Encodings and Arithmetic Operations in Membrane Computing [*]

Artiom Alhazov, Cosmin Bonchiş, Cornel Izbaşa, Gabriel Ciobanu

June 19, 2006

1 Introduction

Membrane systems represent a new abstract model inspired by cell compartments and molecular membranes. Such a system is composed of various compartments, each compartment with a different task, and all of them working simultaneously to accomplish a more general task of the whole system. A detailed description of the membrane systems (also called P systems) can be found in [7]. A *membrane system* consists of a hierarchy of membranes that do not intersect, with a distinguishable membrane, called the *skin membrane*, surrounding them all. The membranes produce a delimitation between *regions*. For each membrane there is a unique associated region. Regions contain multisets of *objects*, *evolution rules* and possibly other membranes. Only rules in a region delimited by a membrane act on the objects in that region. The multisets of objects from a region correspond to the "chemicals swimming in the solution in the cell compartment", while the rules correspond to the "chemical reactions possible in the same compartment". Graphically, a membrane structure is represented by a Venn diagram in which two sets can be either disjoint, or one is a subset of the other. We refer mainly to the so-called *transition membrane systems*. Other variants and classes are introduced [7].

The membrane systems represent a new abstract machine. For each abstract machine, the theory of programming introduces and studies various paradigms of computation. For instance, Turing machines and register machines are mainly related to imperative programming, and λ-calculus is related to functional programming. Looking at the membrane systems from the point of view of programming theory, we intend to provide useful results for future definitions and implementations of P system-based programming languages, that is, programming languages that generate P systems as an executable form. The authors of such languages will certainly have to face the problem of number encoding using multisets, since the multiset is the support structure of P systems. We attempt to show that the problem of number encoding using multisets is an interesting

[*]Work partially supported by the CEEX Programme, project "ForMol" 47/2005

and complex one, with many possible approaches for various purposes. We outline a few of these approaches, and detail the most compact encoding using one membrane, also comparing it to the most compact encoding using strings. Such a comparison offers some hints about information encoding in general, specifically how do we most compactly encode information over structures that have underlying order (strings), or just multiplicity (multisets), or neither (sets).

Previous work related to number encodings using multisets was done in [3], where the encoding is done by allocating a membrane for each digit, and so "stringising" the multiset, constructing a string-like structure over it. Then this string-like structure is used to encode numbers in the classical manner. Our approach does not attempt to superimpose this string structure over the multiset, but tries to use only the already present quality of multiset elements, multiplicity, to encode numbers and, by extension, information. Thus, this approach is a more native one and might be easier to use in related biochemical experiments. Another advantage is that using this approach it is possible to represent arbitrarily large numbers without membrane creation, division or dissolution and without infinitely many membranes or object types. One disadvantage is that the arithmetic operations have slightly higher complexity. We have implemented the arithmetic operations, and each example is tested with our web-based simulator available at `http://psystems.ieat.ro/`.

2 Motivation

The development of P systems needs to be supported by efficient means of information encoding over multisets. As a first step, we consider the case of number encodings over multisets, similarly to how Church numerals are constructed in $\lambda-calculus$. Arguably, on any string-based abstract machine, numbers are most easily encoded by assigning each digit to a position in the string, that is, using positional number systems in an appropriate base, dependent on the alphabet of the machine. By contrast, on multiset-based machines (as P systems are), the positioning, while perfectly implementable, is not free, meaning we pay a certain price for it. So far, the most widely used number encoding over multisets is the natural encoding, meaning the natural number n is encoded using n objects of the same kind, e.g. by a^n. We envision these main types of number encodings over multisets (without string objects):

- *Positional encodings*
  These are obtained by superimposing a string structure over the multiset in various ways and representing the number using a positional number system.
  - The digits of the number in a certain base are naturally-encoded in successive membranes. The next digit is encoded in an inner (or outer) membrane, as in [3]. The shortcoming of this type of encodings lies in that, to represent arbitrarily large numbers, we need a **membrane structure with infinite depth**.

  - The digits of the number are encoded using different objects for each digit, e.g. if the number is $n = 3 \cdot 10^3 + 4 \cdot 10^2 + 2 \cdot 10 + 5$, we could represent it as $a^3b^4c^2d^5$. By convention, the a objects represent the first digit, the b objects the second one and so on. We are unaware of work in this area, however, this type of encodings are easily obtained from the previous type, by flattening the membrane structure in a natural way. As a consequence, whereas in the above type we need infinitely many membranes to represent arbitrary numbers, in this case we need **infinitely many object types**.

- *Non-positional encodings*
  In this case, the idea is not to superimpose a string structure over the multiset, so we do not have positioning. Instead, we rely only on the multiplicity of objects, a defining, native characteristic of the multiset. Below we consider several such number encodings. We also show that the natural encoding already mentioned is actually a first member in a family of such encodings. The main disadvantage of this approach is a greater encoding length, but this can be much improved over the natural encoding in other variants of this type. However, the encoding length and the complexity of arithmetic operations over numbers encoded in this manner, are generally greater than in positional encodings. The significant advantage of non-positional encodings is the fact that they use **a single membrane** and **finitely many object types**. This can be very desirable for practical reasons.

- *Hybrid encodings*
  Hybrid encodings derived from the above types can be considered, and may be practically useful, but are not investigated here.

It is important to note the fact that there is a qualitative difference between positional and non-positional encodings. The difference lies in that, in the case of positional encodings there is at least one additional *unbounded* characteristic of the membrane system with respect to the non-positional ones.

In the first positional encoding type, to represent arbitrarily large numbers, the *membrane structure has unbounded depth*, since each digit is assigned to a region, and the numbers to be represented have an unbounded number of digits.

In the second positional case, to represent arbitrarily large numbers, the P system alphabet has arbitrarily many elements, there is an *unbounded number of object types.*

By contrast, in the non-positional encodings, that rely solely on the multiplicity of multiset elements, the only unbounded aspect is the number of object instances, which is also unbounded in the case of positional encodings. So, using non-positional encodings, we can represent arbitrarily large numbers with a *finite* number of object types, given that the multiset elements have *unbounded* multiplicity. This is an important *qualitative difference* between positional and non-positional encodings, which may also be of great practical interest.

Using non-positional encodings, if we do not have enough room for object instances representing a number inside a membrane, there may be a way to obtain/create a larger membrane. Using positional encodings of the first kind, we would need additional membranes, or, for the second kind, additional object types, which probably map to additional substances corresponding to those types. These ideas can be understood easily by considering encodings of each kind:

Decimal	Binary	Pos. I	Pos. II	Non-pos.
0	0	$[\lambda]$	$[\lambda]$	$[\lambda]$
1	1	$[a^1]$	$[a^2]$	$[0]$
2	10	$[a^1[\lambda]]$	$[a^2b^1]$	$[1]$
3	11	$[a^1[a^1]]$	$[a^2b^2]$	$[0^2]$
4	100	$[a^1[\lambda[\lambda]]]$	$[a^2b^1c^1]$	$[0^1 1^1]$
...	...	...	...	...

We can see that in the last column the number of object types does not increase, but stays equal to 2, which is the chosen base. In the *Pos. I* column, the membrane structure gains depth as the number gains more digits. In the *Pos. II* column, the alphabet increases as the number gains more digits.

3 Combinatorics over multisets

To develop encoding and decoding algorithms for the above encodings we start with a short review of combinatorics over multisets based on [1].

Let M be a multiset.

Definition: An $r-$*permutation* of M is an ordered arrangement of r objects of M. If $|M| = n$ then an $n-$*permutation* of M is called a *permutation* of M.

Theorem 4.1 [1]. Let M be a multiset of k different types where each type has infinitely many elements. Then the number of $r-$*permutations* of M equals k^r.

Definition: An $r-$*combination* of M is an unordered collection of r objects from M. Thus an $r-$*combination* of M is itself an $r-sub-multiset$ of M. For a multiset $M = \{\infty a_1, \infty a_2, ..., \infty a_n\}$, an $r-$*combination* of M is also called an $r-$*combination* with repetition allowed of the n-set $S = \{a_1, a_2, ..., a_n\}$. The number of $r-$*combinations with repetition allowed* of an n-set is denoted by $\left\langle \begin{matrix} n \\ r \end{matrix} \right\rangle$.

Theorem 5.1 [1]. Let $M = \{\infty a_1, \infty a_2, ..., \infty a_n\}$ be a multiset of n types. Then the number of $r-$*combinations* of M is given by

$$\left\langle \begin{matrix} n \\ r \end{matrix} \right\rangle = \begin{pmatrix} n+r-1 \\ r \end{pmatrix} = \begin{pmatrix} n+r-1 \\ n-1 \end{pmatrix}$$

4 Number encodings over multisets

We consider two types of encodings with useful features: the most compact encoding (MCE) and the Gray style most compact encoding ($GSMCE$).
The natural encoding, which represents a number n with n objects, is actually MCE_1.

Gray style encodings are based on the fact that a minimal change is performed on the encoded number to obtain its successor or predecessor. The change means either replacing an existing symbol or adding a new symbol of the same type as the existing ones.

4.1 Most compact encoding using one membrane (MCE).

The natural encoding is easy to understand and work with, but it has the disadvantage that for very large numbers the P system membranes will contain a very large number of objects, which is undesirable for practical reasons. First we analyze the most compact encoding using two object types (binary case) and then briefly the ternary case.

Decimal	MCE_1 $Natural$ $encoding$	MCE_2	MCE_3	$GSMCE_2$
0	λ	λ	λ	λ
1	a^1	$0^1 1^0$	$0^1 1^0 2^0$	$0^1 1^0$
2	a^2	$0^0 1^1$	$0^0 1^1 2^0$	$0^0 1^1$
3	a^3	$0^2 1^0$	$0^0 1^0 2^1$	$0^0 1^2$
4	a^4	$0^1 1^1$	$0^2 1^0 2^0$	$0^1 1^1$
5	a^5	$0^0 1^2$	$0^1 1^1 2^0$	$0^2 1^0$
6	a^6	$0^3 1^0$	$0^1 1^0 2^1$	$0^3 1^0$
7	a^7	$0^2 1^1$	$0^0 1^2 2^0$	$0^2 1^1$
8	a^8	$0^1 1^2$	$0^0 1^1 2^1$	$0^1 1^2$
9	a^9	$0^0 1^3$	$0^0 1^0 2^2$	$0^0 1^3$
10	a^{10}	$0^4 1^0$	$0^3 1^0 2^0$	$0^0 1^4$
11	a^{11}	$0^3 1^1$	$0^2 1^1 2^0$	$0^3 1^1$
12	a^{12}	$0^2 1^2$	$0^2 1^0 2^1$	$0^2 1^2$
13	a^{13}	$0^1 1^3$	$0^1 1^2 2^0$	$0^1 1^3$

Table 1: Number encodings

We denote by $N(b, m)$ the number of numbers encoded in base b with m objects. Here are some values for the function $N(b, m)$:

b/m	0	1	2	3	4	5	6	7	8	9
2	1	2	3	4	5	6	7	8	9	10
3	1	3	6	10	15	21	28	36	45	55
4	1	4	10	20	35	56	84	120	165	220
5	1	5	15	35	70	126	210	330	495	715
6	1	6	21	56	126	252	462	792	1287	2002

Table 2: Number of numbers encoded in base b with m objects

For our representations, the definition of $r - combinations$ is useful in determining the number of objects represented with m objects in a multiset with b types. b indicates our **base**.

$$N(b,m) = \left\langle \begin{array}{c} b \\ m \end{array} \right\rangle = \left(\begin{array}{c} b-1+m \\ m \end{array} \right) = \left(\begin{array}{c} b-1+m \\ b-1 \end{array} \right) \tag{1}$$

which is also the number of $m - combinations$ of a multiset of b types.

Based on the Pascal formula: $\left(\begin{array}{c} n \\ r \end{array} \right) = \left(\begin{array}{c} n-1 \\ r \end{array} \right) + \left(\begin{array}{c} n-1 \\ r-1 \end{array} \right)$ and its extended form:

$$\left(\begin{array}{c} n \\ r \end{array} \right) = \sum_{i=0}^{r} \left(\begin{array}{c} n-1-i \\ r-i \end{array} \right) \tag{2}$$

we replace in (2) $b-1+m$ for n and m for r,

$$N(b,m) = \left\langle \begin{array}{c} b \\ m \end{array} \right\rangle = \left(\begin{array}{c} b-1+m \\ m \end{array} \right) = \sum_{i=0}^{m} \left(\begin{array}{c} (b-1)-1+(m-i) \\ (m-i) \end{array} \right) = \sum_{i=0}^{m} \left\langle \begin{array}{c} b-1 \\ m-i \end{array} \right\rangle = \sum_{i=0}^{m} \left\langle \begin{array}{c} b-1 \\ i \end{array} \right\rangle = \sum_{i=0}^{m} N(b-1,i)$$

and obtain that

$$N(b,m) = \sum_{i=0}^{m} N(b-1,i) \tag{3}$$

To develop encoding and decoding algorithms for a certain base b we have to solve the equation:

$$\sum_{i=0}^{m-1} N(b,i) - n = 0$$

Since from (3) we have

$$\sum_{i=0}^{m-1} N(b,i) = N(b+1,m-1) = \left\langle \begin{array}{c} b+1 \\ m-1 \end{array} \right\rangle = \left(\begin{array}{c} b+m-1 \\ m-1 \end{array} \right) = \frac{(b+m-1)!}{b!(m-1)!} = \frac{\prod_{i=0}^{b-1}(m+i)}{b!}$$

then we obtain that

$$\sum_{i=0}^{m-1} N(b,i) - n = 0 \Leftrightarrow \frac{\prod_{i=0}^{b-1}(m+i)}{b!} - n = 0. \tag{4}$$

The integer part of its greatest real positive root of (4) represents m, i.e. the number of objects needed to represent the natural number n.

We also note that

$$\frac{\prod_{i=0}^{b-1}(m+i)}{b!} = \sum_{i=1}^{b} \left[\begin{array}{c} b \\ i \end{array} \right] m^i \tag{5}$$

where $\left[\begin{array}{c} b \\ i \end{array} \right]$ are the Stirling numbers of the first kind b cycle i.

Equation (5) generates some notable number sequences:

b	Sequence name
2	triangular numbers (2-simplex)
3	tetrahedral numbers (3-simplex)
4	pentatopal numbers (4-simplex)
k	k-simplex numbers

The integer part of its greatest real positive root of Equation (4) will represent m, i.e. the number of objects needed to represent the natural number n.

We also note that $n = O(m^b)$, where b is the base, as opposed to the most compact encoding on a string, where $n = O(b^m)$.

4.1.1 Most compact binary encoding or Cantor encoding (MCE_2 or CE_2)

We call the binary case of the most compact encoding using one membrane the **Cantor encoding**, because it is the inverse of the Cantor pairing function. The Cantor pairing function is the bijection π:$\mathbb{N}$x$\mathbb{N} \to \mathbb{N}$ defined by:

$$\pi(k_1, k_2) = \frac{1}{2}(k_1 + k_2)(k_1 + k_2 + 1) + k_2$$

It is a variant of the Hopcroft-Ullman function, and together they are the only quadratic functions with real coefficients that are bijections from $\mathbb{N}$x$\mathbb{N}$ to $\mathbb{N}$. They were introduced naturally by Cantor in the proof of $|\mathbb{Q}| = |\mathbb{N}| = \varkappa_0$. For more details see [4]. Here we present the original introduction of this encoding, though knowing the encoding is the inverse of the Cantor pairing function a much easier formulation would be possible.

To minimize the number of objects (object instances) we encode natural numbers in the way depicted in Table 1, the MCE_2 column. We use two object types to illustrate this encoding, thus obtaining a binary encoding over multisets (unordered binary encoding). We derive the encoding and decoding procedures as follows:

To **encode** the natural number n, we first have to determine the number m of objects needed to represent it, and then the object types. The length of the representation is the size of the multiset containing the number. We represent the number 0 as λ.

Decimal	MCE_2
0	λ
1	0
2	1
3	00
4	01
5	11
6	000
7	001
8	011
9	111
10	0000
11	0001
12	0011

Number of numbers	Number of objects
1	0
2	1
3	2
4	3
...	...
$m+1$	m

Table 3: The binary encoding

In the binary encoding we notice that we can represent $m+1$ numbers with m objects, for $m > 0$. Thus, the number n represented with m objects will have before it at least $\sum_{i=1}^{m} i$ numbers. So m is the greatest natural number that verifies: $\sum_{i=1}^{m} i = \frac{m(m+1)}{2} \leq n$. The sequence $\frac{m(m+1)}{2} = C_{m+1}^{2}$ represents the triangular numbers. To find m we thus need to solve this equation: $\frac{x(x+1)}{2} - n = 0$.The roots are $x_{1,2} = \frac{-1 \pm \sqrt{8n+1}}{2}$. The greatest (and only positive) root is $x_1 = \frac{-1+\sqrt{8n+1}}{2}$,and $m = [x_1] = [\frac{-1+\sqrt{8n+1}}{2}]$.

To determine the types of these m objects, we notice that the first number encoded with m objects will have all objects of type 0. The position of n between the numbers represented with m objects is given by the difference $n - \frac{m(m+1)}{2}$.So there will be $k = n - \frac{m(m+1)}{2}$ objects of type 1, the rest being 0. We **decode** the number encoded using m objects with k objects of type 1 as

$$n = \frac{m(m+1)}{2} + k.$$

5 P systems for most compact encodings

We present the P systems that implement the arithmetic operations on numbers encoded using the unary and binary case of the most compact encoding. All P systems presented below have corresponding XML input files [8] we use with WebPs, a Web-based P system simulator [5]. The notations we use are standard in membrane computing [7], so that we do not recall any definition.

5.1 Natural encoding - MCE_1

Addition
Time complexity: $O(1)$

Addition is trivial; we consider n objects a and m objects b, and the rule $a \rightarrow b$ in a system with only one membrane, starting with the initial configuration $[_0 a^n b^m]_0$. The rule is applied in parallel as many times as possible and thus all objects b are replaced with a. The remaining number of objects a represents the addition $n + m$.

Subtraction
Time complexity: $O(1)$

The difference of two numbers is described in the following way: given n objects a and m objects b, a rule $ab \rightarrow \lambda$ says that one object a and one object b are deleted. Consequently, all the pairs ab are erased. The remaining number of objects represents the difference between n and m.

$$\begin{array}{rcl}
\Pi_1 & = & (V, \mu, w_0, (R_0, \emptyset), 0), \\
V & = & \{a, b\}, \\
\mu & = & [_0\ ]_0, \\
w_0 & = & a^n b^m, \\
R_0 & = & \{ab \rightarrow \lambda\}.
\end{array}
\qquad
\begin{array}{rcl}
\Pi_2 & = & (V, \mu, w_0, (R_0, \rho_0), 0), \\
V & = & \{a, b, s, u, +, -\}, \\
\mu & = & [_0\ ]_0, \\
w_0 & = & a^n b^m u, \\
R_0 & = & \{r_1 : ab \rightarrow \lambda, \\
 & & r_2 : au \rightarrow +s,\ r_3 : bu \rightarrow -s, \\
 & & r_4 : a \rightarrow s,\ r_5 : b \rightarrow s\}. \\
\rho_0 & = & \{r_1 > r_2, r_1 > r_3, \\
 & & r_2 > r_4, r_3 > r_5\}.
\end{array}$$

Subtraction is performed by system Π_2, which produces the difference as well the sign of the result, encoded as a $+$ or $-$ object.

Multiplication

We developed P systems with and without promoters for multiplication. Figure 5.1 presents a P system Π_1 without promoters (but with priorities) for multiplication of n (objects a) by m (objects b), the result being the number of objects d in membrane 0. Initially only the rule $au \rightarrow v$ can be applied, generating an object v which activates the rule $bv \rightarrow dev$ m times, and then $av \rightarrow u$. Now $eu \rightarrow dbu$ is applied m times, followed by $au \rightarrow v$. The procedure is repeated until no object a is present within the membrane. We note that each time when one object a is consumed, then m objects d are generated.

Figure 1: Π_1 multiplier without promoters (natural encoding)

Figure 5.1 presents a P system Π_2 with promoters for multiplication of n (objects a) by m (objects b), the result being the number of objects d in membrane 0. The object a is a promoter in the rule $b \rightarrow bd|_a$, i.e., this rule can only be applied in the presence of object a. The available m objects b are used in order to apply m times the rule $b \rightarrow bd|_a$ in parallel; based on the availability of a objects the rule $au \rightarrow u$ is applied in the same time. The procedure is repeated until no object a is present within the membrane. We note that each time when one object a is consumed, then m objects d are generated.

Figure 2: Π_2 multiplier with promoters (natural encoding)

The important aspects related to the complexity of both multipliers are presented in the following table

	Type of objects	No of rules	No of priority levels	Time complexity
Π_1	6	4	2	$O(n \cdot m)$
Π_2	4	2	2	$O(n)$

Table 4: Minimal P systems for multiplication

As a particular case, it would be quite easy to compute n^2 by just placing the same number n of objects a and b. An interesting feature of the previous system is that the computation may continue after reaching a certain result, and so the system acts as a P transducer [6].

Thus if initially there are n (objects a) and m (objects b), the system evolves and produces $n \cdot m$ objects d. Afterward, the user can inject more objects a and the system continues the computation obtaining the same result as if the objects a are present from the beginning. For example, if the user wishes to compute $(n + k) \cdot m$, it is enough to inject k objects a at any point of the computation. Therefore this example emphasizes the asynchronous feature and a certain degree of reusability and robustness.

Division – Figure 3

We implement division as repeated subtraction – see the system in Figure 3. We compute the quotient and the remainder of n_2 (objects a in membrane 1) divided by n_1 (objects a in membrane 0) in the same P system evolution. The evolution starts in the outer membrane by applying the rule $a \rightarrow b(v, in_1)$. Therefore the rule $a \rightarrow b(v, in_1)$ is applied $n_1$times converting the objects a into objects b, and object v is injected in the inner membrane 1. The evolution continues with a subtraction step in the inner membrane, with the rule $av \rightarrow e$ applied n_1 times whenever possible. Two cases are distinguished in the inner membrane:

Figure 3: P system for division (natural encoding)

- If there are more objects a than objects v, only the rules $es \rightarrow s(r, out)(q, out)$ and $e \rightarrow (r, out)$ are applicable. Rule $es \rightarrow s(r, out)(q, out)$ sends out to membrane 0 a single q (restricted by the existence of a single s into this membrane) for each subtraction step. The number of objects q represents the quotient. On the other hand, both rules send out n_1 objects r (equal to the number of objects e). The evolution continues in the outer membrane by applying $br \rightarrow b'|_{\neg v}$ of $n_1$times, meaning the objects b are converted into objects b' by consuming the objects r only in the absence of v. Then the rule $b' \rightarrow a$ produces the necessary objects a to repeat the entire procedure.

- When there are less objects a than objects v in the inner membrane we get a division remainder. After applying the rule $av \rightarrow e$, the remaining objects v activate the rule $v \rightarrow (v, out)$. Therefore all these objects v are sent out to the parent membrane 0, and the rules $es \rightarrow s(r, out)(q, out)$ and $e \rightarrow (r, out)$ are applied. Due to the fact that we have objects v in membrane 0, the rule $br \rightarrow b'|_{\neg v}$ cannot be applied. Since n_2 is not divisible by n_1, the number of the left objects r in membrane 0 represents the remainder of the division. A final cleanup is required in this case, because an object q is sent out even if we have not a "complete" subtraction step; the rule $qsr \rightarrow r|_v$ removes that extra q from membrane 0 in the presence of v. This rule is applied only once because we have a unique s in this membrane too.

5.2 Successor, predecessor, adder and multiplier P systems for MCE_2

Successor MCE_2 – **Figure 4**
Time complexity: $O(1)$

Complexity proof: by the evolution we can observe that:
– **If** an object 0 appears in the encoding, then the rule $0s \rightarrow 1$ transforms a single 0 into an 1; **1 time unit**
– **else**, if the number is encoded using only objects 1, the rule $1s \rightarrow 00t_0$ transforms one object 1 into two 0's and produce a promoter t_0, **1 time unit**. The rule $1 \rightarrow 0|_{t_0}$ promoted by t_0, transforms all others objects 1's into 0's, **1 time unit** (because of the maximally-parallel rewriting – *MPR*).

Consequently, the time complexity of the successor is $O(1)$ because the evolution ends in 1 or 2 time units.
P system evolution
The successor of a number in this encoding is computed in the following manner: either we have an object 0 and the rule $0s \rightarrow 1$ transforms one 0 into an 1, or we have a number encoded using only objects 1 and the rule $1s \rightarrow 00t_0$ transforms one object 1 in two objects 0 (increasing the length of encoding) and generate an object t_0 which promotes the rule $1 \rightarrow 0|_{t_0}$. This rule transforms all other objects 1s into 0s.

Figure 4: Successor in MCE_2

Predecessor MCE_2 – **Figure 5**
Time complexity: $O(1)$

Complexity proof: by the evolution we can observe that:
- **If** an object 0 appears in the encoding then the rule $1s \rightarrow 0$ transforms a single 1 into an 0; **1 time unit**
- **else**, if the number is encoded using only objects 0, the rule $0s \rightarrow t_0$ erase

an objects 0, **1 time unit** and generate an object t_0 which promotes the rule $0 \to 1|_{t_0}$. This rule transforms all the other objects 0s into 1s, **1 time unit** (because of the MPR).
Thus, the time complexity of the predecessor is $O(1)$ because the evolution ends in 1 or 2 time units.
P system evolution
The predecessor of a number is computed by turning an 1 into a 0 by the rule $1s \to 0$ whenever we have objects 1; otherwise we consume one 0 and produce an object t_0 by the rule $0s \to t_0$, and transform all the other objects 0 into 1 by rule $0 \to 1|_{t_0}$ promoted by t_0.

Figure 5: Predecessor in MCE_2

Addition in MCE_2 – Figure 6
Time complexity: $O(n)$

Complexity proof: Considering that we have the number n encoded in predecessor and the number m encoded in successor. Because the evolution of the addition is means incrementing a number while decrementing the other until we cannot decrement anymore, we can count n decrements (n predecessor evolutions, each $O(1)$ time complexity) and the same number of increments (n successor evolutions, each with time complexity $O(1)$). Consequently, addition ends in $2n$ steps, with time complexity $O(n)$.
P system evolution
We implement addition by coupling the predecessor and successor through a "communication token". We use the general idea that we add two natural numbers by incrementing a number while decrementing the other until we cannot decrement anymore.

Figure 6: Addition in MCE_2

The evolution is started by the predecessor computation in the outer membrane which injects a communication token s into the inner membrane. For each predecessor cycle (except the first one) the inner membrane computes the successor passing back the token s. Since we want to stop the computation when the predecessor is reaching 0, we omit computing the successor for one predecessor cycle: the first token s is eaten-up by the single object p present in the inner membrane.

Multiplication MCE_2 – Figure 7
Time complexity: $O(n1 \cdot n2)$

Complexity proof: Considering that we have the number $n1$ encoded in predecessor and the number $n2$ encoded in adder. The multiplication evolves by performing $n1$ times the addition of $n2$ with the result memorized in the output membrane (this result start by 0). A predecessor ($O(1)$) decrease $n1$ until reaches 0 and for each decreasing an addition ($O(n2)$) is performed. Consequently, the multiplication ends in $n1 \cdot O(1) \cdot O(n2) = O(n1 \cdot n2)$ time complexity. If $n1 = n2 = n$, $O(n1 \cdot n2) = O(n^2)$.

Figure 7: Multiplier in MCE_2

P system evolution
We implement multiplication in a similar manner to addition, coupling a **predecessor** with an **adder**. The idea is to provide the first number to a predecessor, and perform the addition iteratively until the predecessor reaches 0. The predecessor is computed in membrane 0, and in membranes 1, bk, and 2, we have a modified adder. The evolution is started by the predecessor working over the first number, in the outer membrane 0. The predecessor activates the adder by passing a communication token w. The adder is modified to use an extra backup membrane which always contains the second number, which we named bk (to suggest that it contains a backup of the second number). When the adder is triggered by the predecessor, it signals the backup membrane bk which supplies a fresh copy of the second number to the adder (bk fills membrane 1 with the encoding of the second number) and a new addition iteration is performed. At the end of the iteration, the adder sends out a token s to the predecessor in membrane 0. The procedure is repeated until the predecessor reaches 0.

5.3 Multiple-iterations successor and predecessor

Multiple-iterations successor MCE_2 – **Figure 8**
Time complexity: $O(p/m) = O(p/\sqrt{n})$

Complexity proof: by the P system evolution we observe:
– the rule $0s \rightarrow 1$ consumes as many s as possible (maximum m[= the length of n]); **1 time unit** (because of MPR)
– the rule $su \rightarrow 0t$ generates an single 0 (the length of n is increasing by 1 [$m = m + 1$]); **1 time unit**
– the rule $1 \rightarrow 0|_t$ transforms all (m) objects 1 into 0; **1 time unit**
for time complexity the rule $t \rightarrow u$ is not important because it can be applied in the same time with the previous one.

In the first 3 time units $m+1$ objects s are consumed, in the next 3 time units $m + 2$ objects s are consumed, and so on (m is increasing), until all the objects s are consumed. We **compute all p iterations in $3k$ time units** (where k is from $p = \sum_{i=1}^{k}(m + i)$), meaning the time complexity is $O(3k)$. If we consider that the codification length doesn't grow for each 3 time units, is the same like for the first 3 unit times $m + 1$, it is obtaining $p = \sum_{i=1}^{k}(m + 1) = k(m + 1)$; further $k = \frac{p}{m+1}$. Consequently, the time complexity is $O(3k) = O(\frac{3p}{m+1}) = O(\frac{p}{m})$. We obtained $O(p/m)$ to compute p successor iterations with *Multiple-iteration successor*, better than simple *successor* which needs $p \cdot O(1) = O(p)$ to compute p successor iterations.

Figure 8: Multiple-iterations successor

P system evolution
The multiple-iterations successor performs p successor iterations on the number n. The number p of iterations is the number of s objects. In this encoding the multiple-iterations successor is computed in the following manner. Considering the order of priority, the rule $0s \to 1$ is applied; it consumes as many s as possible and objects 0 are transformed into objects 1. Then if objects s still exist, the rule $su \to 0t$ generates a single 0, and generates a t which promotes the rule $1 \to 0|_t$, transforming all objects 1 into objects 0. Together with one 0 generated by the $su \to 0t$ rule, the number of objects in the encoding is increased. The last rule $t \to u$ converts the object t into an u which allows the second rule to consume a single s. If the objects s are not entirely consumed, then this process is repeated.

Multiple-iterations predecessor MCE_2 – **Figure 9**
Time complexity: $O(m) = O(\left\lfloor \frac{-1+\sqrt{8n+1}}{2} \right\rfloor) = O(\sqrt{n})$

Complexity proof: by the P system evolution we observe:
– the rule $1s \to 0$ consumes as many s as possible (maximum m[= the length of n]); **1 time unit** (because of MPR)
– the rule $0su \to t$ erases an single 0 (the length of n is decreasing by 1 [$m = m - 1$]); **1 time unit**
– the rule $0 \to 1|_s$ transforms all (m) objects 0 into 1; **1 time unit**
for time complexity the rule $t \to u$ is not important because it can be applied in the same time unit with the first one ($1s \to 0$).

In the first 3 time units $m + 1$ objects s are consumed, in the next 3 time units m objects s are consumed, and so on (m is decreasing), until all the objects s are consumed. We **compute all p iterations in $3k$ time units** (where k is from $p = \sum_{i=1}^{k}(m - i)$), meaning the time complexity is $O(3k)$. If we consider,in the worst case, that m is decreasing until it reaches 0 (decoder), it is obtaining $k = m$ and $p = \sum_{i=0}^{m}(m - i)$. Consequently, the time complexity is $O(3k) = O(3m) = O(m)$. We obtained $O(m)$ to compute p predecessor iterations with *Multiple-iteration* predecessor, better than simple *predecessor* which needs $p \times O(1) = O(p)$‘ to compute p predecessor iterations.

Figure 9: Multiple-iterations predecessor

P system evolution
The multiple-iterations predecessor performs p predecessor iterations on the number n. The number of iterations is the number of objects s. The multiple-iterations predecessor is computed in the following manner. Considering the order of priority, the rule $1s \to 0$ is applied, consuming as many s as possible, and objects 1 are transformed into objects 0. If we still have objects s, the rule $0su \to t$ removes a single 0 and generate one t which promotes the rule $0 \to 1|_t$ which transforms all 0's into 1's. The number of objects in the encoding is

decreased by the rule $0su \rightarrow t$. The last rule $t \rightarrow u$ converts the object t into an u which allows the second rule to consume a single s. If the objects s are not entirely consumed, then this process is repeated.

Decoder MCE_2 **– Figure 10**
Time complexity: $O(m) = O(\sqrt{n})$

Complexity proof: Time complexity of Decoder MCE_2 is the same as for the Multiple-iteration predecessor MCE_2
P system evolution
The decoder is an multiple-iterations predecessor that performs n predecessor iterations on the number n (the encoded number). Instead of consuming s objects it produces d objects. The number of d objects is n when the system stops.

Figure 10: Decoder MCE_2

Optimized adder MCE_2 **– Figure 11**
Time complexity: $O(\sqrt{n})$

Complexity proof: by the P system evolution we observe that the optimized adder contains a multi-iteration predecessor in one membrane and a multi-iterations successor in the other. Because the successor performs its iterations in an asynchronous manner without any response to the predecessor the *time complexity* is given by the worst time complexity between multi-iteration predecessor ($O(p/m)$) and multi-iterations successor ($O(m) = O(\sqrt{n})$). The worst case is when p is of order n and it is obtained $O(\frac{p}{m}) = O(\frac{n}{\sqrt{n}}) = O(\sqrt{n})$. Consequently, the time complexity of the optimized adder is $O(\sqrt{n})$.

Figure 11: Optimized adder

P system evolution
The optimized adder contains in membrane 0 a multiple-iteration predecessor (Decoder), and in membrane 1 a multiple-iterations successor. Each membrane contains a term of the addition. As opposed to the simple adder where the predecessor and the successor perform a synchronization after each iteration, in this optimized adder the predecessor compute in one step multiple iterations, and sends multiple objects s to the successor. The successor performs its iterations in an asynchronous manner (without any response to the predecessor). The evolution stops when the predecessor stops.

5.4 Successor P systems for MCE_3 – Figure 12 Time complexity: $O(1)$

P system evolution
The successor of a number in this encoding is computed in the following manner, we have 4 possibilities:

- either, we have an object 1 and the rule $1s \to 2$ transforms one 1 into one 2,
- or, we have a number encoded using objects 0 and 2 then the rule $0s \to 1t_1|_2$ transforms one 0 into 1 and generate the object t_1 that promotes the rule $2 \to 1|_{t_1}$ which transforms all other 2's into 1's
- or, the encoding contains only objects 2 then the rule $2s \to 00t_0$ transforms one object 2 into two objects 0 and a t_0 which promotes the rule $2 \to 0|_{t_0}$. This rule transforms all other 2's into 0's. In this case the length of the encoding is increased.
- or, the encoding contains only objects 0 and then the rule $0s \to 1$ transforms one object 0 into one object 1.

Figure 12: Optimized adder

6 Conclusion

The most compact encodings over multisets represent n in $O(m^b)$ where b is the base of the encoding, and m is the codification length. When we consider strings instead of multisets, and the position becomes a relevant information, then the most compact encoding of n is of order $O(b^m)$. This fact provides some hints about information encoding in general, allowing to compare the most compactly encoded information over structures as simple sets, multisets, and strings of elements from a multiset (where position is relevant).

	Singularity	Multiplicity	Position
Media	set	multiset	string
Encoding length	constant	$\sqrt[b]{n}$	$log_b n$
$number(base, length)$	-	$n = O(m^b)$	$n = O(b^m)$

A primary conclusion is that the effect of considering position as relevant over the elements of a multiset is the reduction of the encoding length from $\sqrt[b]{n}$ to $log_b n$. On the other hand, the encodings over multisets are much closer to the computational models inspired by biology, and can help to improve their computation power.

References

[1] B. Chen. Permutations and Combinations. Electronic materials on *Combinations on Multisets*, Hong Kong University of Science and Technology, 2005.

[2] R. L. Graham, D.E. Knuth, O. Patashnik. Concrete Mathematics. *Addison-Wesley, Reading, MA*, 1990

[3] A. Atanasiu. Arithmetic with membranes, *Pre-proceedings of the Workshop on Multiset Processing (Curtea de Argeş, August 21-25, 2000)*, pages 1 - 17.

[4] E.W. Weisstein et al. "Pairing Function." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/PairingFunction.html

[5] C. Bonchiş, G.Ciobanu, C. Izbaşa, D. Petcu. A Web-based P systems simulator and its parallelization. In C. Calude et al. (Eds.): *Unconventional Computing*, LNCS vol.3699, Springer, 58-69, 2005.

[6] G. Ciobanu, Gh. Păun, Gh. Ştefănescu. *P Transducers*, *New Generation Computing* vol.24, 1-28, 2006.

[7] Gh. Păun. *Membrane Computing. An Introduction.* Springer, 2002.

[8] A. Alhazov, C. Bonchiş, G. Ciobanu, C. Izbaşa. *Encodings and Arithmetic Operations in Membrane Computing. Simulator examples appendix*, `http://psystems. ieat.ro/xml.pdf`.