On Flip-Flop Membrane Systems with Proteins

Andrei Păun[1,2], Alfonso Rodríguez-Patón[2]

[1] Department of Computer Science/IfM
Louisiana Tech University
P.O. Box 10348, Ruston, LA 71272, USA
apaun@latech.edu

[2] Universidad Politécnica de Madrid - UPM
Facultad de Informática
Campus de Montegancedo S/N, Boadilla del Monte, 28660 Madrid, Spain
arpaton@fi.upm.es

Summary. We consider once more the membrane systems with proteins on membranes. This model is bridging the membrane systems and brane calculi areas together, thus it is interesting to study it in more depth. We have improved previous results in the area and also defined a new variant of these systems based on time as the output of the computation. The new model allows (due to its flexibility) even stronger improvements with respect to the number of proteins needed to perform the computation.

1 Introduction

We continue the work on a membrane systems model combining membrane systems and brane calculi as introduced in [14]. In brane calculi introduced in [5], one works only with objects – called proteins – placed on membranes, while the evolution is based on membrane handling operations, such as exocytosis, phagocytosis, etc. In the membrane computing area we have rules associated with each region defined by a membrane, and in the recent years the rules in membrane computing have been considered mainly to work on symbol objects rather than other structures such as strings. The extension considered in [14] and in [15] was to have both types of rules (both at the level of the region delimited by membranes and also at the level of membrane controlled by a protein). The reason for considering both extensions was that in biology, many reactions taking place in the compartments of living cells are controlled/catalysed by the proteins embedded in the membranes bilayer. For instance, it is estimated that in the animal cells, the proteins constitute about 50% of the mass of the membranes, the rest being lipids and small amounts of carbohydrates. There are several types of such proteins embedded in the membrane of the cell; one simple classification places these proteins into two classes, that of integral proteins (these molecules can "work" in both inside the membrane as well as also in the region outside the membrane), and that of peripheral proteins (macromolecules that can only work in one region of the cell) – see [1].

In the present paper we continue the discussion in the direction of membrane systems with proteins, but we extend the model to have also a "more natural" output of the computation with ideas from [8].

Briefly, the systems that we consider in this paper extend the original definition by using the paradigm of time as the output of a computation as previously introduced in [6] and [8]. The idea originates in [17] as Problem W; the novelty is that instead of the "standard" way to output, like the multiplicities of objects found at the end of the computation in a distinguished membrane as it was defined in the model from [14] and in [15], it seems more "natural" to consider certain *events* (i.e., configurations) that may occur during a computation and to relate the output of such a computation with the time interval between such distinguished configurations. Our system will compute a set of numbers similarly with the case of "normal" symport/antiport systems as defined in [14], but the benefit of the current setting is that the computation and the observance of the output are now close to the biology and to the tools used for cell biology (fluorescence microscopy, FACS).

2 The Types of Rules in the System

In what follows we assume that the reader is familiar with membrane computing basic elements, e.g., from [16] and from [19], as well as with basic elements of computability, so that we only mention here a few notations we use. The rules based on proteins on membranes were described in detail in [14], and we refer the interested reader to that publication and to [15] for further details.

As usual, we represent multisets of objects from a given alphabet V by strings from V^*, and the membrane structures by expressions of correctly matching labeled parentheses. The family of recursively enumerable sets of natural numbers is denoted by NRE.

In the P systems which we consider below, we use two types of objects, *proteins* and usual *objects*; the former are placed **on** the membranes, the latter are placed **in** the regions delimited by membranes. The fact that a protein p is on a membrane (with label) i is written in the form $[_i p|$. Both the regions of a membrane structure and the membranes can contain multisets of objects and of proteins, respectively.

We consider the following types of rules for handling the objects and the proteins; in all of them, a, b, c, d are objects, p is a protein, and i is a label ("cp" stands for "change protein"), where p, p' are two proteins (possibly equal; if $p = p'$, then the rules of the type *cp* become rules of the type *res*; i.e. restricted):

Type	Rule	Effect (besides changing also the protein)
1cp	$[_i p\|a \to [_i p'\|b$	
	$a[_i p\| \to b[_i p'\|$	modify an object, but not move
2cp	$[_i p\|a \to a[_i p'\|$	
	$a[_i p\| \to [_i p'\|a$	move one object unmodified
3cp	$[_i p\|a \to b[_i p'\|$	
	$a[_i p\| \to [_i p'\|b$	modify and move one object
4cp	$a[_i p\|b \to b[_i p'\|a$	interchange two objects
5cp	$a[_i p\|b \to c[_i p'\|d$	interchange and modify two objects

An intermediate case between *res* and *cp* can be that of changing proteins in a restricted manner, by allowing at most two states for each protein, $p, \bar{p}$, and the rules working either in a *res* manner (without changing the protein), or changing it from p to $\bar{p}$ and back (like in the case of bistable catalysts). Rules with such flip-flop proteins are denoted by $nff, n = 1, 2, 3, 4, 5$ (note that in this case we allow both rules which do not change the protein and rules which switch from p to $\bar{p}$ and back).

Both in the case of rules of type ff and of type *cp* we can ask that the proteins are always moved in another state (from p into $\bar{p}$ and vice versa for *ff*). Such rules are said to be of *pure* ff or *cp* type, and we indicate the use of pure ff or *cp* rules by writing ffp and *cpp*, respectively.

We can use these rules in devices defined in the same way as the symport/antiport P systems (hence with the environment containing objects, in arbitrarily many copies each – we need such a supply of objects, because we cannot create objects in the system), where also the proteins present on each membrane are mentioned.

That is, a *P system with proteins on membranes* is a device of the form

$$\Pi = (O, P, \mu, w_1/z_1, \ldots, w_m/z_m, E, R_1, \ldots, R_m, i_o),$$

where:

1. m is the degree of the system (the number of membranes);
2. O is the set of objects;
3. P is the set of proteins (with $O \cap P = \emptyset$);
4. μ is the membrane structure;
5. $w_1, \ldots, w_m$ are the (strings representing the) multisets of objects present in the m regions of the membrane structure μ;
6. $z_1, \ldots, z_m$ are the multisets of proteins present on the m membranes of μ;
7. $E \subseteq O$ is the set of objects present in the environment (in an arbitrarily large number of copies each);
8. $R_1, \ldots, R_m$ are finite sets of rules associated with the m membranes of μ;
9. i_o is the output membrane, an elementary membrane from μ.

The rules can be of the forms specified above, and they are used in a non-deterministic maximally parallel way: in each step, a maximal multiset of rules is used, that is, no rule can be applied to the objects and the proteins which remain unused by the chosen multiset. As usual, each object and each protein can be involved in the application of only one rule, but the membranes are not considered as involved in the rule applications, hence the same membrane can appear in any number of rules at the same time.

If, at one step, two or more rules can be applied to the same objects and proteins, then only one rule will be non-deterministically chosen. At each step, a P system is characterized by a configuration consisting of all multisets of objects and proteins present in the corresponding membranes (we ignore the structure μ, which will not be changed, and the objects from the environment). For example, $C = w_1/z_1, \ldots, w_m/z_m$ is the initial configuration, given by the definition of the P system. By applying the rules in a non-deterministic maximally parallel manner, we obtain transitions between the configurations of the system. A finite sequence of configurations is called computation. A computation halts if it reaches a configuration where no rule can be applied to the existing objects and proteins.

Only halting computations are considered successful, thus a non-halting computation will yield no result. With a halting computation we associate a result, in the form of the multiplicity of objects present in region i_o in the halting configuration. We denote by $N(\Pi)$ the set of numbers computed in this way by a given system Π. (A generalization would be to distinguish the objects and to consider vectors of natural numbers as the result of a computation, but we do not examine this case here.)

We denote, in the usual way, by $NOP_m(pro_r;$*list-of-types-of-rules*$)$ the family of sets of numbers $N(\Pi)$ generated by systems Π with at most m membranes, using rules as specified in the list-of-types-of-rules, and with at most r proteins present on a membrane. When parameters m or r are not bounded, we use $*$ as a subscript.

The new definition introduced by the current paper is the addition of time to the above model, in brief, *P system with proteins on membranes and time* is a device of the form

$$\Pi = (O, P, \mu, w_1/z_1, \ldots, w_m/z_m, E, R_1, \ldots, R_m, C_{start}, C_{stop}),$$

where:

1. $m,\ O,\ P,\ \mu,\ w_1, \ldots, w_m,\ z_1, \ldots, z_m,\ E,\ R_1, \ldots, R_m$ are as defined above;
2. $C_{start},\ C_{stop}$ are regular subsets of $(O^*)^m$, describing configurations of Π. We will use a regular language over $O \cup \{\$\}$ to describe them, the special symbol $\$ \notin O$ being used as a marker between the configurations[3] in the different regions of the system. More details are given in [8] and [12].

[3] We express by these configurations restrictions that need to be satisfied by each of the current multisets in their respective regions so that the overall configuration can be satisfied.

As an example for the C_{start} and C_{stop} configurations, let us give the following restriction[4] $C = b^3d^7(O - \{a, b, d\})^*$ for a single membrane (the proofs obtained below need only one membrane, thus we can simplify the notation by not using the symbol \$). This means that in the region delimited by the only membrane in the system, the configuration C is satisfied if and only if we do not have any symbol of type a, we must have exactly 3 symbols of type b and exactly 7 symbols of type d. Any other symbol not mentioned is not restricted, e.g. we can have any number of symbols of type c.

We emphasize the fact that in the definition of Π we assume that C_{start} and C_{stop} are regular. Other, more restrictive, cases can be of interest but we do not discuss them here.

We can now denote the systems as above based on time with $NTOP_m(pro_r;$*list-of-types-of-rules*$)$ the family of sets of numbers $N(\Pi)$ generated by systems Π with at most m membranes, using rules as specified in the list-of-types-of-rules, and with at most r proteins present on a membrane. When parameters are not bounded we replace them by $*$.

3 Register Machines

In the proofs from the next sections 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 (counters) which can hold any natural number, and which are handled according to a program consisting of labeled 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) –, and checked whether they are non-empty.

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

- $l_1 : (\texttt{ADD}(r), l_2, l_3)$, $1 \leq r \leq m$ (add 1 to register r and go non-deterministically to one of the instructions with labels l_2, l_3),
- $l_1 : (\texttt{SUB}(r), l_2, l_3)$, $1 \leq r \leq m$ (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).

We say that a register machine has no ADD instructions looping to the same label (or *without direct loops*) if there are no instructions of the form

[4] C can be written also in the following form $C = (a^0b^3d^7)$

$l_1 : (\texttt{ADD}(r), l_1, l_2)$ or $l_1 : (\texttt{ADD}(r), l_2, l_1)$ in R. For instance, an instruction of the form $l_1 : (\texttt{ADD}(r), l_1, l_2)$ can be replaced by the following instructions, where l_1' is a new label: $l_1 : (\texttt{ADD}(r), l_1', l_2)$, $l_1' : (\texttt{ADD}(r), l_1, l_2)$. The generated set of numbers is not changed.

A register machine generates a natural number in the following manner: we start computing with all m registers being empty, with the instruction labeled by l_0; if the computation reaches the instruction $l_h : \texttt{HALT}$ (we say that it halts), then the values of register 1 is the number generated by the computation. The set of numbers computed by M in this way is denoted by $N(M)$. It is known (see [11]) that non-deterministic register machines with three registers generate exactly the family NRE, of Turing computable sets of numbers. Moreover, without loss of generality, we may assume that in the halting configuration all registers except the first one, where the result of the computation is stored are empty.

4 Previous Results

In [14] the following results were proved:

Theorem 1.

$$\begin{aligned} NOP_1(pro_2; 2cpp) &= NRE. \quad \textit{(Theorem 5.1 in [14])} \\ NOP_1(pro_*; 3ffp) &= NRE. \quad \textit{(Theorem 5.2 in [14])} \\ NOP_1(pro_2; 2res, 4cpp) &= NRE. \quad \textit{(Theorem 6.1 in [14])} \\ NOP_1(pro_2; 2res, 1cpp) &= NRE. \quad \textit{(Theorem 6.2 in [14])} \\ NOP_1(pro_*; 1res, 2ffp) &= NRE. \quad \textit{(Theorem 6.3 in [14])} \end{aligned}$$

As an extension of the work reported in [14], a significant amount of energy was devoted to the flip-flopping variant of these membrane systems which resulted in the paper [9]. S.N Krishna was able to prove several results in [9] improving Theorem 5.2, and Theorem 6.3 from [14]:

Theorem 2.

$$\begin{aligned} NOP_1(pro_7; 3ffp) &= NRE. \quad \textit{(Theorem 1 in [9])} \\ NOP_1(pro_7; 2ffp, 4ffp) &= NRE. \quad \textit{(Theorem 2 in [9])} \\ NOP_1(pro_7; 2ffp, 5ffp) &= NRE. \quad \textit{(Corollary 3 in [9])} \\ NOP_1(pro_{10}; 1res, 2ffp) &= NRE. \quad \textit{(Theorem 4 in [9])} \\ NOP_1(pro_7; 1ffp, 2ffp) &= NRE. \quad \textit{(Theorem 6 in [9])} \\ NOP_1(pro_9; 1ffp, 2res) &= NRE. \quad \textit{(Theorem 7 in [9])} \\ NOP_1(pro_9; 2ffp, 3res) &= NRE. \quad \textit{(Theorem 9 in [9])} \\ NOP_1(pro_8; 1ffp, 3res) &= NRE. \quad \textit{(Theorem 10 in [9])} \\ NOP_1(pro_9; 3res, 4ffp) &= NRE. \quad \textit{(Theorem 11 in [9])} \end{aligned}$$

$$NOP_1(pro_8; 2ffp, 5res) = NRE. \quad \textit{(Theorem 13 in [9])}$$

A close reading of the theorems mentioned above will yield some improvements that are given in the following section.

5 New Results

We start this section by first discussing the results from [9] which we mentioned in Theorem 2. The main idea in all the proofs reported in [9] was to simulate register machines (it is known that such devices with 3 registers are universal). The novelty of the proof technique in [9] was to consider for all ADD instructions associated with a particular register a single protein, similarly we use one protein for all the SUB instructions associated with a specific register. Thus in the proofs of the results mentioned in Theorem 2 we will have 6 proteins used for the simulation of the instructions in the register machine, (both ADD and SUB instructions for the 3 registers in the machine) the other(s) protein(s) being needed mainly for the test with zero processing in the simulation of SUB instructions.

The main observation that we want to make at this point is the fact that register machines with three registers out of which one (the output register) is non-decreasing are still universal, thus all the results from [9] are better by one protein without any major changes in their proofs. This is due to the fact that we only need two proteins to simulate the SUB instructions, and also the proof technique allows for such a modification. Subsequently, the following results were shown in [9]:

Theorem 3.

$$\begin{aligned}
NOP_1(pro_6; 3ffp) &= NRE. \quad \textit{(Theorem 1 in [9])}\\
NOP_1(pro_6; 2ffp, 4ffp) &= NRE. \quad \textit{(Theorem 2 in [9])}\\
NOP_1(pro_6; 2ffp, 5ffp) &= NRE. \quad \textit{(Corollary 3 in [9])}\\
NOP_1(pro_9; 1res, 2ffp) &= NRE. \quad \textit{(Theorem 4 in [9])}\\
NOP_1(pro_6; 1ffp, 2ffp) &= NRE. \quad \textit{(Theorem 6 in [9])}\\
NOP_1(pro_8; 1ffp, 2res) &= NRE. \quad \textit{(Theorem 7 in [9])}\\
NOP_1(pro_8; 2ffp, 3res) &= NRE. \quad \textit{(Theorem 9 in [9])}\\
NOP_1(pro_7; 1ffp, 3res) &= NRE. \quad \textit{(Theorem 10 in [9])}\\
NOP_1(pro_8; 3res, 4ffp) &= NRE. \quad \textit{(Theorem 11 in [9])}\\
NOP_1(pro_7; 2ffp, 5res) &= NRE. \quad \textit{(Theorem 13 in [9])}
\end{aligned}$$

We will proceed now to consider the same framework, but with the extra feature of the output based on time. We show that we can improve the result from Theorem 11 from [9]:

Theorem 4. *$NRE = NTOP_1(pro_7, 3res, 4ffp)$.*

Proof. We consider a register machine $M = (m, B, l_0, l_h, R)$ and we construct the system

$$\Pi = (O, P, [_1\ ]_1, \{l_0, b\}/P, E, R_1, C_{start}, C_{stop})$$

with the following components:

$$\begin{aligned}
O &= \{a_r, a'_r \mid 1 \leq r \leq 3\} \cup \{i, i', l_i, l'_i, l''_i, l'''_i, l^{iv}_i, L_i, L'_i \mid 0 \leq i \leq h\} \\
&\quad \cup \{o, o_1, o_2, b, h, \dagger\}. \\
E &= \{a_r, a'_r \mid 1 \leq r \leq 3\} \cup \{i \mid 0 \leq i \leq h\} \cup \ \{o\}. \\
P &= \{p_1, p_2, p_3, s_2, s_3, p, t\}. \\
C_{start} &= l''_h(O - \{l''_h, \dagger\})^*, \text{ in other words, } l''_h \\
&\qquad \text{appears exactly once and there are no copies of } \dagger \text{ in the membrane,} \\
&\qquad \text{and the rest of the symbols can} \\
&\qquad \text{appear in any multiplicity as they are ignored.} \\
C_{stop} &= (O - \{a_1\})^*, \text{ in this case } a_1 \text{ does not appear in the membrane.}
\end{aligned}$$

The proteins p and t are of the type 3res while all the others are of the type 4ffp. Proteins p and p_i are used in the simulation of ADD instructions of register i, proteins p, t and s_i are used in the simulation of SUB instructions of register i, and protein p, t, s_2 and s_3 are used in the simulation of the instructions for counting or termination.

The system has the following rules in R_1:

For an **ADD instruction** $l_1 : (\texttt{ADD}(r), l_2, l_3) \in R$, we consider the rules as shown in Table 1.

Step	Rules	Type	Environment	Membrane
1	$a'_r[_1 p_r \mid l_1 \to l_1[_1 p'_r \mid a'_r$	4ffp	El_1	ba'_r
2 or	$a_r[_1 p'_r \mid a'_r \to a'_r[_1 p_r \mid a_r$ **and** $l_1[_1 p \mid\to [_1 p \mid l_2$	4ffp, 3res	E	bl_2a_r
2	$a_r[_1 p'_r \mid a'_r \to a'_r[_1 p_r \mid a_r$ **and** $l_1[_1 p \mid\to [_1 p \mid l_3$	4ffp, 3res	E	bl_3a_r

Table 1. Steps for ADD instruction for Theorem 4.

We simulate the work of the ADD instruction in two steps. First we send out the current instruction label l_1 and bring in a copy of the (padding) symbol a'_r using the protein p_r. Next we simultaneously apply the rules to replace a'_r with a_r using the protein p'_r and we bring in the next instruction label l_2 or l_3 according to the currently simulated rule l_1. Of course, l_1 uniquely identifies which rule was simulated, thus there is no ambiguity about which symbols l_i are able to enter the membrane at this time. Let us now consider the case of the SUB instructions:

For a **SUB instruction** $l_1 : (\texttt{SUB}(r), l_2, l_3) \in R$ we consider the rules as shown in Table 2.

Step	Rules	Type	Environment	Membrane
1	$[_1 p \mid l_1 \to l_1'[_1 p \mid$	3res	El_1'	ba_r
2	$l_1'[_1 p \mid\to [_1 p \mid l_1''$	3res	E	$bl_1''a_r$
3	$o[_1 s_r \mid l_1'' \to l_1''[_1 s_r' \mid o$	4ffp	El_1''	boa_r
4	$l_1''[_1 p \mid\to [_1 p \mid l_1'''$ **and** $[_1 t \mid o \to o_1[_1 t\mid$	3res, 3res	Eo_1	$bl_1'''a_r$
5	$[_1 p \mid l_1''' \to l_1^{iv}[_1 p \mid$ **and** $o_1[_1 t \mid\to [_1 t \mid o_2$	3res, 3res	El_1^{iv}	bo_2a_r
	Register r is non-empty			
6	$l_1^{iv}[_1 s_r' \mid a_r \to a_r[_1 s_r \mid l_1^{iv}$ **and** $[_1 t \mid o_2 \to o[_1 t \mid$	4ffp, 3res	Eoa_r	bl_1^{iv}
7	$[_1 p \mid l_1^{iv} \to 2'[_1 p \mid$	3res	$E2'$	b
8	$2'[_1 p \mid\to [_1 p \mid l_2$	3res	E	bl_2
	Wrong Computation			
6	$l_1^{iv}[_1 t \mid\to [_1 t \mid L_3'$ **and** $[_1 p \mid o_2 \to \dagger[_1 p \mid$	3res, 3res	$E\dagger$	$bL'3a_r$
7	$\dagger[_1 t \mid\to [_1 t \mid \dagger$	3res	E	$b\dagger a_r$
	Register r is empty			
6	$[_1 t \mid o_2 \to o[_1 t \mid$	3res	Eo	b
7	$l_1^{iv}[_1 t \mid\to [_1 t \mid L_3'$	3res	E	bL_3'
8	$3[_1 s_r' \mid L_3' \to L_3'[_1 s_r \mid 3$	4ffp	EL_3'	$b3$
9	$[_1 p \mid 3 \to 3'[_1 p \mid$	3res	$E3'$	b
10	$3'[_1 p \mid]ra[_1 p \mid l_3$	3res	E	bl_3

Table 2. Steps for SUB instruction for Theorem 4.

We simulate the work of the SUB instruction in several steps (eight if the register is not empty and ten if it is empty). We first send out the current label as l_1' using the protein p. At the next step the symbol l_1' is brought in as l_1''. Next we exchange l_1'' and o using the protein s_r (the protein s_r is moved in its primed version of the flip-flop). We can now apply two rules in parallel and bring in l_1'' as l_1''' while sending out o as o_1. Next, l_1''' is sent out as l_1^{iv} while we bring in o_1 as o_2 in parallel.

In this moment our system will perform the checking of the contents of the register r. If the register is not empty, then l_1^{iv} will enter the membrane, decreasing the register and at the same time another marker o_2 is sent outside as o to help identify the correct case later. At the next stage l_1^{iv} will be sent out as $2'$ using protein p. Finally $2'$ will return as the next instruction label to be brought in (in this case l_2 as the register is not empty). If l_1^{iv} comes back in the membrane through the protein t instead of s_r', we will have a wrong computation. In this case we can send out o_2 as symbol $\dagger$ in parallel using the protein p (as this is the only channel available at this time to o_2, t being used by l_1^{iv}). Next we can bring in a copy of the symbol $\dagger$ into the membrane. The application of this rule will never satisfy the starting configuration; hence, we will not be able to use the time counter.

If the register is empty, after step 5 we have l_1^{iv} in the environment and o_2 in the membrane, and the protein associated with the subtract rule for the register r (s_r) is primed. At this moment l_1^{iv} cannot enter the membrane through the protein s_r' as there are no a_r objects in the membranes with which it must be exchanged.

There are two choices: either l_1^{iv} enters the membrane through t (and we get the wrong computation case as above) or t is used by o_2, and then l_1^{iv} sits one step in the environment. At the next step we have the "branching point": rather than exchanging with a_r (which will be present in the membrane in the case when the register is not empty), l_1^{iv} comes into the membrane as L_3' through t. Next we use the protein s_r' to exchange L_3' and 3, and then send out 3 as $3'$ using protein p. Now we bring in $3'$ as the next instruction to be simulated l_3.

Terminating/counting work: It is clear that at the end of the simulation, if the register machine has reached the final state, we will have the halting instruction symbol in the membrane along with one copy of the symbol b and multiple copies of the three different objects associated with their respective registers. At that time we will have the computed value encoded as the multiplicity of the object a_1 that is associated with the output register. We will also have in the system the label of the halting instruction, l_h; thus, the rule $([_1 p \mid l_h \rightarrow l_h'[_1 p \mid)$ can be applied only when the simulation is performed correctly. At the next step, using the protein s_2 we exchange l_h' and b.

The terminating/counting work is done by the rules as shown in Table 3.

Step	Rules	Type	Env	Membrane
1	$[_1 p \mid l_h \rightarrow l_h'[_1 p \mid$	3res	El_h'	$ba_1^{n_1}a_2^{n_2}a_3^{n_3}$
2	$l_h'[_1 s_2 \mid b \rightarrow b[_1 s_2' \mid l_h'$	4ffp	Eb	$l_h'a_1^{n_1}a_2^{n_2}a_3^{n_3}$
3	$b[_1 p \mid \rightarrow [_1 p \mid l_h''$ **and** $[_1 t \mid l_h' \rightarrow l_h''[_1 t \mid$	3res, 3res	El_h''	$l_h''a_1^{n_1}a_2^{n_2}a_3^{n_3}$
4	$h[_1 s_2 \mid l_h'' \rightarrow l_h''[_1 s_2' \mid h$ **or** $h[_1 s_2' \mid l_h'' \rightarrow l_h''[_1 s_2 \mid h$	4ffp	$El_h''a_1$	$l_h''a_1^{n_1}a_2^{n_2}a_3^{n_3}h$
	and $l_h''[_1 s_3 \mid a_1 \rightarrow a_1[_1 s_3' \mid l_h''$	4ffp		
	or $l_h''[_1 s_3' \mid a_1 \rightarrow a_1[_1 s_3 \mid l_h''$			

Table 3. Steps for terminating/counting instructions for Theorem 4.

Next we apply two rules in parallel and bring in b as l_h'' while sending out l_h' as l_h'', satisfying the C_{start} configuration. One can note that if there are no copies of a_1 in the membrane, then also the configuration C_{stop} is satisfied at the same time, thus our system would compute the value zero in that case. Next we exchange h from the environment with l_h'' and l_h'' from the environment with a_1 until we reach the stopping configuration. For any other value encoded in the multiplicity of a_1 it will take exactly the same number of steps to push the number of copies of object a_1 from the membrane. □

An interesting observation is the fact that the object b is used for the counting at the end of the computation. If one considers the same construct for membrane systems with proteins as defined in [14] (the "classical" systems with the output the multiplicity of objects in the membrane), then our construction is still valid even in the case of systems without time, thus we have the following theorem also proven:

Theorem 5. $NRE = NOP_1(pro_7, 3res, 4ffp)$.

The theorem above is valid as one can restrict the register machine to be simulated (without loss of generality) to the case when the machine halts with the non-output registers empty.

Thus it can be seen that we are able to improve the result shown in Theorem 10 in [9] both for systems based on multiplicity output and also for systems based on time. The next result improves significantly Theorem 11 from [9], in this case for systems based on time, and later one we will discuss also about the non-timed systems.

Theorem 6. *$NRE = NTOP_1(pro_3, 2ffp, 5res)$.*

Proof. We consider a register machine $M = (m, B, l_0, l_h, R)$ and we construct the system

$$\Pi = (O, P, [_1\ ]_1, \{l_0, b, e\}/P, E, R_1, C_{start}, C_{stop})$$

with the following components:

$$\begin{aligned}
O &= \{l_i, l_i' \mid 0 \leq i \leq h\} \cup \{a_1, a_2, a_3, b, o, y\}.\\
E &= \{a_1, a_2, a_3, o\}.\\
P &= \{p, q, s\}.\\
C_{start} &= (O - \{b\})^*, \text{ in other words, there are no copies of } b \text{ in the membrane,}\\
&\quad\ \text{and the rest of the symbols}\\
&\quad \text{can appear in any multiplicity as they are ignored.}\\
C_{stop} &= (O - \{a_1\})^*, \text{ in this case } a_1 \text{ does not appear in the membrane.}
\end{aligned}$$

Protein q is of type 5res while all the others are of the type 2ffp. Proteins p and q are used in the simulation of the ADD instruction, proteins q and s are used in the simulation of the SUB instruction, and protein q is used in the simulation of the instructions for counting or termination.

The system has the following rules in R_1:

For an **ADD instruction** $l_1 : (\texttt{ADD}(r), l_2, l_3) \in R$, we consider the rules as shown in Table 4.

Step	Rules	Type	Environment	Membrane
1	$a_r[_1 q \mid l_1 \rightarrow l_1'[_1 q \mid a_r$	5res	El_1'	bea_r
2	$l_1'[_1 q \mid e \rightarrow e[_1 q \mid l_2$	5res	Ee	bl_2a_r
2	$l_1'[_1 q \mid e \rightarrow e[_1 q \mid l_3$	5res	Ee	bl_3a_r
3	$e[_1 p \mid\rightarrow [_1 p' \mid e$ or $e[_1 p' \mid\rightarrow [_1 p \mid e$	2ffp, 2ffp	E	bea_r

Table 4. Steps for ADD instruction for Theorem 6.

We simulate the work of the ADD instruction in two steps. First we send out the current instruction label l_1 as l_1' and bring in a copy of the symbol a_r using the protein q. Next we apply the rule to send out e using the protein q and we bring

l'_1 in as the new instruction label. To simulate the non-deterministic behavior of these machines we have two rules that do the same job, the only difference being the next instruction label being brought back in the system. It is clear that the simulation of the ADD instruction is performed correctly. The work is finished in this case by the rule $(e[_1 p \mid\to [_1 p' \mid e)$ or $(e[_1 p' \mid\to [_1 p \mid e)$.

For a **SUB instruction** $l_1 : (\mathtt{SUB}(r), l_2, l_3) \in R$ we consider the rules as shown in Table 5.

Step	Rules	Type	Environment	Membrane
1	$[_1 s \mid l_1 \to l_1 [_1 s' \mid$	2ffp	El_1	bea_r
	Register r is non-empty			
2	$o[_1 s' \mid\to [_1 s \mid o$ **and** $l_1[_1 q \mid a_r \to a_r[_1 q \mid l'_1$	2ffp, 5res	Ea_r	$beol'_1$
3	$o[_1 q \mid l'_1 \to l'_1[_1 q \mid l_2$	5res	El'_1	$beol_2$
4	$l'_1[_1 q \mid o \to o[_1 q \mid y$	5res	Eo	bey
	Register r is empty			
2	$o[_1 s' \mid\to [_1 s \mid o$	2ffp	El_1	beo
3	$l_1[_1 q \mid o \to o[_1 q \mid l_3$	5res	Eo	bel_3

Table 5. Steps for SUB instruction for Theorem 6.

We simulate the work of the SUB instruction in several steps (four if the register is not empty and three if it is empty). At step 1 we first send out the current label l_1 using the protein s. If the register is not empty, at step 2, l_1 will enter the membrane, decreasing the register and at the same time the symbol o is brought in. At the next stage (step 3) l'_1 will be sent out using protein q, and o will return as the next instruction label to be brought in (in this case l_2 as the register is not empty). Finally l'_1 will return as the symbol y while sending out o, so that no extra copies of o are left in the membrane so that future SUB simulations will be performed correctly. The symbols y will accumulate in the membrane.

In the case when the register to be decremented is empty, we perform the same initial step, sending out the current label using the protein s. This time l_1 cannot enter the membrane at the step 2 as there is no a_r in the membrane to help bring it in. So l_1 will wait for one step in the environment. o is entering the membrane at step 2, so at the step 3 l_1 can now come into the membrane through q and is changed into the label of the next instruction to be simulated l_3.

The **terminating/counting work** stage is done by the rules as shown in Table 6.

Step	Rules	Type	Environment	Membrane
1	$o[_1 q \mid l_h \to l_h[_1 q \mid y$	5res	El_h	$bea_1^{n_1}a_2^{n_2}a_3^{n_3}$
2	$l_h[_1 q \mid b \to l'_h[_1 q \mid y$	5res	Ebl'_h	$ea_1^{n_1}a_2^{n_2}a_3^{n_3}$
3	$l'_h[_1 q \mid a_1 \to l'_h[_1 q \mid y$	5res	$El'_h a_1$	$ea_1^{n_1}a_2^{n_2}a_3^{n_3}$

Table 6. Steps for terminating/counting instructions for Theorem 6.

It is clear that at the end of the simulation, if the register machine has reached the final state, we will have the halting instruction symbol in the system membrane, along with one copy of the symbol b and multiple copies of the three different objects associated with the respective registers and the symbol y. At that time we will have the computed value encoded as the multiplicity of the object a_1 that is associated with the output register. We will also have in the system the label of the halting instruction, l_h, thus the rule $(o[_1 q \mid l_h \rightarrow l_h[_1 q \mid y)$ can be applied only when the simulation is performed correctly. At the next step, using the protein q we bring in l_h as y while sending out b as l'_h, satisfying the C_{start} configuration. One can note that if there are no copies of a_1 in the membrane, then also the configuration C_{stop} is satisfied at the same time, thus our system would compute the value zero in that case. Next we bring in l'_h as y while sending out a_1 as l'_h until we reach the stopping configuration. For any other value encoded in the multiplicity of a_1 it will take exactly the same number of steps to push the a_1-s out of the membrane. □

Thus it can be seen that by using time as the output, we are able to improve the result shown in Theorem 13 from [9], where seven proteins were required for universality, as opposed to the three used in the above proof.

If one wants to still restrict the discussion to only the case of the non-timed systems, with the price of one protein we can remove the objects y and e from the membrane (by first modifying them into some other symbols such as y' and o' and then expelling them to the environment). In this way it is easy to see that our proof for Theorem 6 leads to the following theorem:

Theorem 7. $NRE = NOP_1(pro_4, 2ffp, 5res)$.
Membrane systems with proteins on membranes are universal for one membrane and rules of the type 2ffp and 5res using only four proteins.

6 Final Remarks

We have shown that previous results about membrane systems with proteins on membranes can be improved in what concerns the number of proteins, we have also extended the model to have the output encoded as the time between two configurations and this has lead to a significant improvement as opposed to the previous results reported in [9]. Additional similar improvements are under investigation.

Acknowledgments

A. Păun gratefully acknowledges the support in part by LA BoR RSC grant LEQSF (2004-07)-RD-A-23 and NSF Grants IMR-0414903 and CCF-0523572. We acknowledge the significant improvements to the paper suggested by the anonymous referees.

References

1. B. Alberts, A. Johnson, J. Lewis, M. Raff, K. Roberts, P. Walter: *Molecular Biology of the Cell*, 4th ed. Garland Science, New York, 2002.
2. A. Alhazov, R. Freund, Yu. Rogozhin, Some Optimal Results on Symport/Antiport P Systems with Minimal Cooperation, M.A. Gutiérrez-Naranjo et al. (eds.), Cellular Computing (Complexity Aspects), ESF PESC Exploratory Workshop, Fénix Editora, Sevilla (2005), 23–36.
3. A. Alhazov, R. Freund, Yu. Rogozhin, Computational Power of Symport / Antiport: History, Advances and Open Problems, R. Freund et al. (eds.), Membrane Computing, International Workshop, WMC 2005, Vienna (2005), revised papers, *LNCS* 3850, Springer (2006), 1–30.
4. F. Bernardini, A. Păun, Universality of Minimal Symport/Antiport: Five Membranes Suffice, WMC03 revised papers in *LNCS* 2933, Springer (2004), 43–54.
5. L. Cardelli: Brane Calculi – Interactions of Biological Membranes. In *Computational Methods in Systems Biology. International Conference CMSB 2004, Paris, France, May 2004, Revised Selected Papers. LNCS*, 3082, Springer, Berlin, 2005, 257–280.
6. M. Cavaliere, R. Freund, Gh. Păun, Event–Related Outputs of Computations in P Systems, M.A. Gutiérrez-Naranjo et al. (eds.), Cellular Computing (Complexity Aspects), ESF PESC Exploratory Workshop, Fénix Editora, Sevilla (2005), 107–122.
7. R. Freund, A. Păun, Membrane Systems with Symport/Antiport: Universality Results, in *Membrane Computing. Intern. Workshop WMC-CdeA2002, Revised Papers* (Gh. Păun, G. Rozenberg, A. Salomaa, C. Zandron, eds.), *LNCS*, 2597, Springer, Berlin (2003), 270–287.
8. O.H. Ibarra, A. Păun, Counting Time in Computing with Cells, Proceedings of DNA11 conference, June 6-9, 2005, London Ontario, Canada, 112–128.
9. S.N. Krishna, Combining Brane Calculus and Membrane Computing, Proc. Bio-Inspired Computing – Theory and Applications Conf., BIC-TA 2006, Wuhan, China, September 2006, Membrane Computing Section and *Journal of Automata Languages and Combinatorics* in press.
10. M.L. Minsky, Recursive Unsolvability of Post's Problem of "Tag" and Other Topics in Theory of Turing Machines, *Annals of Mathematics*, 74 (1961), 437–455.
11. M.L. Minsky: *Computation: Finite and Infinite Machines.* Prentice Hall, Englewood Cliffs, New Jersey, 1967.
12. H. Nagda, A. Păun, A. Rodríguez-Patón, P Systems with Symport/Antiport and Time, Pre-proceedings of Membrane Computing, International Workshop, WMC7, Leiden, The Netherlands, 2006, 429–442
13. A. Păun, Gh. Păun, The Power of Communication: P Systems with Symport/Antiport, *New Generation Computing*, 20, 3 (2002) 295–306.
14. A. Păun, B. Popa, P Systems with Proteins on Membranes. *Fundamenta Informaticae* 72(4), (2006), 467–483.
15. A. Păun, B. Popa, P Systems with Proteins on Membranes and Membrane Division, Proc. 10th DLT Conf., Santa Barbara, USA, 2006, LNCS 4036, Springer, Berlin, 2006, 292–303.
16. Gh. Păun: *Membrane Computing – An Introduction.* Springer-Verlag, Berlin, 2002.
17. Gh. Păun, Further Twenty-six Open Problems in Membrane Computing, the Third Brainstorming Meeting on Membrane Computing, Sevilla, Spain, February 2005.
18. G. Rozenberg, A. Salomaa, eds., *Handbook of Formal Languages*, 3 volumes, Springer-Verlag, Berlin, 1997.
19. The P Systems Website: `http://psystems.disco.unimib.it`.