = \lambda, w_2 = S$, where S is the axiom of G.

Assume that the symbol e is rewritten with a rule $e \to_{\text{pref}} a' e'$, for some $a \in N \cup T \cup \{E\}$. Immediately, we can use the membrane creation rule which puts together in a membrane with label a the strings starting with a'. If the string $w_1 E w_2$ does not start with a' (this means that it either starts with some b' with $a \neq b$, or with A_i for some $r_i : AB \to CD \in P$), then it remains in the skin membrane and the initial symbol of it will introduce the trap-symbol Z which is never removed, hence no terminal string is obtained. The same result is obtained if we use any other membrane creation rule.

If both strings start with the same symbol a' and we create a membrane with the label a, then both strings arrive in this membrane. At the first step we remove the initial symbol a' from both strings. At the next step we will use a rule $\alpha \to_{\text{suf}} \alpha u X$, for some rule $a \to u$ from P, as well as the rule $e' \to_{\text{pref}} e''$. The

dissolving rule $[_a e'']_a \to e''$ can now be used and we return to membrane 0 the strings $w'uX, e''$, where $w = aw'$ and $a \to u \in P$. By rules $X \to_{\text{suf}} \lambda, e'' \to_{\text{pref}} e$ we get $w'u, e$, and the process can be iterated.

Note that in this way we have simulated the rule $a \to u$ from P by erasing an occurrence of a from the leftmost position of the current string and adding u in the rightmost position; in the particular case when $u = a$ we get in this way a circular permutation of the string. By such permutations, we can "rotate" the string in such a way that the rules of P can be simulated in any desired position. More precisely, we work with strings of the form $w_1 E w_2$ such that $w_2 w_1$ is a sentential form of G (the symbol E is introduced in order to indicate the beginning of strings of G; note that E can be moved from the left end of the string to the right end of the string like any other symbol).

If in the skin membrane we do not use the rule $X \to_{\text{suf}} \lambda$, but we rewrite again the leftmost symbol, then even if we lead again to a configuration with both strings in the same membrane a, the rightmost symbol of the string is X, hence we cannot use a rule $\alpha \to_{\text{suf}} \alpha u X$ for $\alpha \in N \cup T \cup \{E\}$, but the rule $X \to_{\text{suf}} Z$, which introduces the trap-symbol.

If we start by using a rule $e \to_{\text{pref}} A_i e'$, associated with a non-context-free rule $r_i : AB \to CD$ of G, then we can create a membrane with label i (all other possibilities introduce the trap-symbol). We correctly continue only in the case when also the "main string" (that generated from ES) also starts with A_i. In membrane i, we erase the symbols A_i from both strings, then e' becomes $B_i e'$ and at the same time the first symbol of the "main string" is also rewritten; if it is B, as prescribed by the rule r_i, then we replace it with B'_i, if not, then the trap-symbol is introduced by a rule $a \to_{\text{pref}} Z$. In the first case, both strings are moved into the newly created membrane with label i'. The symbols B'_i are removed, then e' becomes e'' at the same time with using a rule $\alpha \to_{\text{suf}} \alpha CDX$, for some $\alpha \in N \cup T \cup \{E\}$. Now, the dissolving rule $[_{i'} e'']_{i'} \to e''$ can be used, the strings arrive in membrane i, where X can be removed at the same time when e'' is replaced by e. In the presence of this string, membrane i is dissolved, too, hence we return to membrane 0 the strings $w'CD, e$, where $w = ABw'$, that is, the rule $r_i : AB \to CD$ was correctly simulated (again, erasing the left hand side, AB, from the beginning of the string and adding the right hand side, CD, to the end of the string).

The process can be iterated and in this way all derivations in G can be simulated. Moreover, we continue without introducing the trap-symbol Z only when this simulation is correct.

At any moment after using the rule $X \to_{\text{suf}} \lambda$ in membrane 0 we can create a membrane with label f. The strings which begin with E are send to this membrane. If there is no such a string, then the membrane f is immediately dissolved; if such a string exists, then at the next step the symbol E will be removed and the string remains forever in membrane f. If the string is terminal with respect to G, then it is also accepted in the language $L(\Pi)$, if not, then it is "lost". The computation can stop when using the rule $e \to_{\text{pref}} Z$: only by using the symbol e and its primed variants we can dissolve the membranes with

labels a, i, i'. Because the string derived from ES can be moved to membrane f only if it begins with the symbol E, we ensure that the string is in the same permutation as the corresponding string of G. Consequently, $L(G) = L(\Pi)$, and the proof is complete (note that the maximal number of membranes which are simultaneously present in a configuration is three, but the total number of types of membranes depends on the number of symbols and on the number of non-context-free rules in the starting grammar).

Claim 2. $RE = LP_{4/*}(create, dissolve, any)$.

Proof. Consider a matrix grammar $G = (N, T, S, M, F)$ in the binary normal form, that is, with $N = N_1 \cup N_2 \cup \{S, \#\}$ and the matrices in M of the form $(X \to \alpha, A \to u)$, for $X \in N_1, \alpha \in N_1 \cup \{\lambda\}, A \in N_2, u \in (N_2 \cup T)^*$, or of the form $(X \to Y, A \to \#)$, for $X, Y \in N_1, A \in N_2$; let $(S \to XA)$ be the initial matrix. Assume that we have k matrices of the first type (with rules not used in the appearance checking mode), labeled by $m_i, 1 \leq i \leq k$, and h matrices of the second type (with a rule used in the appearance checking mode), labeled by $m_{k+i}, 1 \leq i \leq h$. (Note that G is not necessarily in the *strong* binary normal form.)

We construct the P system Π with the total alphabet

$$V = N \cup T \cup \{X_i \mid X \in N_1, 1 \leq i \leq k+h\} \cup \{A' \mid A \in N_2\} \cup \{e, e', f, Z\},$$

the terminal alphabet T, one initial membrane, with the label 0 and containing the strings XA, e, for $(S \to XA)$ being the initial matrix of G, and the following sets of rules:

$$\begin{aligned}
R_0 : \; & X \to X_i, \\
& X_i \to Z, \\
& e \to X_i e', \text{ for } 1 \leq i \leq k+h, \\
& [_0 \,]_0 \to [_0 [_i X_i \text{ is a prefix}]_i X_i \text{ is not a prefix}]_0, \text{ for } 1 \leq i \leq k+h, \\
& [_0 \,]_0 \to [_0 [_f f \text{ is a prefix}]_f f \text{ is not a prefix}]_0, \\
& e \to \lambda; \\
R_i : \; & e' \to e, \\
& A' \to Z, \text{ for all } A \in N_2, \\
& [_i \,]_i \to [_i [_{i'} X_i \text{ is a prefix}]_{i'} X_i \text{ is not a prefix}]_i, \\
& [_i e]_i \to e, \\
& [_i f e]_i \to f e; \\
R_{i'} : \; & X_i \to Y, \text{ for } m_i : (X \to Y, A \to u), Y \in N_1, \\
& X_i \to f, \text{ for } m_i : (X \to \lambda, A \to u), \\
& A \to A', \\
& A' \to u, \text{ for } m_i : (X \to \alpha, A \to u),
\end{aligned}$$

$[_{i'}\ ]_{i'} \to [_{i'}[_{i''} Ye \text{ is a prefix}]_{i''} Ye \text{ is not a prefix}]_{i'}$,
$[_{i'}\ ]_{i'} \to [_{i'}[_{i''} fe \text{ is a prefix}]_{i''} fe \text{ is not a prefix}]_{i'}$,
$[_{i'} e]_{i'} \to e$,
$[_{i'} fe]_{i'} \to fe$;

$R_{i''} : Y \to \lambda$,
$[_{i''} e]_{i''} \to e$,
$[_{i''} fe]_{i''} \to fe$,
for all $1 \leq i \leq k$;

$R_{k+i} : X_{k+i} \to Y$,
$A \to Z$, for $m_{k+i} : (X \to Y, A \to \#)$,
$e' \to e$,
$[_{k+i}\ ]_{k+i} \to [_{k+i}[_{(k+i)'} Ye \text{ is a prefix}]_{(k+i)'} Ye \text{ is not a prefix}]_{k+i}$,
$[_{k+i} e]_{k+i} \to e$;

$R_{(k+i)'} : Y \to \lambda$,
$[_{(k+i)'} e]_{(k+i)'} \to e$,
for all $1 \leq i \leq h$;

$R_f : f \to \lambda$.

Membrane f is the output one. At any moment, we have exactly two strings in the system, one derived from the axiom XA and one derived from the axiom e, for $(S \to XA)$ the initial matrix of G. The strings derived from e are used for controlling the rewriting of the "main string", that derived from XA and which will lead to a terminal string of $L(G)$. Membranes with labels i, i', i'', for $1 \leq i \leq k$, are used for simulating matrices m_i without appearance checking rules, while membranes with labels $k+i, (k+i)'$, for $1 \leq i \leq h$, are used for simulating the matrices with appearance checking rules. This is done as follows.

After each use of a rewriting rule in the skin membrane we can create a membrane with the label $i, 1 \leq i \leq k+h$. The strings starting with X_i are moved to this membrane, those which do not start with X_i remain in the skin membrane; the only rules which can be applied to them here are of the form $X_j \to Z$, hence the trap-symbol is introduced. Therefore, always when a new membrane is created, we must have strings starting with the same symbol.

Assume that this is the case, for some $X_i, 1 \leq i \leq k$. In membrane i we replace e' by e, but the other string is unchanged. Immediately, we create a further membrane inside membrane i, with label i' and the strings are moved there. In membrane i' we replace X_i by Y in $X_i e$ and at the same time either the same rule is applied to the string $X_i w$, or the rule $A \to A'$ is applied, for $m_i : (X \to Y, A \to u)$ (if the matrix was a terminal one, then instead of Y we introduce f). The string e (in the last step, fe), creates a membrane with label i'', where only the string Ye (at the final step, fe) is sent. In membrane i'' we erase Y (nothing is done in the last step), and at the same time in the string remained in membrane i' we either replace X_i by Y (by f, for a terminal matrix),

or we prime one more symbol A. After returning e (or fe) to membrane i', also this membrane is dissolved. If the string returned to membrane i still contains an occurrence of A', then the trap-symbol is introduced, otherwise no rule can be applied, and also membrane i is dissolved. We simulate in this way the use of the matrix $m_i : (X \to \alpha, A \to u)$ with $\alpha \in N_1 \cup \{\lambda\}$. If we have returned the string e to the skin membrane, then the process can be iterated; if we have obtained the string fe, then we can create a membrane with label f, where we send the strings which begin with f. In membrane f we erase f and we accept the strings which are terminal with respect to G. The computation stops by using the rule $e \to \lambda$. (This rule can be used at any time, but without the help of the symbol e and of its primed variants, we cannot handle the other string, it remains forever in an inner membrane.)

Assume now that we have introduced the symbol X_{k+i}, for some $1 \leq i \leq h$, in both strings from the skin membrane. We create a membrane with the label $k+i$, where both strings are sent. In both strings, we replace X_{k+i} by Y, for $m_{k+i} : (X \to Y, A \to \#)$, and, because of Ye, we create the membrane with the label $(k+i)'$, where the string Ye is sent. The other string remains in membrane $k+i$. If any occurrence of A there exists, then the rule $A \to Z$ must be used and the trap-symbol is introduced, otherwise the string remains unchanged. In parallel, in membrane $(k+i)'$ the symbol Y from Ye is erased, then this membrane is dissolved. If at the same time we create again a membrane $(k+i)'$ inside membrane $k+i$, it is empty, so it is immediately dissolved. Because of e, also membrane $k+i$ is dissolved. In this way, we correctly simulate the use of the matrix m_{k+i}, with the second rule used in the appearance checking mode. Again, the process can be iterated.

Consequently, $L(G) = L(\Pi)$.

The maximal number of membranes simultaneously present in the system is 4 (this number is reached when all membranes $0, i, i', i''$ are present), which concludes the proof.

References

1. J. Dassow, Gh. Păun, *Regulated Rewriting in Formal Language Theory*, Springer-Verlag, Berlin, 1989.
2. R. Freund, Gh. Păun, On the number of non-terminals in graph-controlled, programmed, and matrix grammars, submitted, 2000.
3. Gh. Păun, Six nonterminals are enough for generating each r.e. language by a matrix grammar, *Intern. J. Computer Math.*, 15 (1984), 23–37.
4. G. Rozenberg, A. Salomaa, eds., *Handbook of Formal Languages*, Springer-Verlag, Heidelberg, 1997.