Journal of Universal Computer Science, vol. 8, no. 3 (2002), 369-381
submitted: 21/2/01, accepted: 1/3/02, appeared: 28/3/02 © *J.UCS*

Membrane Computing: The Power of (Rule) Creation

Fernando Arroyo
Dpto. de Lenguajes, Proyectos y Sistemas Informáticos
Escuela de Informática – U.P.M.
Carretera de Valencia Km. 7, 28031 Madrid, Spain
E-mail: farroyo@eui.upm.es

Angel Baranda
Dpto. de Inteligencia Artificial
Facultad de Informática – U.P.M.
Campus de Motegancedo, Boadilla del Monte, 28660 Madrid, Spain

Juan Castellanos
Dpto. de Inteligencia Artificial
Facultad de Informática – U.P.M.
Campus de Motegancedo, Boadilla del Monte, 28660 Madrid, Spain
E-mail: jcastellanos@fi.upm.es

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

Abstract: We consider a uniform way of treating objects and rules in P systems: we start with multisets of rules, which are consumed when they are applied, but the application of a rule may also produce rules, to be applied at subsequent steps. We find that this natural and simple feature is surprisingly powerful: systems with only one membrane can characterize the recursively enumerable languages, both in the case of rewriting and of splicing rules; the same result is obtained in the case of symbol-objects, for the recursively enumerable sets of vectors of natural numbers.

Key Words: Molecular computing, Membrane computing, Chomsky hierarchy, Rewriting, Splicing
Category: F.1.1, F.4.2, F.4.3

1 Introduction

P systems are a class of theoretical computing models [9] abstracting from the way in which the living cell works. Informally, in the regions delimited by a membrane structure (a membrane is understood as a three dimensional vesicle), certain objects evolve according to given rules; both the objects and the rules are localized, associated with the regions. The objects can also pass through membranes, sometimes the membranes can dissolve or divide. In this way, transitions between configurations of the system are obtained; a sequence of transitions is a computation, and the result of a (halting) computation consists of all objects which leave the system during the computation.

[1] Work supported by a grant of NATO Science Committee, Spain, 2000–2001, and by Facultad de Informatica, Universidad Politecnica de Madrid

Two main classes of P systems were considered: with objects described by symbols (then we work with multisets of symbols placed in regions), or with objects described by strings of symbols (then we can work with sets or with multisets of strings placed in regions). By using the rules in a nondeterministic maximally parallel manner (in each time unit, all objects which can evolve should evolve), one defines *transitions* among the configuratins of the system. A sequence of transitions is a *computation* with which a *result* can be associated in certain ways, as a language or as a set of vectors of natural numbers. In most cases, the computational completeness is obtained, that is, characterizations of recursively enumerable (Turing computable) languages or sets of vectors (relations) of natural numbers. When posssibilities to generate an exponential working space are provided, e.g., by membrane division or by string-objects replication, then NP-complete problems can be solved in polynomial (often, linear) time.

An up-to-date bibliography of the area, including papers which illustrate the above assertions, can be found at the web address `http://bioinformatics.bio.disco.unimib.it/psystems`.

We emphasize the fact that P systems are abstract (symbolic) computing devices, of automata and language theory type, only inspired by the cell structure and functioning, and are *not at all* intended to be a model of the living cell with a biochemical relevance. For motivations, relations to biochemistry or to other computing models based on multiset processing (e.g., the Gamma language – a comprehensive survey in [1]) using or not membranes (like in the Chemical Abstract Machine, [3]), etc., we refer to [9], [12], [13]. Also, we stress the obvious fact that this paper is just a technical contribution, of a theoretical computer science type, to the rather vivid area of research which is membrane computing, using its already well established terminology and notations; the topic can be of a larger relevance, for instance, when placing it in the framework of non-monotonic (linear) logics, or with respect to the trade-off between universality (programmability), efficiency, and learnability ([4]), etc[2].

Specifically, the starting point of our work is the fact that in all variants of P systems which were investigated up to now, the rules are considered as inexhaustible: the same rule can be used for processing arbitrarily many objects, at arbitrarily many steps.

We consider here a very natural variant, related to the observation that in the cell biochemistry the "evolution rules" correspond to chemical reactions which are controlled/enhanced/promoted by certain chemical compounds (enzymes, catalysts, etc), which cannot be distinguished from the other "objects" in the cell, hence they also appear in a certain amount, and are modified during the chemical reactions. Formally, this means that also the rules are present in each region of a P system in the form of a multiset, they are consumed and reproduced during using them.

This way of influencing the next-rule-to-be-applied proves to be a very powerful programming technique for the work of a P system, for all variants: with symbol-objects and with string-objects, in the latter case with rewriting and

[2] As one of the referees has suggested, the paper could have the title "On the power of a non-monotonic multiset rule based computational paradigm", which however looks too general (hence dishonest), as we strictly refer to the specific formal model introduced in [9].

with splicing. In all these cases, the computational completeness is obtained, even for systems using only one membrane.

The idea of rules ("program instructions") which evolve when they ar applied (executed) can be related to a very modern and attractive area of computer science, that of evolving programs/computers, which are modified during their work, learning from their "experience". For P systems, the idea of handling objects and rules in the same way was already considered, in a different form, in [6], and it was also formulated as a general research topic in [11].

2 Rewriting P Systems with Rule Creation

We introduce here only the class of P systems that we will investigate in this paper. As usual, a membrane structure is represented by a string of labeled parentheses, and with each membrane we associate a *region*, which is referred to by the label of the membrane. For an alphabet V we denote by V^* the free monoid generated by V under the operation of concatenation; λ is the empty string. The family of recursively enumerable languages is denoted by RE, while the Parikh mapping associated with an alphabet V is denoted by Ψ_V. The family of all Parikh images of languages from R is denoted by $PsRE$ (this is the family of all recursively enumerable sets of vectors of natural numbers).

For the few elements of formal language theory we use here we refer to [15].

A multiset M over a set X is a mapping $M : X \longrightarrow \{\mathbf{N}\}$; a multiset over a finite support X is represented either in the form $\{(a, M(a)) \mid a \in X\}$, or by a string over X (the number of occurrences of each $a \in X$ in a string $w \in X^*$ represents the number of copies of a in the multiset).

A membrane structure will be represented by a string of labeled parentheses. (For basic elements of membrane computing we refer to the initial paper [9] and to the recent survey [13].)

A *rewriting P system* (of degree $m \geq 1$) with *rule creation* is a construct

$$\Pi = (V, T, \mu, L_1, \ldots, L_m, lab, R, R_1, \ldots, R_m),$$

where:

1. V is the alphabet of the system;
2. $T \subseteq V$ is the *terminal* alphabet;
3. μ is a membrane structure with m membranes, injectively labeled by $1, 2, \ldots, m$;
4. $L_1, \ldots, L_m$ are finite languages over V, representing the strings initially present in the regions $1, 2, \ldots, m$ of the system;
5. lab is a finite set of labels for the rules of Π;
6. R is a finite set of possible evolution rules; a rule is of the form $r : A \rightarrow x(tar)/z$, where $r \in lab, z \in lab^*$ and $A \rightarrow x$ is a context-free rule[3] over V, with $A \in V, x \in V^*$, and $tar \in \{here, out, in\}$;

[3] Note that in this paper, as usually done in the whole area of membrane computing, one works with *minimalistic* models, with systems based on as reduced as possible ingredients, hence as elegant as possible from a mathematical point of view. The reactions in biochemistry transform multisets of chemical compounds into other multisets of chemical compounds, usually triggered/enhanced/promoted/catalysed in various ways, but we are not concerned with such "practical features". A simple observation which illustrates the purely theoretic character of our approach: we do

7. $R_1, \ldots, R_m$ are finite multisets of *rules* from R associated with the regions of μ.

In a system as above, transitions are defined as usual in rewriting P systems (in each region, each string which can be rewritten by a rule from that region is rewritten), with the following important differences: (1) at all steps, in all regions we have a given number of copies of each rule; initially, this is as indicated by the multisets R_i; (2) in any region we can rewrite strings only by the existing rules (if we have a number of different strings which can be rewritten, but we have a smaller number if rules, then we rewrite only the number of strings for which we have (copies of) rules; (3) when applying a copy of a rule $r : A \to x/z$ to a string in a region i, this copy is consumed, and the rules indicated by the labels from z are introduced in region i (so, they are available at the next step of the computation); only rules from the initially specified set R can be introduced during a computation; a rule from a region of the system which is not used, may stay any number of steps in a region.

Note that the strings are supposed to appear in one copy each, so we count the number of *different* strings, while for rules we count the number of *copies* of each rule, and this number can be greater than one. A variant is to consider also multisets of strings, but we do not explicitly deal with this case here (the universality result from Theorem 1 holds also for such a case, just taking the multiplicity one for each string).

As usual, the string obtained in a rewriting step by using a rule $A \to x(tar)$ is sent to the membrane indicated by tar: *here* says that the string remains in the same region, *out* says that the string has to leave the membrane, and *in* says that the string has to be moved into one of the adjacently lower membranes, nondeterministically chosen; if there is no membrane inside the region and the rule has $tar = in$, then it cannot be applied. In this paper, the indication *here* will be omitted when presenting the rules.

A sequence of transitions forms a computation and the result of a computation is the set of strings over T sent out of the system during the computation. The language generated in this way by a system Π is denoted by $L(\Pi)$.

The family of all languages $L(\Pi)$, computed as above by rewriting P systems Π of degree at most $m \geq 1$ with rule creation is denoted by $RP_m(rule)$.

Note that in this section (the same in the following one) we do not consider halting computations, as usual in the P systems area, but we accept all strings sent out of the system, irrespective whether or not the computation will eventually halts.

As announced before, even P systems with only one membrane and with the possibility of rule creation are computationally universal. Let us denote by RE the family of recursively enumerable languages and by $PsRE$ the family of Parikh images of recursively enumerable languages (this is the family of recursively enumerable sets of vectors of natural numbers).

Theorem 1. $RE = RP_1(rule)$.

Proof. We prove only the inclusion $RE \subseteq RP_1(rule)$; the opposite inclusion can be proved in a straightforward manner (or we can invoke the Turing-Church

not care about the consevation law, and we just play with – context-free – rewriting rules in the classic sense of language theory, extended when this is the case to "rewriting" multisets of symbols.

thesis for it).

We start from the characterization of recursively enumerable languages by means of *matrix grammars with apearance checking*, [5], [15]. Such a grammar 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}. It is known that $MAT_{ac} = RE$.

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

So, let $G = (N, T, S, M, F)$ be a matrix grammar with appearance checking in the binary normal form, with $N = N_1 \cup N_2 \cup \{S, \#\}$ and rules of the four forms mentioned above. Assume that we have k matrices of the forms 2 and 4, $m_i : (X \to \alpha, A \to x)$, with $X \in N_1, \alpha \in N_1 \cup \{\lambda\}, A \in N_2, x \in (N_2 \cup T)^*$, $1 \leq i \leq k$, and h matrices of type 3, $m_{k+i} : (X \to Y, B \to \#)$, with $X, Y \in N_1, B \in N_2, 1 \leq i \leq h$. (The labels $m_i, 1 \leq i \leq k+h$, are supposed to be associated in a one-to-one manner with matrices.)

We construct the P system (of degree one)

$$\Pi = (V, T, [_1\ ]_1, L_1, lab, R, R_1),$$

with the following components:

$$\begin{aligned} V &= N \cup T \cup \{X' \mid X \in N_1\} \cup \{D, \#\}, \\ L_1 &= \{XA, D\}, \text{ for } (S \to XA) \text{ being the initial matrix of } G, \\ lab &= \{r_i, r_i', r_i'' \mid 1 \leq i \leq k\} \\ &\quad \cup \{r_j, r_{j,1}, r_{j,2}, r_{j,3} \mid k+1 \leq j \leq k+h\}, \\ R_1 &= r_1 r_2 \ldots r_k r_{k+1} \ldots r_{k+h}, \end{aligned}$$

$$\begin{aligned}
R = \{&r_i : X \to X'/r'_i,\\
&r'_i : A \to x/r''_i,\\
&r''_i : X' \to Y/r_i \mid \text{for } m_i : (X \to Y, A \to x),\\
&\qquad 1 \leq i \leq k, \text{ of type 2}\}\\
\cup\ \{&r_i : X \to X'/r'_i,\\
&r'_i : A \to x/r''_i,\\
&r''_i : X' \to \lambda(out)/\lambda \mid \text{for } m_i : (X \to Y, A \to x),\\
&\qquad 1 \leq i \leq k, \text{ of type 4}\}\\
\cup\ \{&r_j : X \to X'/r_{j,1}r_{j,2},\\
&r_{j,1} : B \to \#/\lambda,\\
&r_{j,2} : D \to DB/r_{j,3},\\
&r_{j,3} : X' \to Y/r_j \mid \text{for } m_j : (X \to Y, B \to \#),\\
&\qquad k+1 \leq j \leq k+h, \text{ of type 3}\}.
\end{aligned}$$

Let us examine the work of the system Π.

We start with two strings in the system, XA and D, hence in all steps we will have only two strings available. Moreover, initially, all rules $r_1, \ldots, r_{k+h}$ are available, in only one copy each. Such a rule r_i will start the simulation of the corresponding matrix m_i from M.

Assume that we have a configuration where a string of the form $Xw, X \in N_1, w \in (N_2 \cup T)^*$, is present in the system, together with a string $D\#^n, n \geq 0$ (initially, $n = 0$), and we also have one copy of each rule $r_i, 1 \leq i \leq k+h$. No rule can rewrite the string $D\#^n$, but if any rule can be applied to the string Xw, then exactly one should be chosen. (If no rule can rewrite X, then we stop without sending out any strings, hence without having any output.)

Assume that we apply the rule $r_i : X \to X'/r'_i$ associated with a matrix $m_i : (X \to Y,\ A \to x)$ of type 2. We obtain the string $X'w$, the rule r_i is consumed, the rule $r'_i : A \to x/r''_i$ is introduced. If this rule cannot be used, then no available rule can be used for rewriting the current string (no rule $r_j, 1 \leq j \leq k+h$, can rewrite X'). In such a case, the computation gets stuck, and we get no output. If the rule $r'_i : A \to x/r''_i$ can be used, then it is consumed and the rule $r''_i : X' \to Y/r_i$ is introduced. This new rule replaces X' by Y, thus completing the simulation of the matrix m_i, and reintroduces the rule r_i. We return to a configuration as that we have started with, hence the process can be iterated.

If the rule r_i was associated with a terminal matrix $m_i : (X \to \lambda, A \to x)$, then the associated rule r''_i is of the form $r''_i : X' \to \lambda(out)/\lambda$, hence the string is sent out of the system (and the computation stops). If the string is terminal, then it is accepted in the language $L(\Pi)$, otherwise it is "lost".

Note that the string $D\#^n$ was not involved into these computation steps (and at the end of a computation this string remains inside the system).

Assume now that we have started by using a rule $r_j : X \to X'/r_{j,1}r_{j,2}$ associated with a matrix $m_j : (X \to Y, B \to \#)$, $k+1 \leq j \leq k+h$, of type 3. The string Xw becomes $X'w$, the rule r_j is consumed, the rules $r_{j,1} : B \to \#/\lambda$ and $r_{j,2} : D \to DB/r_{j,3}$ are introduced. If the string w contains at least one occurrence of the symbol B, then the rule $r_{j,1} : B \to \#/\lambda$ must be applied, because this is the only rule which can rewrite this string. The symbol # can never be removed, so the string will never lead to a terminal one, that is, we get no output. Assume that the string w does not contain the symbol B, hence the rule $r_{j,1} : B \to \#/\lambda$ remains non-used this step. The other newly introduced rule, $r_{j,2} : D \to DB/r_{j,3}$, can be applied to the current string $D\#^n$ and we get the string $DB\#^n$; the rule $r_{j,2} : D \to DB/r_{j,3}$ disappears, and the rule $r_{j,3} : X' \to Y/r_j$ is introduced. Now, both rules $r_{j,1} : B \to \#/\lambda$ and $r_{j,3} : X' \to Y/r_j$ can be used, and, because they rewrite different strings (and are the only rules available for these strings), they must be used. The first rule produces the string Yw, which represents the correct result of simulating the matrix m_j, with the rule $B \to \#$ used in the appearance checking mode, the second rule replaces the symbol B with the symbol $\#$. At the same time, the rule r_j is again introduced. This means that we return to a configuration as that we have started with, hence the process can be iterated.

Consequently, all terminal derivations with respect to the grammar G can be simulated in the system Π and, conversely, all terminal strings generated by the system Π are strings which can also be generated by G. Thus, $L(G) = L(\Pi)$, which concludes the proof.

3 Splicing P Systems with Rule Creation

The strings in a P system can also be processed by using the *splicing* operation introduced in [7] as a formal model of the DNA recombination under the influence of restriction enzymes and ligases (see a comprehensive information about splicing in [14]).

Consider an alphabet V and two symbols $\#, \$$ not in V. A *splicing rule* over V is a string $r = u_1\#u_2\$u_3\#u_4$, where $u_1, u_2, u_3, u_4 \in V^*$. For such a rule r and for $x, y, w \in V^*$ we define

$$(x, y) \vdash_r w \text{ iff } x = x_1u_1u_2x_2,\ y = y_1u_3u_4y_2,\ w = x_1u_1u_4y_2, \\ \text{for some } x_1, x_2, y_1, y_2 \in V^*.$$

(One cuts the strings x, y in between u_1, u_2 and u_3, u_4, respectively, and one concatenates the prefix of the first string with the suffix of the second string obtained in this way.)

A *splicing P system* (of degree $m \geq 1$) *with rule creation* is a construct

$$\Pi = (V, T, \mu, L_1, \ldots, L_m, lab, R, R_1, \ldots, R_m),$$

where the components $V, T, \mu, L_1, \ldots, L_m, lab$ are exactly as in the case of rewriting P systems, $R_i, 1 \leq i \leq m$, are multisets of rules associated with the regions $1, 2, \ldots, m$ of μ, and R is the set of all possible rules, of the form $r : u_1\#u_2\$[u_3\#u_4](tar)/z$ or $r : [u_1\#u_2]\$u_3\#u_4(tar)/z$, where $r \in lab, u_1\#u_2\$u_3\#u_4$ is a usual splicing rule over V, $tar \in \{here, out, in\}$, and $z \in lab^*$. (As usual, the indication *here* will not be explicitly specified.)

A transition in Π is defined by applying the splicing rules from each region of μ, in parallel, to all possible strings from the corresponding regions, and following the target indications associated with the rules. Using a rule $r : u_1\#u_2\$[u_3\#u_4](tar)/z$ means to have a string $w = w_1u_1u_2w_2$ such that the splicing $(w, u_3u_4) \vdash w_1u_1u_4$ is obtained. Symmetrically, using a rule r :

$[u_1\#u_2]\$u_3\#u_4(tar)/z$ means to have a string $w = w_1u_3u_4w_2$ such that the splicing $(u_1u_2, w) \vdash u_1u_4w_2$ is obtained. That is, one term of the splicing is provided by the rule itself, by means of the strings u_3u_4, u_1u_2, respectively. (We need a transition from one string-object to one string-object, and this is a possibility to achieve such a string-to-string operation; this idea was also used in [8].)

The transitions in a splicing P system with rule creation are defined as in a rewriting system: each string which can be spliced must be spliced, the used rules are consumed in this way, but new rules can be introduced.

The result of a computation consists of all strings over T which are sent out of the system during the computation. We denote by $L(\Pi)$ the language of all strings of this type. By $SP_m(rule)$ we denote the family of languages $L(\Pi)$ generated by splicing P systems as above, of degree at most $m \geq 1$.

As expected (see [14] for the power of the splicing operation: splicing plus a "weak" control leads to "easy" characterizatios of recursively enumerable languages), we obtain again a universality result, for systems with one membrane only.

Theorem 2. $RE = SP_1(rule)$.

Proof. We prove again only the inclusion $RE \subseteq SP_1(rule)$, and to this aim we consider a type-0 Chomsky grammar $G = (N, T, S, P)$. We add to P all rules of the form $\alpha \to \alpha$, for $\alpha \in N \cup T$, as well as the rule $B \to B$, for a new symbol B. We denote by P' the augmented set of rules and we assume that its rules are labeled in a one-to-one manner with $p_1, \ldots, p_k$.

We construct the splicing P system

$$\Pi = (V, T, [_1\ ]_1, L_1, lab, R, R_1),$$

with the following components:

$$\begin{aligned}
V &= N \cup T \cup \{B, X, Y, Y', Z\},\\
L_1 &= \{XBSY\},\\
lab &= \{r_i, r_i', r_i'' \mid 1 \leq i \leq k\} \cup \{r_0, r_0'\},\\
R_1 &= r_0 r_1 r_2 \ldots r_k,\\
R &= \{r_i : \lambda\#uY\$[Z\#Y']/r_i',\\
&\qquad r_i' : [Xv\#Z]\$X\#\lambda/r_i'',\\
&\qquad r_i'' : \lambda\#Y'\$[Z\#Y]/r_i \mid \text{for } r_i : u \to v, 1 \leq i \leq k\}\\
&\quad \cup \{r_0 : \lambda\#BY\$[Z\#\lambda]/r_0',\\
&\qquad r_0' : [\lambda\#Z]\$X\#\lambda(out)/\lambda\}.
\end{aligned}$$

The system Π simulates the terminal derivations in the grammar G by using the rotate-and-simulate technique usual in the splicing area: with each sentential form of G one associates a circular permutation in Π, with the end markers X and Y, and with the "real" beginning of the string marked with the symbol B. Specifically, strings Xw_1Bw_2Y are produced in Π, corresponding to sentential forms w_2w_1 of G. Initially, we have $w_1 = \lambda, w_2 = S$, where S is the axiom of G.

Assume that we have a string XwY and all rules $r_0, r_1, \ldots, r_k$ available (this is true at the beginning). If no rule can be applied, then the string remains unchanged and we get no output. Assume that we can use a rule $r_i : \lambda\#uY\$[Z\#Y']/r'_i$ associated with some $p_i : u \to v$ from P'. Note that we can have $u \to v$ from P, or $u = v = \alpha$, where α is a symbol from $N \cup T \cup \{B\}$. If $XwY = Xw_1uY$, then we get the string Xw_1Y' (that is, the string u was removed from the right hand end of the string w, and the marker Y was replaced by Y'), the rule $r_i : \lambda\#uY\$[Z\#Y]/r'_i$ is removed, and the rule $r'_i : [Xv\#Z]\$X\#\lambda/r''_i$ is introduced. This new rule is the only one which can be applied to the string (all other available rules need the symbol Y in the right hand end of the string), hence we have to use it. In this way, we get the string Xvw_1Y', the rule $r'_i : [Xv\$Z]\$X\#\lambda/r''_i$ is removed and the rule $r''_i : \lambda\#Y'\$[Z\#Y]/r_i$ is introduced. This rule will replace Y' by Y, it will disappear, and introduces again the rule $r_i : \lambda\#uY\$[Z\#Y']/r'_i$. In this way, we have simulated the use of the rule $u \to v$, by removing u from the right hand end of the string w and adding v in the left hand end. If we had $u = v = \alpha$ for $\alpha \in N \cup T \cup \{B\}$, then we have circularly permuted the string with one symbol. Note that the symbol B is treated as any other symbol during rotating the string, but it cannot be rewritten by rules from P.

All derivations in G can be simulated in this manner (modulo circularly permuting the obtained string).

Assume that at some moment we use the rule $r_0 : \lambda\#BY\$[Z\#\lambda]/r'_0$. This means that the string was of the form $XwBY$, hence w is in the same permutation as in the grammar G. The suffix BY is removed (from now on no rule r_i can be applied) and the rule $r'_0 : [\lambda\#Z]\$X\#\lambda(out)/\lambda$ is introduced. This is the only rule applicable to the string Xw, and using it means to also remove the left marker X. The obtained string, w, is immediately sent out of the system. If it is terminal, then it is accepted in $L(\Pi)$, otherwise it is "lost". Consequently, $L(G) = L(\Pi)$.

The maximal length of the strings u_1, u_2, u_3, u_4 from a splicing rule $u_1\#u_2\$u_3\#u_4$ is called the *radius* of the rule. The maximal radius of a rule in a P system is called the radius of the system. In the previous proof we have not tried to have a reduced radius of the constructed system, but a system of radius three can be obtained if we start from a grammar G in the Kuroda normal form, that is, with rules of the forms $A \to a, A \to \lambda, A \to BC, AB \to CD$, where A, B, C, D are nonterminal symbols and a is a terminal symbol. Moreover, it is easy to see that by introducing appropriate sequences of rules, we can obtain a system of radius two: instead of cutting uY in one step, we can do it in two steps, and we can proceed in the same way when introducing Xv in the left hand end of the string. The details are left to the reader.

4 P Systems with Symbol-Objects

The objects in a P system can also be of an "atomic" type, without a structure, that is, described by symbols from a given alphabet. In such a case, the result of a computation will be a number or a vector of numbers, counting the objects sent out of the system during a computation. This makes necessary, on the one hand, to work with multisets of objects, on the other hand, to consider only halting computations, in order to have a criterion of accepting or not a number as being computed.

In such a framework, a P system with rule creation is a construct

$$\Pi = (V, T, \mu, w_1, \ldots, w_m, lab, R, R_1, \ldots, R_m),$$

with the components $V, T, \mu, lab, R_1, \ldots, R_m, R$ as in a rewriting P system, with $w_1, \ldots, w_m$ strings over V representing the multisets of objects present in the initial configuration, and with the difference that the rules in R are of the form $r : a \rightarrow u/z$, where $r \in lab$, $a \in V, u \in (V \times \{here, out, in\})^*$, and $z \in lab^*$. That is, the target indications are associated with each symbol-object from u (as usual, we omit to write the indication *here*), hence the objects introduced by the same rule can follow different trajectories through the system membranes.

The multiplicities of terminal objects sent out of the system during a halting computation (a computation halts if a configuration is reached where no rule can be applied) is the result of the computation. (More formally, the result of a computation is $\Psi_T(w)$, the Parikh vector associated with the multiset w of objects sent out of the system.) Note that the non-terminal symbols sent out and the terminal symbols which remain in the system are ignored when considering the result of the computation, and that a non-halting computation gives no output.

We denote by $N(\Pi)$ the set of al vectors of natural numbers computed by a system Π as above and by $PsP_m(rule)$ the family of all such sets, generated by P systems with rule creation of degree at most $m \geq 1$.

Theorem 3. $PsRE = PsP_1(rule)$.

Proof. The proof of the inclusion $PsRE \subseteq PsP_1(rule)$ is similar to that of Theorem 1, with the differences entailed by the act that this time we work with a multisets of symbols-objects (and each symbol can be processed separately, simultaneously with all symbols available). We start from a matrix gramar with appearance checking in the binary normal form $G = (N, T, S, M, F)$; with the same notations as in the proof of Theorem 1, we construct the system

$$\Pi = (V, T, [_1\ ]_1, L_1, lab, R, R_1),$$

with the following components (in the writing of R, the morphism $h : (N_2 \cup T)^* \longrightarrow (N_2 \cup (T \times \{out\}))^*$ is defined by $h(A) = A$ for $A \in N$, and $h(a) = (a, out)$ for $a \in T$):

$$\begin{aligned}
V &= N \cup T \cup \{X' \mid X \in N_1\} \cup \{D, \#\}, \\
L_1 &= \{XA\}, \text{ for } (S \rightarrow XA) \text{ being the initial matrix of } G, \\
lab &= \{r_i, r_i', r_i'' \mid 1 \leq i \leq k\} \\
&\quad \cup \{r_j, r_{j,1}, r_{j,2}, r_{j,3}, r_{j,4} \mid k+1 \leq j \leq k+h\} \\
&\quad \cup \{r_v\}, \\
R_1 &= r_1 r_2 \ldots r_k r_{k+1} \ldots r_{k+h} r_v, \\
R &= \{r_i : X \rightarrow X'/r_i', \\
&\qquad r_i' : A \rightarrow h(x)/r_i'', \\
&\qquad r_i'' : X' \rightarrow Y/r_i \mid \text{for } m_i : (X \rightarrow Y, A \rightarrow x), \\
&\qquad\qquad 1 \leq i \leq k, \text{ of type 2}\} \\
&\quad \cup \{r_i : X \rightarrow X'/r_i',
\end{aligned}$$

$$r_i' : A \to h(x)/r_i'',$$
$$r_i'' : X' \to \lambda/\lambda \mid \text{for } m_i : (X \to Y, A \to x),$$
$$*1cm1 \leq i \leq k, \text{ of type 4}\}$$
$$\cup \{r_j : X \to X'D/r_{j,1}r_{j,2},$$
$$r_{j,1} : B \to \#/\lambda,$$
$$r_{j,2} : D \to B/r_{j,3},$$
$$r_{j,3} : \# \to \#(out)/r_{j,4},$$
$$r_{j,4} : X' \to Y/r_j \mid \text{for } m_j : (X \to Y, B \to \#),$$
$$k+1 \leq j \leq k+h, \text{ of type 3}\}$$
$$\cup \{r_v : \# \to \#\#/r_v r_v\}.$$

The simulation of matrices of types 2 and 4 proceeds in the same way as in the proof of Theorem 1 (but we have to have in mind that this time we work with a multiset of symbols, not with a string; moreover, each terminal symbol is immediately sent out of the system, while the nonterminals remain inside).

The case of matrices with rules used in the appearance checking mode is different. Assume that we have a multiset described by a string Xw (in the presence of all rules $r_1, \ldots, r_{k+h}r_v$) and that we use a rule $r_j : X \to X'D/r_{j,1}r_{j,2}$. It replaces X with the symbols X' and D, disappears, and introduces the new rules $r_{j,1} : B \to \#/\lambda$ and $r_{j,2} : D \to B/r_{j,3}$. Thus, if the multiset contains the symbol B, then the rule $r_{j,1} : B \to \#/\lambda$ must be used. It introduces the trap-object # which makes possible the use of the "virus" rule $r_v : \# \to \#\#/r_v r_v$. This last rule will exponentially multiply both the symbol # and itself, so the computation will continue forever, no result is obtained.

If the rule $r_{j,1} : B \to \#/\lambda$ cannot be used, then it remains in the system. At the next step, the rule $r_{j,2} : D \to B/r_{j,3}$ previously introduced will replace D with B, also introducing the rule $r_{j,3} : \# \to \#(out)/r_{j,4}$. This rule cannot be used immediately, because we have no copy of # available, but the rule $r_{j,1} : B \to \#/\lambda$ can now be applied to the single occurrence of B introduced above. We get one copy of #, hence at the next step we can use either the rule $r_{j,3} : \# \to \#(out)/r_{j,4}$ or the virus rule. In the latter case, the computation will never finish, but we can avoid this by using the first rule, which sends the object # out of the system. In this way, the virus rule cannot be applied, it remains in the system. By using the rule $r_{j,3} : \# \to \#(out)/r_{j,4}$ we have also introduced the rule $r_{j,4} : X' \to Y/r_j$, which returns the configuration to a form as that we have started with: the symbol from N_1 is again unprimed, and all rules $r_1, \ldots, r_{k+h}$ are again present in one copy. This corresponds to a correct simulation of the matrix $m_j : (X \to Y, B \to \#)$. The process can be iterated.

Note that the rule $r_{j,3} : \# \to \#(out)/r_{j,4}$ is always introduced one step before introducing the object #, hence we can always avoid the use of the virus rule, but if the rule $r_{j,1} : B \to \#/\lambda$ is applicable, then it will lead the activation of the virus rule, because it is used before introducing the rule $r_{j,3} : \# \to \#(out)/r_{j,4}$.

We obtain the equality $Ps(\Pi) = \Psi_T(L(G))$, which concludes the proof.

5 Final Remarks

We have considered P systems with the multiplicity of rules taken into consideration and with the possibility to create new rules by using existing rules. Rather surprisingly, this biochemically motivated feature proves to be very powerful: the computational universality is obtained for systems with only one membrane (which is very unusual in membrane computing). This is true both for P systems with symbol-objects and with string-objects, in the latter case both for rewriting and for splicing evolution rules.

This result shows that, in some sense, the membrane structure is no longer important here (although sending objects outside of the system is a specific feature to membrane computing which is essentially used in the proofs), but what matters is the interplay between objects and (multisets of) rules. This is somehow similar to the regulated rewriting devices, [5] (or to controlled splicing systems), [14], but we have an essential difference between our systems with rule creation and, for instance, programmed grammars, where there also is a control of the next-rule-to-be-used: we consume and create rules, their set is dynamically evolving; moreover, we work with sets of strings (or multisets of symbols), which are processed simultaneously by the existing rules (the number of strings is checked against the number of available rules), and not with a unique sentential form, as in a grammar.

One can make a step further in handling the rules in the same manner as handling the objects, and to associate target indications also with the newly created rules (this would correspond, for instance, to the fact that when producing an enzyme in a membrane, it is possible to produce it for a neighboring region, hence it has to exit and act in the surrounding region). Because we have already universality (with only one membrane) without this additional feature, we cannot improve in the previous results; maybe, such systems with enhanced possibilities to handle the rules can be interesting from other points of view. In particular, we do not know how useful are such systems from a complexity point of view (can hard problems be solved in a polynomial time in this framework, as it is the case [10] for several classes of P systems with an enhanced parallelism?).

Another research question concerns the possible usefulness of the power of rule creation, in the above discussed sense, from a practical computer science viewpoint; we have in mind the possible implementation of P systems on usual electronic computers, as attempted in [2], [16]. Taking into account the simplicity of the variant dealt with here, it is of interest to try to consider it for implementation (maybe, after also introducing features able to provide an exponential working space, for increasing the computational effficiency). We do not enter into details, as the practical relevance of our model/results is out of the scope of this paper.

Acknowledgements. Thanks are due to two anonymous referees for comments and suggestions about the form of the presentation of the paper.

References

1. J.-P. Banâtre, P. Fradet, D. Le Métayer, Gamma and the chemical reaction model: fifteen years after, in *Multiset Processing. Mathematical, Computer Science, and*

Molecular Computing Points of View (C.S. Calude, Gh. Păun, G. Rozenberg, A. Salomaa, eds.), *Lecture Notes in Computer Science*, Springer-Verlag, in press.

2. A. Baranda, F. Arroyo, J. Castellanos, R. Gonzalo, Towards an electronic implementation of membrane computing: A formal description of nondeterministic evolution in transition P systems, *Proc. 7th Intern. Meeting on DNA Based Computers* (N. Jonoska, N.C. Seeman, eds.), Tampa, Florida, USA, 2001, 273–282.
3. G. Berry, G. Boudol, The chemical abstract machine, *Theoretical Computer Sci.*, **96** (1992), 217–248.
4. M. Conrad, The price of programmability, in *The Universal Turing Machine: A Half-Century Survey* (R. Herken, ed.), Kammerer and Unverzagt, Hamburg, 1988, 285–307.
5. J. Dassow, Gh. Păun, *Regulated Rewriting in Formal Language Theory*, Springer-Verlag, Berlin, 1989.
6. R. Freund, Generalized P systems, *Fundamentals of Computation Theory, FCT'99*, Iaşi, 1999 (G. Ciobanu, Gh. Păun, eds.), *LNCS* **1684**, Springer, 1999, 281–292.
7. T. Head, Formal language theory and DNA: An analysis of the generative capacity of specific recombinant behaviors, *Bulletin of Mathematical Biology*, **49** (1987), 737–759.
8. M. Margenstern, Gh. Păun, Y. Rogozhin, On the power of the crowd: Set-conditional string processing, submitted, 2001.
9. Gh. Păun, Computing with membranes, *Journal of Computer and System Sciences*, **61**, 1 (2000), 108–143 (see also *Turku Center for Computer Science-TUCS Report* No 208, 1998, www.tucs.fi).
10. Gh. Păun, Computing with membranes; Attacking NP-complete problems, in vol. *Unconventional Models of Computation* (I. Antoniou, C.S. Calude, M.J. Dinneen, eds.), Springer-Verlag, London, 2000, 94–115.
11. Gh. Păun, Computing with membranes (P systems): Twenty six research topics, *Auckland University, CDMTCS Report* No 119, 2000 (www.cs.auckland.ac.nz/CDMTCS).
12. Gh. Păun, From cells to computers: Computing with membranes (P systems), *BioSystems*, **59**, 3 (2001), 139–158.
13. Gh. Păun, G. Rozenberg, A guide to membrane computing, *Theoretical Computer Science*, to appear.
14. Gh. Păun, G. Rozenberg, A. Salomaa, *DNA Computing. New Computing Paradigms*, Springer-Verlag, Berlin, 1998.
15. G. Rozenberg, A. Salomaa, eds., *Handbook of Formal Languages*, Springer-Verlag, Heidelberg, 1997.
16. Y. Suzuki, H. Tanaka, On a LISP implementation of a class of P systems, *Romanian J. of Information Science and Technology*, 3, 2 (2000), 173–186.