
Membrane Computing: Some Non-Standard Ideas

Gheorghe PĂUN

Institute of Mathematics of the Romanian Academy
PO Box 1-764, 70700 București, Romania, and
Research Group on Mathematical Linguistics
Rovira i Virgili University
Pl. Imperial Tàrraco 1, 43005 Tarragona, Spain
E-mail: george.paun@imar.ro, gp@astor.urv.es

Summary. We introduce four new variants of P systems, which we call non-standard because they look rather “exotic” in comparison with systems investigated so far in the membrane computing area: (1) systems where the rules are moved across membranes rather than the objects processed by these rules, (2) systems with reversed division rules (hence entailing the elimination of a membrane when a membrane with an identical contents is present nearby), (3) systems with accelerated rules (or components), where any step except the first one takes half of the time needed by the previous step, and (4) *reliable* systems, where, roughly speaking, all possible events actually happen, providing that “enough” resources exist. We only briefly investigate these types of P systems, the main goal of this note being to formulate several related open problems and research topics.

1 Introduction

This paper is addressed to readers who are already familiar with membrane computing – or who are determined to become familiar from other sources than a standard prerequisites section – which will not be present in this paper. Still, we repeat here a phrase many times used: “A friendly introduction to membrane computing can be found in [5], while comprehensive details are provided by [4], or can be found at the web address <http://psystems.disco.unimib.it>.”

Of course, calling “exotic” the classes of P systems we are considering here is a matter of taste. This is especially true for the first type, where the objects never change the regions, but, instead, the rules have associated target indications and can leave the region where they are used. Considering migratory rules can remind of the fact that in the cell the reactions are catalyzed/driven/controlled by chemicals which can move through membranes like any other chemicals – the difference is that here the “other chemicals”

are not moving at all. Actually, P systems with moving rules were already investigated, by Rudi Freund and his co-workers, in a much more general and flexible setup, where both the objects and the rules can move (and where universality results are obtained – references can be found in [4]).

We find the functioning of such systems pretty tricky (not to say difficult), and only a way to generate the Parikh sets of matrix languages (generated by grammars without appearance checking) is provided here. Maybe the reader will prove – or, hopefully, disprove – that these systems are universal.

The second type of “exotic” systems starts from the many-times-formulated suggestion to consider P systems where for each rule $u \rightarrow v$ one also uses the rule $v \rightarrow u$. This looks strange in the case of membrane dividing rules, $[_i a]_i \rightarrow [_i b]_i [_i c]_i$, where we have to use the reversed rule $[_i b]_i [_i c]_i \rightarrow [_i a]_i$, with the meaning that if the two copies of membrane i differ only in the objects b, c , then we can remove one of them, replacing the distinguished object of the remaining membrane by a . This is a very powerful operation – actually, we have no precise estimation of “very powerful”, but only the observation that in this way we can compute in an easy manner the intersection of two sets of numbers/vectors which are computable by means of P systems with active membranes.

The third idea is to imagine P systems where the rules “learn from experience”, so that the application of any given rule takes one time unit at the first application, half of this time at the second application, and so on – the $(i + 1)$ th application of the rule takes half of the time requested by the i th application. In this way, arbitrarily many applications of a rule – even infinitely many! – take at most two time units... Even the systems which never stops (that is, are able to continuously apply rules) provide a result after a known number of *external* time units. Note this important difference, between the *external time*, measured in “physical” units of equal length, and the *computational time*, dealing with the number of rules used. When the rules are to be used in parallel, delicate synchronization problems can appear, because of the different time interval each rule can request, depending on how many times each rule has already been used in the computation.

However, characterizations of Turing computable numbers are found by means of both cooperative multiset rewriting-like rules and by antiport rules of weight 2 where the rules are accelerated. Clearly, the computation of infinite sets is done in a finite time. Universality reasons show that *all* Turing computable sets of numbers can be computed in a time bounded in advance. A concrete value of this bound remains to be found, and this seems to be an interesting (mathematical) question.

Of course, the acceleration can be considered not only at the level of rules, but also at the level of separate membranes (the first transition in such a membrane takes one time unit, the second one takes half of this time, and so on), or directly at the level of the whole system. These cases are only mentioned and left as a possible research topic (for instance, as a possible

way to compute “beyond Turing” – as it happens with accelerated Turing machines, see [1], [2], [3]).

The same happens with the idea to play a similar game with the space instead of the time. What about space compression (whatever this could mean) or expansion (idem)? The universe itself is expanding, so let us imagine that in some regions of a P system the objects (and membranes) are doubled from time to time, just like that, without applying individual rules – still, some rules for controlling this global doubling should be considered. Any quantum computing links? Especially: any application in “solving” hard problems in a fast way?

Finally, the fourth idea originates in the observation that there is a striking difference between classic computer science (complexity theory included), based on (assumed) deterministic functioning of computers, and bio-computations, as carried in DNA computing and as imagined in membrane computing: in we have “enough” molecules in a test tube, then all possible reactions will actually happen. This is far from a mathematical assertion, but many DNA computing experiments, including the history-making one by Adleman, are based on such assumptions (and on the mathematical fact that a positive solution can be effectively checked in a feasible time). The belief is that reality arranges the things around the average, it does not deal with worst cases; probabilistically stated, “the frequency tends to the probability”. If something has a non-zero probability, then eventually it will happen. We just have to provide “enough” possibilities to happen (attempts, resources, etc).

Here, we formulate this optimistic principle in the following way: if we have “enough” objects in a membrane, then *all* rules are applied in each transition. The problem is to define “enough” . . . and we take again an optimistic decision: if each combination of rules has polynomially many chances with respect to the number of combinations itself, then each combination is applied. This is especially suitable (and clear) for the case of string-objects: if we have n combinations of rules which can be applied to an existing string, then each combination will be applied to at least one string, providing that we have at least $n \cdot p(n)$ copies of the string available, for some polynomial $p(x)$ specific to the system we deal with. A P system having this property is said to be *reliable*.

Clearly, a deterministic system is reliable, and the polynomial $p(x)$ is the constant, $p(n) = 1$, so of more interest is to assume this property for non-deterministic P systems. Such systems are able to solve **NP**-complete problems in polynomial time. We illustrate this possibility for **SAT**, which is solved in linear time by means of reliable systems with string-objects able to replicate strings of length one (in some sense, we start with symbol-objects processed by rules of the form $a \rightarrow bb$, we pass to strings by rules $b \rightarrow w$, and then we process the string-objects w and the strings derived from them, as usual in P systems with string-objects).

2 Moving Rules, Not Objects

We consider here only the symbol-object case, with multiset rewriting-like rules. After being applied, the rules are supposed to move through membranes; this migration is governed by a *metarules* of the form (r, tar) , existing in all regions, with the meaning that the multiset-processing rule r , if present in the region, after being used has to go to the region indicated by tar ; as usual, tar can be one of *here*, *out*, *in*.

More formally, a *P system with migrating rules* is a construct

$$\Pi = (O, C, T, R, \mu, w_1, \dots, w_m, R_1, \dots, R_m, D_1, \dots, D_m, i_o),$$

where:

1. O is the alphabet of *objects*;
2. $C \subseteq O$ is the set of *catalysts*;
3. $T \subseteq (O - C)$ is the set of *terminal* objects;
4. R is a finite set of *rules* of the form $u \rightarrow v$, where $u \in O^+, v \in O^*$; the catalytic rules are of the form $ca \rightarrow cu$ for $c \in C, a \in O - C$, and $u \in (O - C)^*$;
5. μ is a membrane structure of degree m , with the membranes labeled in a one to one manner with $1, 2, \dots, m$;
6. w_1, \dots, w_m are strings over O specifying the multisets of objects present in the m regions of μ at the beginning of the computation;
7. R_1, \dots, R_m are subsets of R , specifying the rules available at the beginning of the computation in the m regions of μ ;
8. D_1, \dots, D_m are sets of pairs – we call them *metarules* – of the form (r, tar) , with $r \in R$ and $tar \in \{here, out, in\}$ associated with the m regions of μ ;
9. $1 \leq i_o \leq m$ is the *output* membrane of the system, an elementary one in μ .

Note that the objects from regions are present in the multiset sense (their multiplicity matters), but the rules are present in the set sense – if present, a rule is present *in principle*, as a possibility to be used. For instance, if at some step of a computation we have a rule r in a region i and it remains here for the next step, if the same rule r is sent here from an inner region or from outside the region i , then we will continue to have *the rule* r present, not two or more copies of it.

However, if a rule r is present in a region i , then it is applied in the nondeterministic maximally parallel manner as usual in P systems: we nondeterministically assign the available objects to the available rules, such that the assignation is exhaustive, no further object can evolve by means of the existing rules. There is no target indication in the rules of R , hence all the objects obtained by applying rules remain in the same region. However, the applied rules move according to the pairs (r, tar) from D_i .

Specifically, we assume that we apply pairs (r, tar) , not simply rules r . For instance, if in a region i we have two copies of a and two metarules

$(r, here), (r, out)$ for some $r : a \rightarrow b$ which is present in region i , then the two copies of a can evolve both by means of $(r, here)$, or one by $(r, here)$ and one by (r, out) , or both by means of (r, out) . In the first case, the rule r remains in the same region, in the second case the rule will be available both in the same region and outside it, while in the third case the rule r will be available only in the region outside membrane i (if this is the environment, then the rule might be “lost”, because we cannot take it back from the environment). Anyway, both copies of a should evolve (the maximality of the parallelism). If r is not present, then a remains unchanged; if a is not present, then the rule is not used, hence it remains available for the next step.

Thus, a rule can be used an arbitrarily large number of times in a region, and for each different target tar from pairs (r, tar) used, a different region of the system will have the rule available in the next step.

The sets $D_i, 1 \leq i \leq m$, are not necessarily “ R -complete”, that is, not each rule $r \in R$ has a metarule (r, tar) in each region; if a rule r arrives in a region i for which no metarule (r, tar) is provided by D_i , then the rule cannot be used, it is ignored from that step on (in region i). Thus, in each set D_i we can have none, one, two, or three different pairs (r, tar) for each r , with different $tar \in \{here, out, in\}$.

The computation starts from the initial configuration of the system, that described by $w_1, \dots, w_m, R_1, \dots, R_m$, and evolves according to the metarules from D_1, \dots, D_m . With a halting computation we associate a result, in the form of the vector $\Psi_T(w)$ describing the multiplicity of elements from T present in the output membrane i_o in the halting configuration (thus, $w \in T^*$, the objects from $O - T$ are ignored). We denote by $Ps(\Pi)$ the set of vectors of natural numbers computed in this way by Π .

As usual, the rules from R can be cooperative, catalytic, or noncooperative. We denote by $PsR_M P_m(\alpha)$ the family of sets of vectors $Ps(\Pi)$ computed as above by P systems with rule migration using at most $m \geq 1$ membranes, with rules of type $\alpha \in \{coo, cat, ncoo\}$.

The systems from the proof of the next theorem illustrate the previous definition – and also shed some light on the power of P systems with migrating rules. We *conjecture* (actually, we mainly hope) that these systems are not universal. (In the theorem below, $PsMAT$ is the family of Parikh images of languages generated by matrix grammars without appearance checking.)

Theorem 1. $PsMAT \subset PsR_M P_2(cat)$.

Proof. Let us consider a matrix grammar without appearance checking $G = (N, T, S, M)$ in the binary normal form, that is, with $N = N_1 \cup N_2 \cup \{S\}$ and the matrices from M of the forms (1) $(S \rightarrow XA), X \in N_1, A \in N_2$, (2) $(X \rightarrow Y, A \rightarrow x), X, Y \in N_1, A \in N_2, x \in (N_2 \cup T)^*$, and (3) $(X \rightarrow \lambda, A \rightarrow x), X \in N_1, A \in N_2, x \in T^*$; the sets $N_1, N_2, \{S\}$ are mutually disjoint and there is only one matrix of type (1). We assume all matrices of types (2) and (3) from M labeled in a one to one manner with elements of a set H . Without

any loss of generality we may assume that for each symbol $X \in N_1$ there is at least one matrix of the form $(X \rightarrow \alpha, A \rightarrow x)$ in M (if a symbol X does not appear in such a matrix, then it can be removed, together with all matrices which introduce it by means of rules of the form $Z \rightarrow X$).

We construct the P system with migrating rules

$$\Pi = (O, \{c\}, T, R, [{}_1[{}_2]_2]_1, w_1, w_2, R_1, R_2, D_1, D_2, 1),$$

where:

$$O = N_1 \cup N_2 \cup T \cup \{A' \mid A \in N_2\} \cup \{h, h', h'', h''' \mid h \in H\} \cup \{c, \#\},$$

$$\begin{aligned} R = \{ & r_{1,h} : X \rightarrow h, \\ & r_{2,h} : h \rightarrow h', \\ & r_{3,h} : h' \rightarrow h'', \\ & r_{4,h} : h'' \rightarrow h''', \\ & r_{5,h} : ch''' \rightarrow c\#, \\ & r_{6,h} : h''' \rightarrow \alpha, \\ & r_{7,h} : h \rightarrow Ah'', \\ & r_{8,h} : cA \rightarrow cx' \mid \text{for all } h : (X \rightarrow \alpha, A \rightarrow x) \in M, \\ & X \in N_1, \alpha \in N_1 \cup \{\lambda\}, A \in N_2, x \in (N_2 \cup T)^* \} \end{aligned}$$

$$\begin{aligned} & \cup \{r_B : B' \rightarrow B, \\ & r'_B : B \rightarrow B \mid \text{for all } B \in N_2\} \\ & \cup \{r_X : X \rightarrow XX, \\ & r'_X : X \rightarrow \lambda \mid \text{for all } X \in N_1\} \\ & \cup \{r_\infty : \# \rightarrow \#\}, \end{aligned}$$

where x' is the string obtained by priming all symbols from N_2 which appear in $x \in (N_2 \cup T)^*$ and leaving unchanged the symbols from T ,

$$\begin{aligned} w_1 &= cX_1 \dots X_s, \text{ for } N_1 = \{X_1, X_2, \dots, X_s\}, s \geq 1, \\ w_2 &= cXA, \text{ for } (X \rightarrow XA) \text{ being the initial matrix of } G, \\ R_1 &= \{r_{4,h}, r_{6,h}, r_{7,h}, r_{8,h} \mid h \in H\} \\ &\cup \{r_X, r'_X \mid X \in N_1\}, \\ R_2 &= \{r_{1,h}, r_{2,h}, r_{3,h}, r_{4,h}, r_{5,h} \mid h \in H\} \\ &\cup \{r_B, r'_B \mid B \in N_2\} \\ &\cup \{r_\infty\}, \end{aligned}$$

and with the following sets of metarules:

D_1 contains the pair (r, here) for all rules $r \in R$ with the exception of the rules from the next pairs

$(r_{1,h}, \text{out}), (r_{6,h}, \text{out}), (r_{8,h}, \text{out})$, for all $h \in H$, which are in D_1 , too;

D_2 contains the pair $(r, here)$ for all rules $r \in R$ with the exception of the rules from the next pairs

$(r_{1,h}, in)$, $(r_{6,h}, in)$, $(r_{8,h}, in)$, for all $h \in H$, which are in D_2 , too.

Note that the metarules are “deterministic”, in the sense that for each $r \in R$ there is only one pair (r, tar) in each of D_1 and D_2 , and that the only migrating rules are $r_{1,h}, r_{6,h}, r_{8,h}$, for each $h \in H$. Initially, $r_{1,h}$ is in region 2 and $r_{6,h}, r_{8,h}$ are in region 1.

For the reader convenience, the initial configuration of the system, including the metarules different from $(r, here)$ from the two regions, is pictorially represented in Figure 1.

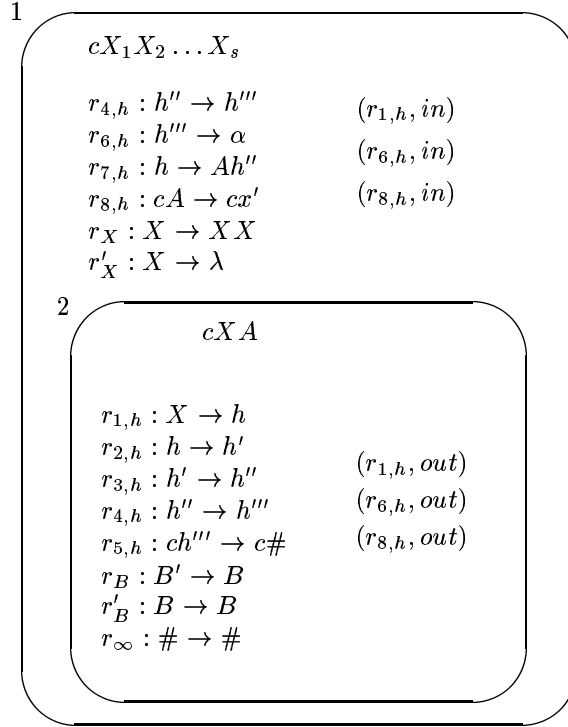


Figure 1: The P systems with migrating rules from Theorem 1.

The skin region contains the rules $X \rightarrow XX$ and $X \rightarrow \lambda$ for each $X \in N_1$; they can produce any necessary copy of symbols from N_1 and can also remove all such symbols in the end of a computation. The inner region contains a rule $B \rightarrow B$ for each $B \in N_2$, hence the computation will continue as long as any symbol from N_2 is present.

Assume that we start with a multiset cXw , $X \in N_1, w \in (N_2 \cup T)^*$ in region 2 – initially, we have here cXA , for XA introduced by the initial matrix

of G . The only possibility (besides rules $r'_B : B \rightarrow B$ which we will ignore from now on) is to use a rule of the form $r_{1,h}$ for some matrix $h : (X \rightarrow \alpha, A \rightarrow x)$. We obtain the object h in region 2, and the rule $r_{1,h}$ goes to region 1.

Assume that in region 1 we have available a copy of X , moreover, we use it for applying the rule $r_{1,h}$; thus, the rule $r_{1,h}$ returns to the inner region. At the same time, in region 2 we use the rule $r_{2,h} : h \rightarrow h'$. Now we have h in region 1 and h' in region 2; no nonterminal from N_1 is available in region 2, so no rule of type $r_{1,g}, g \in H$, can be used. Simultaneously, we use now $r_{3,h} : h' \rightarrow h''$ in region 2 and $r_{7,h} : h \rightarrow Ah''$ in region 1. Both rules remain in the respective regions.

We continue with $r_{4,h} : h'' \rightarrow h'''$ in region 2, and with both $r_{4,h} : h'' \rightarrow h'''$ and $r_{8,h} : cA \rightarrow cx'$ in region 1. The first two rules remain in the respective regions, the last one goes to region 2. In region 2 we have available both rules $r_{5,h} : ch''' \rightarrow c\#$ and $r_{8,h} : cA \rightarrow cx'$ (but not also $r_{6,h} : h''' \rightarrow \alpha$, which is still in region 1). If the rule $r_{r,5} : ch''' \rightarrow c\#$ is used, then the computation will never finish. Thus, we have to use the rule $r_{8,h} : cA \rightarrow cx'$, which simulates the second rule of the matrix h , with the nonterminals of x primed. In parallel, in region 1 we have to use the rule $r_{6,h} : h''' \rightarrow \alpha$, which has then to enter region 2. In this way, at the next step in region 2 we can avoid using the rule $r_{5,h} : ch''' \rightarrow c\#$ and use the rule $r_{6,h} : h''' \rightarrow \alpha$, which completes the simulation of the matrix h (in parallel, the rules r_B – always available – return x' to x).

If X is not present in region 1, or only one copy of X is present and we do not apply to it the rule $X \rightarrow h$, then the rule $r_{7,h}$ is not used in the next step, hence neither $r_{6,h}$ and $r_{8,h}$ after that, and then the use of the rule $r_{5,h} : ch''' \rightarrow c\#$ cannot be avoided, and $\#$ is introduced in membrane 2, making the computation never halt.

In the case when $\alpha \in N_1$, the process can be iterated: after simulating any matrix, the rules $r_{1,h}, r_{6,h}, r_{8,h}$ are back in the regions where they were placed in the beginning; note that the symbols from N_2 present in region 1 are primed, hence the rules $r_{8,h}$ cannot use them, while the possible symbol Y introduced in region 1 is just one further copy of such symbols already present here.

In the case when $\alpha = \lambda$, hence h was a terminal matrix, the computation stops if no symbol $B \in N_2$ is present in region 2 (hence the derivation in G is terminal), or continue forever otherwise. Consequently, $Ps(N) = \Psi_T(L(G))$, which shows that we have the inclusion $PsMAT \subseteq PsR_M P_2(cat)$.

This is a proper inclusion. In fact, we have the stronger assertion $PsR_M P_1(ncoo) - PsMAT \neq \emptyset$, which is proved by the fact that one-membrane P systems can compute one-letter non-semilinear sets of numbers (such sets cannot be Parikh sets of one-letter matrix languages). This is the case with the following system

$$\Pi = (\{a\}, \emptyset, \{a\}, \{h : a \rightarrow aa\}, [1]_1, a, \{h\}, \{(h, here), (h, out)\}, 1).$$

In each step we double the number of copies of a existing in the system; if we use at least once the metarule $(h, here)$, then the computation continues, otherwise the rule is “lost” in the environment and the computation halts. Clearly, $Ps(\Pi) = \{(2^n) \mid n \geq 1\}$, which is not in $PsMAT$. \square

We do not know whether the result in Theorem 1 can be improved, by computing all Parikh images of recursively enumerable languages, or – especially interesting – by using only non-cooperative rules.

The system constructed at the beginning of the previous proof has never replicated a rule, because in each region each rule had only one target associated by the local metarules. In the case when the metarules specify different targets for the same rule, we can either proceed as in the definition given above, or we can restrict the use of metarules so that only one target is used. On the other hand, we can complicate the matter by adding further features, such as the possibility to dissolve a membrane (then both the objects and the rules should remain free in the immediately upper membrane), or the possibility to also move objects through membranes. Furthermore, we can take as the result of a computation the trace of a designated rule in its passage through membranes, in the same way as the trace of a traveller-object was considered in “standard” (symport/antiport) P systems. Then, we can pass to systems with symport/antiport rules; in such a case, *out* will mean that the rule moves up and gets associated with the membrane immediately above the membrane with which the rule was associated (and applied), while *in* will mean going one step down in the membrane structure. Otherwise stated, the targets are, in fact, *here*, *up*, *down*.

Plenty of problems, but we switch to another “exotic” idea.

3 Counter-Dividing Membranes

The problem to consider reversible P systems was formulated several times – with “reversible” meaning both the possibility to reverse computations, like in dynamic systems, and local/strong reversibility, at the level of rules: considering $v \rightarrow u$ at the same time with $u \rightarrow v$. For multiset rewriting-like rules this does not seem very spectacular (although most proofs from the membrane computing literature, if not all proofs, will get ruined by imposing this restriction to the existing sets of rules, at least because the synchronization is lost). The case of membrane division rules looks different/strange. A rule $[{}_i a]_i \rightarrow [{}_i b]_i [{}_i c]_i$ produces two identical copies of the membrane i , replicating the contents of the divided membrane, the only difference between the two copies being that objects b and c replace the former object a . Moreover, generalizations can be considered, with division into more than two new membranes, with new membranes having (the same contents but) different labels. Also, we can divide both elementary and non-elementary membranes.

Let us stay at the simplest level, of rules $[{}_i a]_i \rightarrow [{}_i b]_i [{}_i c]_i$ which deal with elementary membranes only, 2 division, and the same label for the new

membranes. Reversing such a rule, hence considering a rule $[_i b]_i [_i c]_i \rightarrow [_i a]_i$, will mean to pass from two membranes with the label i and identical contents up to the two objects b and c present in them to a single membrane, with the same contents and with the objects b, c replaced by a new object, a . This operation looks rather powerful, as in only one step it compares the contents of two membranes, irrespective how large they are, and removes one of them, also changing one object.

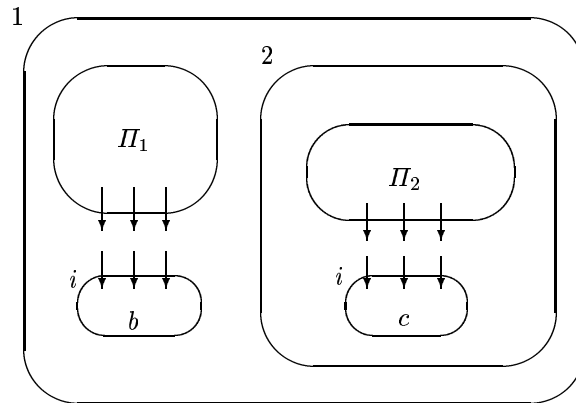


Figure 2: Computing the intersection.

How to use this presumed power, for instance, for obtaining solutions to computationally hard problems, remains to be investigated. Here we are only mentioning the fact that this counter-division operation can be used in order to build a system which computes the intersection of the sets of numbers/vectors computed by two given systems. The idea is suggested in Figure 2. Consider two P systems, Π_1 and Π_2 . Associate with each of them one membrane with label i . Embed one of the systems, say Π_2 , together with the associated membrane i , in a further membrane, with label 2, and then all these membranes into a skin membrane (1, in Figure 2). Let the systems Π_1, Π_2 work and halt; the objects defining the results are moved out of the two systems and then into the corresponding membranes with label i . At some (suitable) moment, dissolve membrane 2. Now, by a rule $[_i b]_i [_i c]_i \rightarrow [_i a]_i$ we check whether or not the two systems have produced the same number (or vector of numbers), and, in the affirmative case, we stop. (For instance, each membrane i can evolve forever by means of suitable rules applied to b and c , respectively; after removing one of the membranes, the object a newly introduced will not evolve, hence the computation can stop.) Of course, several details are to be completed, depending on the type of the systems we work with.

Now, again the question arises: is this way to compute the intersection of any usefulness? We avoid addressing this issue, and pass to the third new type of P systems.

4 Rules (and Membranes) Acceleration

A new type of P systems, but not a new idea: accelerated Turing machines were considered since many years, see [1], [2], [3] and their references. Assume that our machine is so clever that it can learn from its own functioning, and this happens at the spectacular level of always halving the time needed to perform a step; the first step takes, as usual, one time unit. Then, in $1 + \frac{1}{2} + \frac{1}{4} + \dots + \frac{1}{2^n} + \dots = 2$ time units the machine performs an infinity of steps, hence finishing the computation... Beautiful, but exotic...

Let us consider P systems using similarly clever rules, each one learning from its own previous applications and halving the time needed at the next application. Because at the same time we can have several rules used in parallel, in the same region or in separate regions, a problem appears with the cooperation of rules, with the objects some rules are producing for other rules to use. As it is natural to assume, such objects are available only after completing the use of a rule. Because the rules have now different “speeds” of application, we have to take care of the times when they can take objects produced by other rules.

How long – in terms of time units, not of computational steps (of used rules) – lasts a computation? For Turing machines, two time units are sufficient for carrying out any computation. In our case, each rule can take two time units, so a system with n rules, not necessarily used in parallel, will compute for at most $2n$ time units. Thus, if we were able to find accelerated P systems able to compute all Turing computable sets of numbers/vectors, then an upper bound can be obtained on the time needed for all these computations, by considering a universal P system (hence with a given number of rules). The only problem is to find P systems (of various types) able to compute all computable sets of numbers/vectors in the accelerated mode.

It is relatively easy to find such systems – the trick is to ensure that no two rules are used in parallel, and then no synchronization problem appears.

Consider first the case of symbol-object P systems with multiset rewriting-like rules of a cooperative type. It is easy to see that the P system constructed in the proof of Theorem 3.3.3 from [4] computes the same set of numbers with or without having accelerated rules (with the exception of rules dealing with the trap symbol #, which anyway make the computation to never stop, all other rules are not used in parallel).

The same assertion is obtained also for P systems with symport/antiport rules – specifically, for systems with antiport rules of weight 2 and no symport rule. Because of the technical interest in the proof of this assertion, we give it in some details. Let $N(\Pi)$ be the set of numbers computed by a P system Π and let $NOP_m(acc, sym_r, anti_s)$ be the family of sets $N(\Pi)$ computed by systems with at most m membranes, using accelerated symport rules of weight at most r and antiport rules of weight at most s . By NRE we denote the family of Turing computable sets of natural numbers (the length sets of recursively enumerable languages).

Theorem 2. $NRE \subseteq NOP_1(acc, sym_0, anti_2)$.

Proof. Consider a matrix grammar with appearance checking $G = (N, T, S, M, F)$ in the Z-binary normal form. Therefore, we have $N = N_1 \cup N_2 \cup \{S, Z, \#\}$, with these three sets mutually disjoint, and the matrices in M are in one of the following forms:

1. $(S \rightarrow XA)$, with $X \in N_1, A \in N_2$,
2. $(X \rightarrow Y, A \rightarrow x)$, with $X, Y \in N_1, A \in N_2, x \in (N_2 \cup T)^*, |x| \leq 2$,
3. $(X \rightarrow Y, A \rightarrow \#)$, with $X \in N_1, Y \in N_1 \cup \{Z\}, A \in N_2$,
4. $(Z \rightarrow \lambda)$.

Moreover, there is only one matrix of type 1, F consists exactly of all rules $A \rightarrow \#$ appearing in matrices of type 3, and if a sentential form generated by G contains the object Z , then it is of the form Zw , for some $w \in (T \cup \{\#\})^*$ (that is, the appearance of Z makes sure that, except for Z and, possibly, $\#$, all objects which are present in the system are terminal); $\#$ is a trap-object, and the (unique) matrix of type 4 is used only once, in the last step of a derivation.

We construct the following P system with accelerated antiport rules:

$$\begin{aligned}
\Pi &= (O, T, []_1, XA, O, R_1, 1), \\
O &= N_1 \cup N_2 \cup T \cup \{A', Y_A, Y'_A \mid Y \in N_1, A \in N_2 \cup \{Z\}\} \\
&\quad \cup \{\langle Y\alpha \rangle \mid Y \in N_1, \alpha \in N_2 \cup T \cup \{\lambda\}\} \cup \{Z, \#\}, \\
R_1 &= \{(XA, out; \langle Y\alpha_1 \rangle \alpha_2, in), \\
&\quad (\langle Y\alpha_1 \rangle, out; Y\alpha_1, in) \mid \text{for } (X \rightarrow Y, A \rightarrow \alpha_1 \alpha_2) \in M, \\
&\quad \text{with } X, Y \in N_1, A \in N_2, \alpha_1, \alpha_2 \in N_2 \cup T \cup \{\lambda\}\} \\
&\quad \cup \{(X, out; Y_A A', in), \\
&\quad (A' A, out; \#, in), \\
&\quad (Y_A, out; Y'_A, in), \\
&\quad (Y'_A A', out; Y, in) \mid \text{for } (X \rightarrow Y, A \rightarrow \#) \in M, \\
&\quad \text{with } X \in N_1, Y \in N_2 \cup \{Z\}, A \in N_2\} \\
&\quad \cup \{(\#, out; \#, in)\} \\
&\quad \cup \{(X, out; X, in) \mid X \in N_1\}.
\end{aligned}$$

The equality $N(\Pi) = \{|w| \mid w \in L(G)\}$ can easily be checked, even for the accelerated way of working. Indeed, never two rules are used in parallel, hence the acceleration makes no trouble. Now, the matrices $(X \rightarrow Y, A \rightarrow \alpha_1 \alpha_2)$ without appearance checking rules can be directly simulated by means of rules $(XA, out; \langle Y\alpha_1 \rangle \alpha_2, in), (\langle Y\alpha_1 \rangle, out; Y\alpha_1, in)$. A matrix $(X \rightarrow Y, A \rightarrow \#)$ is simulated as follows. First, we use $(X, out; Y_A A', in)$, hence X is sent to the environment and Y_A, A' enter the system (note that all objects are available in the environment in arbitrarily many copies). If any copy of A is present in the system, then the rule $(A' A, out; \#, in)$ must be used and the computation

will never halt. If A is not present, then the object A' waits in the system for the object Y'_A , which is introduced by the rule $(Y_A, out; Y'_A, in)$. By the rule $(Y'_A A', out; Y, in)$ we then send out the auxiliary objects Y'_A, A' and bring in the object Y , thus completing the simulation of the matrix. As long as any symbol $X \in N_1$ is present, the computation continues, at least by the rule $(X, out; X, in)$. Because always we have at most one object $X \in N_1$ in the system, no synchronization problem appears. When the symbol Z is introduced, this means that the derivation in G is concluded. No rule processes the object Z in the system Π . If the object $\#$ is present, then the computation will continue forever, otherwise also the computation in Π stops. Thus, we stop if and only if the obtained multiset – minus the object Z – corresponds to a terminal string generated by G , and this concludes the proof. \square

Several research topics remain to be investigated also in this case. One of them was already mentioned: find a constant U such that each set $H \in NRE$ can be computed in at most U time units by a P system of a given type (for instance, corresponding to the family $NOP_1(acc, sym_0, anti_2)$). To this aim, it is first necessary to find a universal matrix grammar with appearance checking in the Z-normal form (or to start the proof from other universal devices equivalent with Turing machines).

Then, it is natural to consider P systems where certain components are accelerated. In two time units, such a component finishes its work, possibly producing an element from a set from NRE . Can this be used by the other membranes of the system in order to speed-up computations or even to go beyond Turing, computing sets of numbers which are not Turing computable? This speculation has been made from time to time with respect to bio-computing models, membrane systems included, but no biologically inspired idea was reported able to reach such a goal. The acceleration of rules and/or of membranes can be a solution – thought not necessarily biologically inspired.

Indeed, accelerated Turing machines can solve Turing undecidable problems, for instance, the halting problem, in the following sense. Consider a Turing machine M and an input w for it. Construct an accelerated Turing machine M' , obtained by accelerating M and also providing M' with a way to output a signal s if and only if M halts when starting with w on its tape. Letting M' work, in two time units we have the answer to the question whether or not M halts on w : if M halts, then also M' halts and provides the signal s ; if M does not halt, then neither M' halts, but its computation, though infinite, lasts only two time units. So, M halts on w if and only if M' outputs s after two time units. Again, it is important to note the difference between the external time (measured in “time units”), and the internal duration of a computation, measured by the number of rules used by the machine. Also important is the fact that M' is not a proper Turing machine, because “providing a signal s ” is not an instruction in a Turing machine. (This way to get an information about a computation reminds of the way of solving deci-

sion problems by P systems, by sending into the environment a special object *yes*. The difference is that sending objects outside the system is a “standard instruction” in P systems.) Further discussions about accelerated Turing machines and their relation with “so-called Turing-Church thesis” can be found, e.g., in [2].

It is now natural to use acceleration for constructing P systems which can compute Turing non-computable sets of numbers (or functions, or languages, etc). In the string-objects case with cooperative rules this task is trivial: just take a Turing machine and simulate it by a P system; an accelerated Turing machine able of non-Turing computations will lead to a P system with the same property.

The symbol-object case does not look similarly simple. On the one hand, we do not know such a thing like an accelerated deterministic register machine (or matrix grammar with appearance checking). On the other hand, we have to make sure that the simulation of such a machine by means of a P system of a given type faithfully corresponds to the functioning of the machine: the system has to stop if and only if the machine stops. This is not the case, for instance, with the system from the proof of Theorem 2, because of the intrinsically nondeterministic behavior of Π : the trap symbol $\#$ can be introduced because of the attempt to simulate a “wrong” matrix, the derivation in G could correctly continue and eventually correctly halt, but the system fails to simulate it. It seems to be a challenging task to find an accelerated symport/antiport system which can avoid this difficulty (and hence can compute Turing non-computable sets of numbers).

Actually, in [3] one discusses ten possibilities of changing a Turing machine in such a way to obtain “hypercomputations” (computations which cannot be carried out by usual Turing machines); we do not recall these possibilities here, but we only point out them to the reader interested in “going beyond Turing”.

5 Reliable P Systems

As suggested in the Introduction, the intention behind the definition of reliable P systems is to make use of the following observation. Assume that we have some copies of an object a and two rules, $a \rightarrow b$ and $a \rightarrow c$, in the same region. Then, some copies of a will evolve by means of the first rule and some others by means of the second rule. All combinations are possible, from “all copies of a go to b ” to “all copies of a go to c ”. In biology and chemistry the range of possibilities is not so large: if we have “enough” copies of a , then “for sure” part of them will become b and “for sure” the other part will become c . What “enough” can mean depends on the circumstances. At our symbolic level, if we have two copies of a we cannot expect that “for sure” one will become b and one c , but starting from, say, some dozens of copies can ensure that both rules are applied.

We formulate this observation for string rewriting. Assume that a string w can be rewritten by n rules (each of them can be applied to w). The system we work with is *reliable* if all the n rules are used as soon as we have at least $n \cdot n^k$ copies of w , for a given constant k depending on the system. If we work with parallel rewriting and n possible derivations $w \Rightarrow w_i$, $1 \leq i \leq n$, are possible, then each one happens as soon as we have at least $n \cdot n^k$ copies of w .

We consider here polynomially many opportunities for each of the n alternatives, but other functions than polynomials can be considered; constant functions would correspond to a “very optimistic” approach, while exponential functions would indicate a much less optimistic approach.

Instead of elaborating more at the general level (although many topics arise here: give a fuzzy sets, rough sets, or probabilistic definition of reliability; provide some sufficient conditions for it, a sort of axioms entailing reliability; investigate its usefulness at the theoretical level and its adequacy/limits in practical applications/experiments), we pass directly to show a way to use the idea of reliability in solving SAT in linear time.

Consider a propositional formula C in the conjunctive normal form, consisting of m clauses C_j , $1 \leq j \leq m$, involving n variables x_i , $1 \leq i \leq n$. We construct the following P system (with string-objects, and both replicated and parallel rewriting – but the rules which replicate strings always applicable to strings of length one only) of degree m :

$$\begin{aligned}
 \Pi &= (V, \mu, b_0, \lambda, \dots, \lambda, R_1, \dots, R_m), \\
 V &= \{b_i \mid 0 \leq i \leq r\} \cup \{a_i, t_i, f_i \mid 1 \leq i \leq n\}, \\
 \mu &= [{}_1[{}_2 \cdots [{}_{m-1}[{}_m]_m]_{m-1} \cdots]_2]_1, \\
 R_m &= \{b_i \rightarrow b_{i+1} \parallel b_{i+1} \mid 0 \leq i \leq r-1\} \\
 &\cup \{b_r \rightarrow a_1 a_2 \dots a_n\} \\
 &\cup \{a_i \rightarrow t_i, a_i \rightarrow f_i \mid 1 \leq i \leq n\} \\
 &\cup \{(t_i \rightarrow t_i, out) \mid \text{if } x_i \text{ appears in } C_m, 1 \leq i \leq n\} \\
 &\cup \{(f_i \rightarrow f_i, out) \mid \text{if } \sim x_i \text{ appears in } C_m, 1 \leq i \leq n\}, \\
 R_j &= \{(t_i \rightarrow t_i, out) \mid \text{if } x_i \text{ appears in } C_j, 1 \leq i \leq n\} \\
 &\cup \{(f_i \rightarrow f_i, out) \mid \text{if } \sim x_i \text{ appears in } C_j, 1 \leq i \leq n\}, \\
 &\text{for all } j = 1, 2, \dots, m-1.
 \end{aligned}$$

The parameter r from this construction depends on the polynomial which ensures the reliability of the system – see immediately bellow what this means.

The work of the system starts in the central membrane, the one with the label m . In the first r steps, by means of replicating rules of the form $b_i \rightarrow b_{i+1} \parallel b_{i+1}$, we generate 2^r strings b_r (of length one; this phase can be considered as using symbol objects, and then the rules are usual multiset rewriting rules, $b_i \rightarrow b_{i+1}^2$). In the next step, each b_r is replaced by the string $a_1 a_2 \dots, a_n$. No parallel rewriting was used up to now, but in the next step each a_i is replaced either by t_i or by f_i . We have 2^n possibilities of combining

the rules $a_i \rightarrow t_i$, $a_i \rightarrow f_i$, $1 \leq i \leq n$. If 2^r is “large enough” with respect to 2^n , then we may assume that all the 2^n combinations really happen. This is the crucial reliability-based step. For each of the 2^n possibilities we need $(2^n)^k$ opportunities to happen. This means that we need to have a large enough r such that $2^r \geq 2^n \cdot (2^n)^k$, for a constant k (depending on the form of rules, hence on SAT). This means $r \geq n(k+1)$ – hence this is the number of steps we need to perform before introducing the strings $a_1 a_2 \dots a_n$ in order to ensure that we have “enough” copies of these strings.

After having all truth-assignments generated in membrane m , we check whether or not there is at least one truth-assignment which satisfies clause C_m ; all truth-assignments for which C_m is true exit membrane m . In membrane $m-1$ we check the satisfiability of clause C_{m-1} , and again we pass one level up all truth-assignments which satisfy C_{m-1} . We continue in this way until reaching the skin region, where we check the satisfiability of C_1 . This means that after the $r+3$ steps in membrane m we perform further $m-1$ steps; in total, this means $r+m+2$ steps.

The formula C is satisfiable if and only if at least one string is sent out of the system in step $r+m+2$. Because r is linearly bounded with respect to n , the problem was solved in a linear time (with respect to both n and m).

Note that the system is of a polynomial size with respect to n and m , hence it can be constructed in polynomial time by a Turing machine starting from C . If we start from a formula in the 3-normal form (at most three variables in each clause), then the system will be of a linear size in terms of n and m .

Reliability seems to be both a well motivated property, closely related to the biochemical reality, and a very useful one from a theoretical point of view; further investigations in this area are worth pursuing.

References

1. B.J. Copeland, Even Turing machines can compute uncomputable functions, in *Unconventional Models of Computation* (C.S. Calude, J. Casti, M.J. Dinneen, eds.), Springer-Verlag, Singapore, 1998, 150–164.
2. B.J. Copeland, R. Sylvan, Beyond the universal Turing machine, *Australasian Journal of Philosophy*, 77 (1999), 46–66.
3. T. Ord, *Hypercomputation: compute more than the Turing machine*, Honorary Thesis, Department of Computer Science, Univ. of Melbourne, Australia, 2002 (available at <http://arxiv.org/ftp/math/papers/0209/0209332.pdf>).
4. Gh. Păun, *Membrane Computing. An Introduction*, Springer-Verlag, Berlin, 2002.
5. Gh. Păun, G. Rozenberg, A guide to membrane computing, *Theoretical Computer Science*, 287 (2002), 73–100.