On Deterministic P Systems

Rudolf Freund
Faculty of Computer Science
Vienna University of Technology
Favoritenstr. 9–11, A–1040 Vienna, Austria
rudi@emcc.at

Gheorghe Păun
Institute of Mathematics of the Romanian Academy
PO Box 1-764, 014700 Bucureşti, Romania
george.paun@imar.ro

Abstract. We consider P systems working in the accepting mode: an input is provided (by the "user" of the system, placed in the environment) in the form of a number and the system accepts this number or not, provided that a halting computation exists starting from the initial configuration. Two ways to provide the input are considered: the input is introduced in the system in the initial configuration, in the form of the multiplicity of a specified object, or it is provided by means of a *signal*, a special object which is made available in the environment at a certain moment of time after the system has started to work; in this latter case, the number to recognize is identified by the number of steps elapsed since the start of the computation. We examine the possibility to recognize all Turing computable sets of numbers by means of deterministic P systems – and positive results are obtained in both cases, for symport/antiport P systems. The results are rather surprising in what concerns the complexity of these systems: they use a number of membranes and symport/antiport rules of (almost) the same size as in the non-deterministic case, which shows that in this framework the determinism does not restrict the power of our systems.

1 Introduction

The two main classes of problems investigated for the many types of membrane systems (P systems) available in the literature concern the power and the efficiency of these systems. We refer to [12] for basic notions and results, and to the web page http://psystems.disco.unimib.it for a comprehensive source of information. Because P systems are non-deterministic (by definition and, in some sense, by their biological roots)

computing devices, no special attention was paid to the distinction between deterministic and non-deterministic behaviour – except for the case of investigations related to complexity issues, where deterministic (or at least *confluent*) systems were considered, but looking for specific systems able to solve specific problems, and not for systems with a Turing complete power. Moreover, because of the way of defining successful computations as halting computations, the determinism does not make much sense in the case of systems used in the generative mode: either the system does not halt and then it computes nothing, or it halts and then it computes only a singleton.

On the other hand, determinism not only makes sense, but it is natural in the case of accepting P systems, where an input is introduced in the system, the computation starts, and the input is accepted/recognized if (and only if) the computation eventually halts. Whether or not determinism is a restriction is a standard problem for classic automata theory (not always solved – see the case of linear bounded automata). On the other hand, recently, automata-like P systems became a topic much investigated in membrane computing. We mention here only a few papers dealing with accepting P systems, sometimes even called P automata: [2, 3, 5, 9, 10]. However, somewhat surprisingly (but motivated by the "standard" non-deterministic approach in membrane computing), most if not all universality proofs present in these papers deal with non-deterministic systems. This is rather useful in proofs, because "wrong" choices of rules to apply are turned to computations which never halt (typically, a trap object is introduced, which evolves forever, thus preventing the halting of the computation).

Is non-determinism necessary (for obtaining universality)? The question is both mathematically natural and of interest when dealing with decidability questions (when solving decidability questions by means of P systems), where, for instance, we have to make sure that the system works forever because the problem has no solution, not because of a "wrong choice" in a given step of the computation. In particular, this problem has appeared in the framework of looking for ways to compute beyond Turing barrier, by making use of certain acceleration tools (see [1]); when trying to solve, say, the halting problem for Turing machines by devices of a specified type, those devices should behave in a deterministic manner. With this motivation and in this framework, two proofs were given in [1] for deterministic characterizations of Turing computability, by P systems with co-operating rules, also using membrane creation and membrane dissolving features, as well as for symport/antiport systems. While in the first case the "price of determinism" is rather high (co-operative rules alone are known to be universal – but the respective proof is non-deterministic), in the latter case the system used is like in the non-deterministic case: one membrane and only antiport rules of weight at most 2.

What is not investigated in [1] is the case where only symport rules are used, also known to be universal in the non-deterministic case. We settle this case here, again without any increase in the complexity of the system used: one membrane and symport rules of weight 3 suffice, the same as in the generative and the accepting non-deterministic cases, see [7, 8].

In the cases discussed above, a given P system identifies a set of numbers, all those for which the system halts when starting with an input provided in the initial configuration

in the form of the multiplicity of a specified object: a^n is introduced by the "user" of the system in a specified region and then the computation proceeds. In some sense, the non-determinism is placed outside of the system, as the input can be any a^n for n being a natural number.

Here we consider yet another way of providing the input (the number to be recognized), by means of *signals*. Specifically, a special object s is distinguished; the computation starts from the initial configuration of the system, and at some step n the signal s is supposed to appear in the environment; this is an indication that the number $n-1$ is to be recognized, hence if the system will halt, then $n-1$ is accepted, otherwise $n-1$ is rejected (we consider $n-1$ as the number to be recognized in order to handle also the case of the number zero). Of course, the system should sense immediately that s has appeared in the environment, not to deal with a wrong input.

The results we obtain for this way of introducing the number to recognize are similar to those mentioned above for the case when the input is provided in the initial configuration: deterministic systems with antiport rules of weight 2, or with symport rules of weight 3 are universal – with only one membrane in the former case and two membranes in the latter case (thus, in the symport case the result is not identical to the one for the initial input).

Some extensions of these results (for instance, to the case of two signals, with the number to be recognized being the number of steps in between the two consecutive signals) are considered, and some open problems are also formulated.

2 Definitions

The reader is assumed to be familiar with the basic elements of membrane computing, so that we recall here only a few notions, those central for what follows.

We investigate only *(cell-like) P systems with symport/antiport rules*, that is, systems of the form $\Pi = (O, \mu, w_1, \ldots, w_m, E, R_1, \ldots, R_m)$, where O is the alphabet of objects, μ is the membrane structure (of degree $m \geq 1$, with the membranes labelled in a one-to-one manner with $1, 2, \ldots, m$; as usual, we represent the membrane structures by strings of matching labelled parentheses), $w_1, \ldots, w_m$ are strings over O representing the multisets of objects present in the m compartments of μ in the initial configuration of the system, $E \subseteq O$ is the set of objects supposed to appear in the environment in arbitrarily many copies, and $R_1, \ldots, R_m$ are the sets of rules associated with the m membranes of μ; these rules can be of two types (by O^+ we denote the set of all non-empty strings over O):

- *Symport rules*, of the forms (x, in) or (x, out), where $x \in O^+$. When using such a rule, the objects specified by x should enter or exit, respectively, the membrane with which the rule is associated. In this way, objects are sent to or imported from the surrounding region – which is the environment in the case of the skin membrane. (The length of x in a symport rule is called the *weight* of the rule.)

- *Antiport rules*, of the form $(x, out; y, in)$, where $x, y \in O^+$. When using such a

rule for a membrane i, the objects specified by x should exit the membrane and those specified by y should enter from the region surrounding membrane i; this is the environment in the case of the skin membrane. (The maximal length of x, y is called the *weight* of the rule.)

The rules are used in the maximally parallel manner, thus defining transitions from a configuration of the system to the next configuration. A sequence of transitions constitutes a computation, and a computation halts when no rules can be used in the last configuration.

If in each step of a computation there is at most one choice of rules to be applied (that is, either the computation halts, or the next configuration is uniquely determined), then we say that the system is *deterministic*. Otherwise, it is called non-deterministic.

In what follows, we use P systems in the accepting mode: a multiset a^n is introduced in the skin region of the system, for a distinguished object a, and then the computation starts; if the system halts, then we say that n is recognized/accepted, and introduced in the set $N(\Pi)$, of all numbers accepted by Π.

By $DN_aIP_m(sym_r, anti_q)$ we denote the family of all sets $N(\Pi)$ of numbers accepted as above by deterministic P systems with at most m membranes, using symport rules of weight at most r and antiport rules of weight at most q, for $m \geq 1$ and $r, q \geq 0$. The letter I here is used for indicating that the number to be recognized is introduced in the initial configuration of the system (we will replace I by S in the case of using signals for introducing the number we want to analyse – see Section 4).

By NRE we denote the family of Turing computable sets of natural numbers (the length sets of recursively enumerable languages, and this is the origin of this notation). In the proofs given below (as well as in many papers dealing with accepting P systems with symport/antiport) we will use register machines as devices characterizing NRE, hence the Turing computability.

Informally speaking, a register machine consists of a specified number of registers which can hold any natural number, and which are handled according to a specified program consisting of labelled instructions; the registers can be increased or decreased by 1 – the decreasing being possible only if a register holds a number greater than or equal to 1 (we say that it is non-empty).

Formally, a deterministic *register machine* is a device $M = (m, B, l_0, l_h, R)$, where $m \geq 1$ is the number of registers, B is the set of instruction labels, l_0 is the initial label, l_h is the halting label, and R is the set of instructions labelled by elements from B (R is also called the *program* of the machine). The labelled instructions are of the following forms:

- $l_1 : (\texttt{ADD}(r), l_2)$ (add 1 to register r and go to the instruction with label l_2),
- $l_1 : (\texttt{SUB}(r), l_2, l_3)$ (if register r is not empty, then subtract 1 from it and go to the instruction with label l_2, otherwise go to the instruction with label l_3),
- $l_h : \texttt{HALT}$ (the halt instruction, which can only have the label l_h).

A register machine is used to recognize a number in the following manner: we start with all registers being empty, we introduce a number in a distinguished register (say, the first register), and we start computing with the instruction labelled by l_0; if the computation reaches the instruction l_h : HALT (that is, it halts), then the number is recognized, otherwise it is not recognized. The set of all numbers recognized by M is denoted by $N(M)$. It is known (see [4, 11]) that deterministic register machines (with three registers only, but this aspect is not important in what follows) recognize exactly the family NRE, of Turing computable numbers.

3 Universality in the Deterministic (Initial) Case

We start by recalling a result from [1] (Theorem 2 there), also giving the construction of the proof, because the idea on which this construction is based on is essentially used in the proofs below, too.

Theorem 1 $NRE = DN_aIP_1(sym_0, anti_2)$.

Proof. For a given register machine $M = (m, B, l_0, l_h, R)$ we consider the alphabet $U = \{a_1, \ldots, a_m\}$ (the symbol a_i is associated with register i and the contents of this register will be represented by the multiplicity of object a_i in the P system we are going to construct), and consider the P system

$$\begin{aligned}
\Pi &= (O, [_1\,]_1, l_0, O, R_1),\\
O &= U \cup \{l, l', l'', l''', l^{iv} \mid l \in B\},\\
R_1 &= \{(l_1, out; l_2 a_r, in) \mid \text{for } l_1 : (\texttt{ADD}(r), l_2) \in R\}\\
&\cup \{(l_1, out; l_1' l_1'', in),\\
&\quad (l_1' a_r, out; l_1''', in),\\
&\quad (l_1'', out; l_1^{iv}, in),\\
&\quad (l_1^{iv} l_1''', out; l_2, in),\\
&\quad (l_1^{iv} l_1', out; l_3, in) \mid \text{for } l_1 : (\texttt{SUB}(r), l_2, l_3) \in R\}.
\end{aligned}$$

We start with l_0 present in the system, and we introduce also a multiset a_1^n, for n being the number to be recognized. The system behaves like M, simulating its instructions.

Each ADD instruction of M is simulated by a unique rule of Π, while a SUB instruction $l_1 : (\texttt{SUB}(r), l_2, l_3)$ is simulated as follows. The available object l_1 is sent out, in exchange of l_1' and l_1''. The first object checks whether the register is non-empty, and in the affirmative case it exits the system (together with a copy of a_r) and is replaced by l_1'''; if no copy of a_r is available, then l_1' remains in the system and waits. The object l_1'' checks what the other object has done, allowing it a step for acting (this is the step when the rule $(l_1'', out; l_1^{iv}, in)$ is used). The object l_1^{iv} will find inside either one of the objects l_1' (if the register r was empty) and l_1''' (if the register r was not empty), and in each case the next

object to be introduced in the system, l_3 or l_2, respectively, is the correct label to be used in the program of M.

Clearly, the system Π halts if and only if M halts for the same input n, hence $N(M) = N(\Pi)$. ■

This result corresponds to the universality of P systems with one membrane and antiport rules of weight 2, working non-deterministically (in the generating or recognizing mode) – see, [3, 7, 8], etc. It is also known that non-deterministic P systems with one membrane and symport rules of weight 3 are universal. This is true for accepting P systems in the deterministic case, too.

Theorem 2 $NRE = DN_aIP_1(sym_3, anti_0)$.

Proof. We start again by considering a given register machine $M = (m, B, l_0, l_h, R)$ and an alphabet $U = \{a_1, \ldots, a_m\}$, and construct the P system

$$
\begin{aligned}
\Pi &= (O, [\,_1\,]_1, l_0 w_1, E, R_1),\\
O &= U \cup \{l, l', l'', l''', l^{iv}, l^v, l^{vi}, l^{vii}, l^{viii}, l^{ix} \mid l \in B\},\\
w_1 &\text{ contains one occurrence of each } l', l^{iv}, l^v, l^{viii}, l^{ix} \text{ for each } l \in B,\\
E &= U \cup \{l, l'', l''', l^{vi}, l^{vii} \mid l \in B\},\\
R_1 &= \{(l_1 l_1', out),\\
&\qquad (l_1' l_2 a_r, in) \mid \text{for } l_1 : (\texttt{ADD}(r), l_2) \in R\}\\
&\cup \{(l_1 l_1', out),\\
&\qquad (l_1' l_1'' l_1''', in),\\
&\qquad (l_1'' l_1^{iv} a_r, out),\\
&\qquad (l_1''' l_1^v, out),\\
&\qquad (l_1^{iv} l_1^{vi}, in),\\
&\qquad (l_1^v l_1^{vii}, in),\\
&\qquad (l_1^{vii} l_1'' l_1^{viii}, out),\\
&\qquad (l_1^{vii} l_1^{vi} l_1^{ix}, out),\\
&\qquad (l_1^{viii} l_3, in),\\
&\qquad (l_1^{ix} l_2, in) \mid \text{for } l_1 : (\texttt{SUB}(r), l_2, l_3) \in R\}.
\end{aligned}
$$

We start again with l_0 present in the system (as well as with copies of $l', l^{iv}, l^v, l^{viii}, l^{ix}$, for all $l \in B$).

An instruction $l_1 : (\texttt{ADD}(r), l_2)$ is simulated in Π in two steps: l_1 exits together with the "carrier" l_1', which comes back together with the suitable label l_2 and one copy of a_r. (Note that because each $l \in B$ appears only once as the label of an instruction, although l_1'', l_1''' are present in the environment, the rule $(l_1' l_1'' l_1''', in)$ is not available also for l_1 being the label of an ADD instruction, hence no non-determinism appears.)

The case of subtract instructions is more complicated. Take $l_1 : (\texttt{SUB}(r), l_2, l_3)$. Again l_1 exits together with the "carrier" l_1', which brings into the system the object l_1'', which

checks whether any a_r is present, and the object l_1''', which checks what the former object has done. Specifically, in the next step l_1''' exits together with l_1^v, while l_1'' has the necessary time for checking whether any a_r exists; in the affirmative case, it exits together with l_1^{iv}, in the negative case it waits in the system. In the next step l_1^v returns with l_1^{vii} and, if it went out, l_1^{iv} returns with l_1^{vi}. Now, l_1^{vii} will go out either together with l_1'' (which means that the register was empty), or with l_1^{vi} (which means that the register was not empty). In the former case, also l_1^{viii} exits, in the latter case also l_1^{ix} exits, and these objects return with the suitable label l_3, l_2, respectively, as necessary for the correct continuation of the computation.

The process can be iterated (after completing the simulation of each instruction of M, all objects different from those in $U \cup B$ are back in the place where they were in the initial configuration), and the computation of Π halts if and only if the computation of M halts, hence $N(M) = N(\Pi)$. ■

For P systems working in the generating mode, it is also known that the universality holds when symport rules of weight 2 are used, provided that two membranes are used. Of course, the proof is based on a non-deterministic system. It is an *open problem* whether or not a counterpart of this result holds true also for deterministic systems working in the accepting mode. We have some doubts that this is the case, and, if this conjecture would be confirmed, then this would be a nice result: a class of P systems would have been found where the non-deterministic systems are more powerful than the deterministic ones.

4 Introducing the Input by Using Signals

The case discussed in the previous sections can be illustrated as in Figure 1(a), where, as mentioned in the Introduction, the "user command" is assumed to be introduced in the initial configuration of the system, and the system starts immediately to work. Another possibility is as illustrated in Figure 1(b), with the system starting independently of any input, and a signal s appears in the environment sometimes during the computation. If this happens in step n, then this is as requesting from the system to analyse the number $n-1$.

Formally, an accepting P system *with one signal* is a usual P system $\Pi = (O, \mu, w_1, \ldots, w_m, E, R_1, \ldots, R_m)$, with O containing a distinguished object s which is present in no multiset $w_i, 1 \leq i \leq m$, and also $s \notin E$. A number n is accepted by Π if the system starts from the initial configuration, proceeds computing, s appears in the environment and is brought into the system in step $n+1$, and then Π halts in one of the subsequent steps. As usual, we denote by $N(\Pi)$ the set of numbers accepted in this way by Π, and by $DN_aSP_m(sym_r, anti_q)$ we denote the family of sets $N(\Pi)$ accepted by deterministic systems with at most m membranes, using symport rules of weight at most r and antiport rules of weight at most q, for $m \geq 1$ and $r, q \geq 0$, with the input introduced by signals.

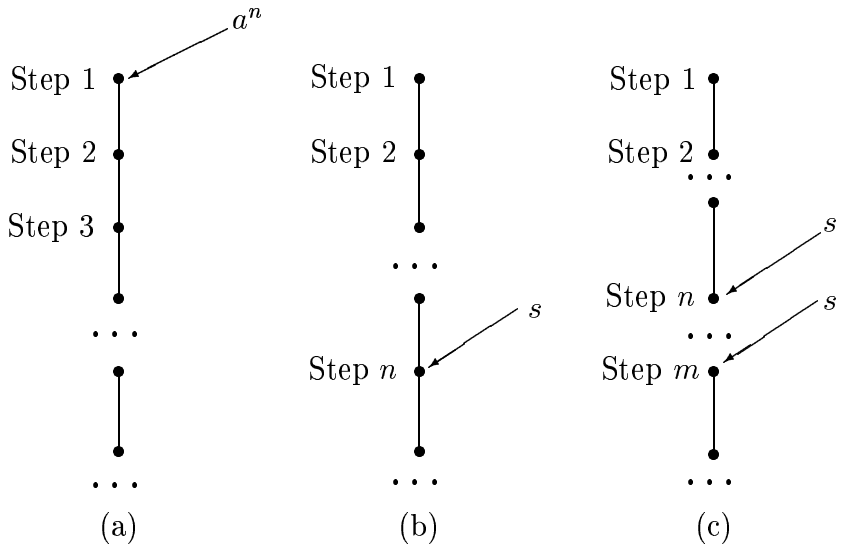

Figure 1: Initial input and input by signals

It is important to note that s appears only once in the environment, after its appearance we know that no further copy of it will ever appear.

Not very unexpected, also this mode of introducing the input leads to universality results – but the proofs are significantly more difficult than in the case of an initial input. The difficulty comes from the fact that we have to check in each time unit whether or not s has appeared in the environment, at the same time counting the steps, in order to know which is the number to be recognized. The checking can be done in the same way as in the proofs of Theorems 1 and 2; instead of checking whether a register r is empty, we check whether s is present, but two problems appear: the checking should be done in the environment, and in each step, while in the proofs of the above theorems the check is done inside the system, in step 2 of simulating the subtract instructions and then we needed several steps for completing the check. This means that in our case we have to use in parallel several independent "checkers", started in consecutive time units, so that in each time unit one checker looks for s in the environment. When one of these checkers will find s, it has to stop all the other checkers, and make the system start to simulate a register machine which has to recognize a number determined by the time when the signal has appeared. This strategy will be essentially used in the proofs below.

We start with a counterpart of Theorem 1.

Theorem 3 $NRE = DN_aSP_1(sym_0, anti_2)$.

Proof. The way to simulate a register machine $M = (m, B, l_0, l_h, R)$ by means of a P system starting with a_1^n in the skin region, using antiport rules of weight 2, is known from the proof of Theorem 1, so that in what follows we only describe the way of producing the initial label l_0 and the multiset a_1^n in the case when s appears in the environment in step $n+1$.

We start with the objects $c_1'c_1''d$ in the unique membrane of the system; all objects necessary below for applying the rules are supposed to be present in the environment – with the experience of the previous proofs and with the explanations we give, the reader should be able to fill in the missing technical details.

In the first step we use the following rules (so, instead of giving the whole set of rules, we present them in combinations, as they are "collaborating" during the computation):

$$(c_1', out; sc_1''', in), \quad (c_1'', out; c_1^{iv}a_1, in), \quad (d, out; c_2'c_2'', in).$$

In this way, if s is present in the environment, c_1 and its primed versions sense it; in parallel, c_1'' is exchanged for c_1^{iv} and also a copy of a_1 is brought in, while the auxiliary object d just brings inside the objects c_2', c_2'', which have the role of checking the appearance of s in the even steps.

Specifically, in the next step – hence in any even subsequent step – we use the following rules:

$$(c_1'''c_2', out; f, in), \quad (c_1^{iv}c_1', out; c_1'c_1'', in), \quad (c_2', out; sc_2''', in), \quad (c_2'', out; c_2^{iv}a_1, in).$$

If c_1' has remained inside (hence s was not detected in the previous step), then c_1^{iv} goes outside together with it and $c_1'c_2''$ return to the system, thus iterating the procedure, and making possible the checking of s in the next step. Simultaneously, c_2 and its primed variants perform a similar sequence of operations, checking whether s has appeared in that (even) moment. We will discuss later the case when the signal was detected, and we indicate the rules which can be used in any odd step but the first one, which was different, because of using the rule $(d, out; c_2'c_2'', in)$.

The rules used in any odd step are:

$$(c_1', out; sc_1''', in), \quad (c_1'', out; c_1^{iv}a_1, in), \quad (c_2'''c_1', out; f, in), \quad (c_2^{iv}c_2', out; c_2'c_2'', in).$$

Note the fact that also $c_2'c_2''$ return to the system in odd steps, providing that s was not detected in the previous step in the environment. It is also important to note that in each step one copy of a_1 is introduced in the system, in odd steps together with c_1^{iv} and in even steps together with c_2^{iv}.

Assume now that s appears. If this happens in an odd step, then c_1''' brings it in the system. In the next step c_1''' exits together with c_2' (which is waiting in the system: because s has appeared, there is no way to use the rule $(c_2', out; sc_2''', in)$), and f is brought in. Simultaneously, c_2'' leaves the system and c_2^{iv} comes in together with one further copy of a_1, but c_2^{iv} will remain in the system, because c_2' was eliminated. The two checkers have completed their job, we know that s has appeared, f indicates this – but we have two more copies of a_1 than necessary (one introduced in the last step and one together with s, in step n, in total $n+1$ copies, while we have to analyse the number $n-1$). The following rules

$$(fa_1, out; f', in),$$
$$(f'a_1, out; l_0, in),$$

both eliminate the two superfluous copies of a_1 and introduce the initial label of the register machine.

Add now to the system all rules from the construction in the proof of Theorem 1. The obtained system will halt if and only if the register machine M halts when starting with $n-1$ in its first register, and this completes the proof. ■

Also a counterpart of Theorem 2 is true, not using only one membrane, but two – and without being able to recognize the number 0. We believe that one membrane only is not sufficient in this case, because we cannot start several checkers (at least two) in consecutive steps when using only symport rules: we cannot keep one checker in the environment, like in the previous proof, because, on the one hand, the respective symbols should be present in the environment in arbitrarily many copies, and, on the other hand, by symport rules we will bring inside objects again and again, hence the computation will never stop. Moreover, we cannot send outside a "carrier", to come back with the necessary objects, because this requests two steps, hence the second checker will be available only from the third step on, with no possibility to check the appearance of the signal in the second step.

Of course, if we ignore the case of having s in step 2 (hence never computing the number 1), or if we change the definition of the systems we work with (for instance, allowing in the environment objects with a finite specified multiplicity), then we can overpass this difficulty and one membrane will suffice. Here we remain in the framework of the above definitions (where we *conjecture* that deterministic P systems with one membrane are strictly less powerful as the non-deterministic ones – in the analysing case, with the input introduced by a signal), and we give the following result only, where NRE' denotes the family of Turing computable non-null numbers.

Theorem 4 $NRE' = DN_aSP_2(sym_3, anti_0)$.

Proof. We proceed again as in the proof of Theorem 3, by indicating how we can check whether the signal has appeared or not in the environment, and, in the affirmative case, stopping the checkers and turning the system to the simulation of a register machine. This time we have to use checkers as suggested by the proof of Theorem 2, hence with more than two steps needed for completing a subtraction instruction, hence a checking. Therefore, we need more than two checkers, starting each one in consecutive steps. Actually, four checkers are sufficient, each one based on objects c_1, c_2, c_3, c_4, respectively, with several primed variants for each of them (from c_i' to c_i^{iv} in each case).

The system we construct will have two membranes (hence a membrane structure of the form $[_1[_2\,]_2]_1$), with the following multisets present in the two regions

$$w_1 = d_1c_1'c_1''c_2'c_2''c_3'c_3''c_4'c_4''d_3'd_4''g,$$
$$w_2 = d_2d_3d_4d_4'e,$$

and with the following rules associated with the two membranes (to the set of rules associated with the skin membrane we also have to add the rules used for simulating the

register machine, that is why we denote the set below by R_1'; the set R_2 is complete as given here):

$$
\begin{aligned}
R_1' &= \{(ge, out), \\
&\quad (gl_0, in)\} \\
&\cup \{d_i c_i' c_i'', out), \\
&\quad (c_i' s f_i, in), \\
&\quad (c_i'' c_i''' a_1, in), \\
&\quad (c_i''' c_i^{iv}, out), \\
&\quad (c^{iv} c_i' d_i, in) \mid i = 1, 2, 3, 4\}, \\
R_2 &= \{(d_2, out), \\
&\quad (d_3', in), \\
&\quad (d_3 d_3', out), \\
&\quad (d_4', out), \\
&\quad (d_4' d_4'', in), \\
&\quad (d_4'' d_4, out)\} \\
&\cup \{(s c_i'' a_1, in) \mid i = 1, 2, 3, 4\} \\
&\cup \{(f_i c_j'' a_1, in) \mid i = 1, 2, 3, 4, j = (i+1) mod\ 4 \text{ or } j = (i+3) mod\ 4\} \\
&\cup \{(f, out), \\
&\quad (se, out)\}.
\end{aligned}
$$

Let us first see how the four checkers, each one realized by an object $c_i, i = 1, 2, 3, 4$, and its primed variants, become active in the first four steps of a computation. Because $d_1 c_1' c_1''$ are present in the skin region, in the first step they exit the system, hence in the next step the signal s can be detected. At the same time, d_2 gets out of the inner membrane, hence in step 2 also $d_2 c_2' c_2''$ can exit, thus being able to detect the presence of s in step 3. In step 1, also d_3' moves from the skin region to the inner region, in the step 2 it comes out together with d_3, hence in step 3 the third checker is activated (that is, $d_3 c_3' c_3''$ exit the system). A similar process is carried out for the fourth checker, starting with d_4' in membrane 2; in step 3 the object d_4 is released in the skin region, hence in step 4 this last checker is activated.

Let us now see how the presence of s is detected. Assume that in some step, some $d_i c_i' c_i''$ were sent out, and assume that s is not present in the next step. Then c_i'' comes back, together with $c_i''' a_1$, then c_i''' exits together with the "carrier" c_i^{iv}, which brings in the objects $c_i' d_i$ which were waiting in the environment. Because c_i'' was already in the system (it is important to note that it has waited two steps inside), the rule $(d_i c_i' c_i'', out)$ can be used again, and the process is iterated.

Assume now that at some step $n+1$ we have the object s in the environment (only one copy appears). In this case, the rule $(c_i' s f_i, in)$ can be used, for the c_i' sent to the environment in the previous step. At the same time, c_i'' comes in, together with c_i''' (and

one further copy of a_1), c_i''' exits together with c_i^{iv} and c_i'' is carried into the inner membrane by s (together with a copy of a_1); after that, c_i^{iv} cannot come back, because c_i' is no longer present in the environment, hence d_i remains in the environment, and this checker will no longer work.

Because s was already produced, no further copy of it will ever appear. In the next step c_{i+1}'' enters the system, and, because no other rule can use it, the rule $(f_i c_{i+1}'' a_1, in) \in R_2$ will "hide" it in the inner membrane; at the same time s exits membrane 2 together with the unique copy of object e. In the next step, a third object c_j'' is brought into the inner membrane by means of s – which also remains in the inner membrane, because we had here only one copy of e. Simultaneously, the object f_i again exits membrane 2, and in the next step will bring here the fourth object c_j'', thus interrupting the work of all checkers.

With each of the four objects of the form c_j'' brought into the inner membrane, also a copy of a_1 was introduced there, thus leaving in the skin region the right number of copies, as necessary for starting the simulation of a register machine asked to recognize the number n. The starting label l_0 of the register machine is introduced as follows: in the first step after introducing s into the system, s enters membrane 2; in the next step, it exits together with e; in the third step, e exits the system together with g; in the fourth step, g returns together with l_0. Therefore, l_0 is present only after removing the four exceeding copies of a_1.

From now on, we proceed as in the proof of Theorem 2 and simulate the work of the register machine, using only the skin region. ■

In the same way as the problem whether the universality can be obtained for systems using symport rules of weight two (and possibly further membranes) has remained open for the case of the initial input, this problem remains *open* for the case of using signals for introducing the input, too.

5 Using Two Signals

The previous model is somewhat "unrealistic", because it is supposed that both the system and the "user" start counting at the same time. However, it is natural to suppose that the system is independent of the "user", it just evolves continuously, checking the appearance of a signal and doing nothing else before detecting the signal. Thus, an attractive variant would be to have two signals, one appearing in the moment when the "user" announces the intention to compute something (to recognize a number) and the second one appearing in the moment when the input is defined: the number to be analysed is the difference between the time units when the two signals appear (minus one, to be able to deal with number zero). It is supposed that the two signals do not appear in the same moment.

Figure 1(c) illustrates this way of introducing the input.

Two cases can be distinguished: using two different signals, s_1 and s_2, or using a unique signal, s, appearing twice.

The first case is much easier than the second one: we just repeat the constructions from the proofs of Theorems 3 and 4, with a double number of checkers, a "team" of checkers

looking for s_1 and the second "team" looking for s_2; of course, the checkers should work with distinct objects, not to interfere before finishing their work; the first "team" of checkers is counting by adding objects a_2, the second one by introducing into the system copies of a_1. Assume that we have detected both signals, and we have in the system n copies of a_2 and m copies of a_1, for some $m, n \geq 1$. Instead of introducing the label l_0, for starting the work of a register machine, we introduce the label g_0, corresponding to the following register machine subprogram:

$$g_0 : (\mathtt{SUB}(2), g_1, g_3),$$
$$g_1 : (\mathtt{SUB}(1), g_2, g_0),$$
$$g_2 : (\mathtt{ADD}(1), g_2),$$
$$g_3 : (\mathtt{SUB}(1), g_2, l_0).$$

This subprogram subtracts n from m, leaving the result in the first register; in the case when $n > m$ the computation never stops, but in the opposite case one subtracts one further unit and then one introduces the label l_0, for starting the work of the register machine which analyses the number $m - n - 1$.

The above subprogram can be simulated by a P system of the same complexity as in the proofs of Theorems 3 and 4, hence we can state the following result (S_2 indicates that we use two different signals, in the way specified above.

Theorem 5 $NRE = DN_aS_2P_1(sym_0, anti_2) = DN_aS_2P_2(sym_3, anti_0)$.

The case of a unique signal appearing twice, in two different time units, can be handled in the same way, but we have to increase the weight of the used rules. For instance, we have the following result (this time, we use S_2' to indicate that the same signal is produced twice).

Theorem 6 $NRE = DN_aS_2'P_1(sym_0, anti_5) = DN_aS_2'P_2(sym_6, anti_0)$.

The idea of the proof is to have again two "teams" of checkers, the first one looking for the first occurrence of s and the second one looking for the second occurrence of s. The first "team" does not introduce objects a_1, but just triggers the start of the work of the second "team", which works exactly as in the proofs of Theorems 3 and 4.

Consider first the case of antiport rules. Assume that the first group of checkers uses the objects c_1, c_2, with primes, and the second one uses the objects e_1, e_2, with primes. Instead of the rules $(c_1', out; sc_1''', in)$, $(c_2', out; sc_2''', in)$, used when s is detected, we consider the rules

$$(c_1', out; sc_1'''e_1'e_1''g), \quad (c_2'', out; sc_2'''e_1'e_2''g),$$

which make available the objects e_1', e_2'' which start the work of the first checker from the second "team" of triggers. The object g is used in the same way as d in the proof of Theorem 3, for bringing from the environment $e_2'e_2''$, the objects necessary to start the work of the second checker of the second "team" of checkers, in the next step. From now

on, we continue as in the proof of Theorem 3, hence the first equality in the theorem holds.

The second equality can be obtained in the same way, starting from the construction in the proof of Theorem 4.

We do not see how we can decrease the weight of the antiport rules back to two and still be able to check whether the second signal appears immediately after the first signal; if in between the two signals we have at least two steps, then we can use again antiport rules of weight two. Thus, either we have to ignore the numbers 0, 1, 2, or we have to change the definition in the sense that if the two signals appear at times n and m, with $m - n \geq 2$, then the number to analyse is $m - n - 2$. We leave to the reader the task to check the details – as well as the question of decreasing back to three (if possible) the weight of symport rules.

6 Final Remarks

We have addressed the important topic from automata theory, whether deterministic accepting P systems with symport/antiport rules are equally powerful as non-deterministic systems. Two cases are considered: with the number to be analysed being introduced in the initial configuration as the multiplicity of a specified object, or with this number introduced by a signal, appearing in the environment at a time which specifies the number. Universality results are obtained in all cases, by systems with a small number of membranes (one in all but one case), and in most cases with either only symport rules of weight 3 or only antiport rules of weight 2.

Besides their mathematical interest, such results are also important for possible implementations of these types of P systems, because of the difficulty of implementing/simulating non-deterministic computing devices on the usual computer. Furthermore, deterministic accepting P systems can be useful in a framework as that from [1], where by accelerating such systems in ways already known for Turing machines or in new ways suggested by biological observations one can get computing devices able to compute more than Turing machines (e.g., solving the halting problem).

The question of considering other classes of P systems from the point of view of the deterministic versus non-deterministic variants remains to be investigated; a case of a special interest is that of catalytic systems, known to be universal (see [6]), but with proofs which involve a high degree of non-determinism.

References

[1] C.S. Calude, Gh. Păun, Bio-steps beyond Turing, CDMTCS Res. Report 226, Auckland Univ., November 2003.

[2] E. Csuhaj-Varjú, Gy. Vaszil, P automata or purely communicating accepting P systems, in [13], 219–233.

[3] R. Freund, C. Martín-Vide, A. Obtulowicz, Gh. Păun, On three classes of automata-like P systems. *Proc. 7th Int. Conf. DLT2003* (Z. Ésik, Z. Fülöp, Eds.), Szeged, Hungary, 2003, Lecture Notes in Computer Science 2710, Springer-Verlag, Berlin, 2003 292–303.

[4] R. Freund, M. Oswald: GP systems with forbidding context, *Fundamenta Informaticae*, **49**, 1-3 (2002), 81–102.

[5] R. Freund, M. Oswald, A short note on analyzing P systems with antiport rules, *Bulletin of the EATCS*, 78 (2002), 231–236.

[6] R. Freund, L. Kari, M. Oswald, P. Sosík, Computationally universal P systems without priorities: two catalysts are sufficient. *To appear in TCS.*

[7] R. Freund, A. Păun, Membrane systems with symport/antiport rules: universality results, in [13], 270–287.

[8] P. Frisco, H.J. Hoogeboom, Simulating counter automata with P systems with symport/antiport, in [13], 288–301.

[9] O.H. Ibarra, On the computational complexity of membrane computing systems. *Submitted.*

[10] M. Madhu, K. Krithivasan, On a class of P automata. *Submitted.*

[11] M. Minsky, *Computation. Finite and Infinite Machines*, Prentice Hall, Englewood Cliffs, NJ, 1967.

[12] Gh. Păun, *Membrane Computing. An Introduction*, Springer-Verlag, Berlin, 2002.

[13] Gh. Păun, G. Rozenberg, A. Salomaa, C. Zandron, eds., *Membrane Computing. International Workshop, WMC 2002, Curtea de Argeş, Romania*, Lecture Notes in Computer Science, 2597, Springer-Verlag, Berlin, 2002.