Encodings and Arithmetic Operations in P Systems*

Artiom Alhazov[1], Cosmin Bonchiş[2], Gabriel Ciobanu[2,3], and Cornel Izbaşa[2]

[1] Research Group on Mathematical Linguistics, Rovira i Virgili University and Institute of Mathematics and Computer Science, Academy of Sciences of Moldova, `artiome.alhazov@estudiants.urv.es`

[2] Research Institute "e-Austria" Timişoara, Romania
`cbonchis@info.uvt.ro`, `cornel@ieat.ro,cizbasa@info.uvt.ro`

[3] Romanian Academy, Institute of Computer Science
Blvd. Carol I nr.8, 700505 Iaşi, Romania
`gabriel@iit.tuiasi.ro`

Summary. Following [2], we present in this paper various number encodings and operations over multisets. We obtain the most compact encoding and several other interesting encodings and study their properties using elements of combinatorics over multisets. We also construct P systems that implement their associated operations. We quantify the effect of adding *order* to a multiset thus obtaining a string, as going from encoding lengths of the number n in base b and time complexities of operations of the order $\sqrt[b]{n}$ to lengths and complexities of order $log_b n$.

1 Introduction

Membrane systems represent a new abstract model inspired by cell compartments and molecular membranes. Essentially, 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 [9]. 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

* Work partially supported by the CEEX grant 47/2005

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 [9].

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. The idea to encode numbers using multisets and to define arithmetical operations in membrane computing is presented in [2]. Here we present more details, emphasizing other interesting aspects related to the number encoding using multisets and P systems. We outline various approaches of the most compact encodings using one membrane, also comparing it to the most compact encoding using strings. This comparison is meant to offer 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), that is how do we encode information using strings, multisets or sets.

Prior work related to number encodings using multisets was done in [1], where the encoding is done by allocating a membrane for each digit, thus "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 numbers with arbitrary number of digits, without membrane creation, division or dissolution. One disadvantage is that the arithmetic operations have slightly higher complexity. We have designed and implemented sequential and parallel software simulators [5, 6]; a web-based implementation is presented in [3]. We have implemented the arithmetic operations, and each example is tested with our web-based simulator available at `http://psystems.ieat.ro/`.

2 Combinatorics Over Multisets (Review)

To develop encoding and decoding algorithms for the above encodings we start with a short review of combinatorics over multisets based on [4].

Let M be a multiset.

2.1 Permutations over multisets

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 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.

2.2 Combinations over multisets

Definition: An $r-combination$ of M is an unordered collection of r objects from M. Thus an $r-combination$ of M is itself an $r-submultiset$ 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 2. 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

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

3 Number Encodings Over Multisets

We consider several encodings with useful features such as the most compact encoding (MCE), Gray-style most compact encoding ($GSMCE$), square root encoding ($SqRE$), and Gray-style square root encoding ($GSqRE$).

Gray-style most compact encoding ($GSMCE$)

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 converting an existing symbol or adding a new symbol (with the remark that after this step cannot be performed an other addition step).

3.1 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	$GSMCE_2$	$SqRE_2$	$GSqRE_2$
0	$\lambda = a^0$	λ=0^01^0	λ=$0^01^02^0$	λ=0^01^0	λ=0^01^0	λ=0^01^0
1	a^1	0=0^11^0	0=$0^11^02^0$	0=0^11^0	00=0^21^0	00=0^21^0
2	a^2	1=0^01^1	1=$0^01^12^0$	1=0^01^1	01=0^11^1	01=0^11^1
3	a^3	00=0^21^0	2=$0^01^02^1$	11=0^01^2	11=0^01^2	11=0^01^2
4	a^4	01=0^11^1	00=$0^21^02^0$	10=0^11^1	0000=0^41^0	1111=0^01^4
5	a^5	11=0^01^2	01=$0^11^12^0$	00=0^21^0	0001=0^31^1	1110=0^11^3
6	a^6	000=0^31^0	02=$0^11^02^1$	000=0^31^0	0011=0^21^2	1100=0^21^2
7	a^7	001=0^21^1	11=$0^01^22^0$	001=0^21^1	0111=0^11^3	1000=0^11^3
8	a^8	011=0^11^2	12=$0^01^12^1$	011=0^11^2	1111=0^01^4	0000=0^41^0
9	a^9	111=0^01^3	22=$0^01^02^2$	111=0^01^3	000000=0^61^0	000000=0^61^0
10	a^{10}	0000=0^41^0	000=$0^31^02^0$	1111=0^01^4	000001=0^51^1	000001=0^51^1
11	a^{11}	0001=0^31^1	001=$0^21^12^0$	1110=0^31^1	000011=0^41^2	000011=0^41^2
12	a^{12}	0011=0^21^2	002=$0^21^02^1$	1100=0^21^2	000111=0^31^3	000111=0^31^3
13	a^{13}	0111=0^11^3	011=$0^11^22^0$	1000=0^11^3	001111=0^21^4	001111=0^21^4

Table 1. Number encodings

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

Definitions, properties

We denote as $N(b,m)$ the number of numbers encoded in base b with m objects.

Here are some values for the function $N(b,m)$:

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**.

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

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

Based on the Pascal formula:

$$\binom{n}{r} = \binom{n-1}{r} + \binom{n-1}{r-1}$$

and its extended form:

$$\binom{n}{r} = \sum_{i=0}^{r} \binom{n-1-i}{r-i} \tag{2}$$

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

$$N(b,m) = \left\langle \begin{matrix} b \\ m \end{matrix} \right\rangle = \binom{b-1+m}{m} = \binom{(b-1)-1+(m-i)}{(m-i)} =$$

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

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 this equation:

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

Since from 3

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

$$\binom{b+m-1}{m-1} = \frac{(b+m-1)!}{b!(m-1)!} = \frac{\prod_{i=0}^{b-1}(m+i)}{b!}$$

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 will represent m, i.e. the number of objects needed to represent the natural number n.

We also note that

$$\frac{\prod_{i=0}^{b-1}(m+i)}{b!} = \sum_{i=1}^{b} \begin{bmatrix} b \\ i \end{bmatrix} m^i \tag{5}$$

where $\begin{bmatrix} b \\ i \end{bmatrix}$ 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 $n = O(b^m)$.

Most compact binary encoding (MCE_2)

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

We can say that the binary case of the most compact encoding using one membrane is a Cantor encoding, just because the related inverse of the Cantor pairing function. The Cantor pairing function is the bijection π:$\mathbb{N}$x$\mathbb{N} \rightarrow \mathbb{N}$ defined by $\pi(k_1, k_2) = \frac{1}{2}(k_1 + k_2)(k_1 + k_2 + 1) + k_2$. In fact, we consider this encoding as a variant of the Hopcroft-Ullman function. Cantor pairing function and Hopcroft-Ullman function 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 [10].

Here we present an explicit description of this encoding; other descriptions using the fact that the encoding is the inverse of the Cantor pairing function are possible.

Encoding

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.

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.

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

Decoding

We decode the number encoded using m objects with k objects of type 1 as $n = \frac{m(m+1)}{2} + k$.

4 P Systems for MCE

We present the P systems that implement the arithmetic operations on numbers encoded using the most compact encodings. We provide the XML code which could be used to test the arithmetical operations defined in this paper, by using our WebPS simulator available at `http://psystems.ieat.ro` (see also [3]).

4.1 Natural encoding MCE_1

Addition
Time complexity: $O(1)$

Addition is trivial; we consider n objects a and m objects b. The rule $b \to a$ says that an object b is transformed in one object a. Such a rule is applied in parallel as many times as possible. Consequently, all objects b are erased. The remaining number of objects a represents the addition $n + m$.

$$\Pi = (V, \mu, w_0, (R_0, \emptyset), 0),$$
$$V = \{a, b\},$$

$$\begin{aligned}\mu &= [_0]_0,\\ w_0 &= a^n b^m,\\ R_0 &= \{b \to a\}.\end{aligned}$$

Simulator example:

```
<psystem>
  <membrane name="0">
    <object name="a" count="2" />
    <object name="a" count="4" />
    ...
  </membrane>
  <query text="count of (objects from 0)" />
</psystem>
```

Subtraction
Time complexity: $O(1)$

Subtraction is described in the following way: given n objects a and m objects b, a rule $ab \to \lambda$ says that one object a and one object b are deleted (this is represented by the empty symbol λ). Consequently, all the pairs ab are erased. The remaining number of objects represents the difference between n and m.

$$\begin{aligned}\Pi &= (V, \mu, w_0, (R_0, \emptyset), 0),\\ V &= \{a, b\},\\ \mu &= [_0]_0,\\ w_0 &= a^n b^m,\\ R_0 &= \{ab \to \lambda\}.\end{aligned}$$

Simulator example:

```
<psystem>
  <membrane name="0">
    <object name="a" count="6" />
    <object name="b" count="4" />
    <rule body="a+b->">
  </membrane>
  <query text="count of (objects from 0)" />
</psystem>
```

Multiplication

Figure 1 presents a P system Π_1 without promoters for multiplication of n (objects a) by m (objects b), the result being the number of objects d in membrane 0. In this P system we use the priority relation between rules; for instance $bv \to dev$ has a higher priority than $av \to u$, meaning the second rule is applied only when the first one cannot be applied anymore. Initially only the rule $au \to v$ can be applied, generating an object v which activates the rule $bv \to dev$ m times, and then $av \to u$. Now $eu \to dbu$ is applied m times, followed by $au \to v$. 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.

$$\begin{aligned}
\Pi_1 &= (V, \mu, w_0, (R_0, \rho_0), 0),\\
V &= \{a, b, e, v, u\},\\
\mu &= [_0]_0,\\
w_0 &= a^n b^m u,\\
R_0 &= \{r_1 : au \to v, r_2 : bv \to dev, r_3 : av \to u, r_4 : eu \to dbu\},\\
\rho_0 &= \{r_2 > r_1, r_4 > r_3\}.
\end{aligned}$$

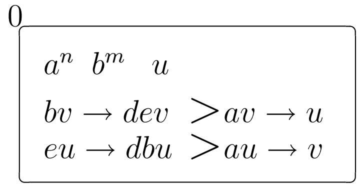

Fig. 1. Π_1 multiplier without promoters (natural encoding)

Simulator example:

```
<psystem>
  <membrane name="0">
    <object name="a" count="4"/>
    <object name="b" count="3"/>
    <object name="u" />
    <rule body="a+u->v"/>
    <rule body="b+v->e+v+d" priority="1"/>
    <rule body="a+v->u"/>
    <rule body="e+u->b+u+d" priority="1"/>
  </membrane>
  <query text="count of (objects from 0 where (objects d)) "/>
</psystem>
```

Figure 2 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. In this P system we use rule with priority and with promoters. The object a is a promoter in the rule $b \to bd|_a$, 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 $b \to bd|_a$ in parallel; based on the priority relation and the availability of a objects (except one a as promoter), the rule $au \to u$ is applied in the same time. The priority relation is motivated because the promoter a is a resource for which the rules $b \to bd|_a$and $au \to u$ are competing. 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.

$$\begin{aligned}
\Pi_2 &= (V, \mu, w_0, (R_0, \rho_0), 0),\\
V &= \{a, b, u\},\\
\mu &= [_0]_0,\\
w_0 &= a^n b^m u,\\
R_0 &= \{r_1 : b \to bd|_a, r_2 : au \to u\},\\
\rho_0 &= \{r_1 > r_2\}.
\end{aligned}$$

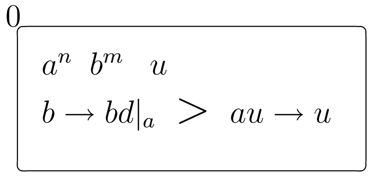

Fig. 2. Π_2 multiplier with promoters (natural encoding)

Simulator example:

```
<psystem>
  <membrane name="0">
    <object name="a" count="4"/>
    <object name="b" count="3"/>
    <object name="v" />
    <rule body="b->b+d|a" priority="1"/>
    <rule body="a+v->v"/>
  </membrane>
  <query text="count of (objects from 0 where (objects d)) "/>
</psystem>
```

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

The membrane systems for multiplication differ from others presented in the literature [9] because they do not have exponential space complexity, and do not require active membranes. 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. Another interesting feature is that the computation may continue after reaching a certain result, and so the system acts as a P transducer [7].

	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

Thus if initially there are n (objects a) and m (objects b), the system evolves and produces $n \cdot m$ objects d. Afterwards, 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

We implement division as repeated subtraction.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 $a \to b(v, in_1)$. The (v, in_1) notation means that the object v is injected into the child membrane 1. Therefore the rule $a \to b(v, in_1)$ 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 $av \to e$ applied n_1 times whenever possible. Two cases are distinguished in the inner membrane:

$$\begin{aligned}
\Pi &= (V, \mu, w_0, w_1, (R_0, \rho_0), (R_1, \rho_1), 0),\\
V &= \{a, b, b', c, s, u, v\},\\
\mu &= [_0[_1]_1]_0,\\
w_0 &= a^{n_1}s,\\
w_1 &= a^{n_2}s,\\
R_0 &= \{a \to b(v, in_1), b' \to a, r_1 : bu \to b'|_{\neg v}, r_2 : u \to \lambda|_{\neg v}, r_3 : csu \to u|_v\},\\
\rho_0 &= \{r_1 > r_2, r_2 > r_3\},\\
R_1 &= \{r_1 : av \to e, r_2 : v \to (v, out),\\
&\qquad r_3 : es \to s(u, out)(c, out), r_4 : e \to (u, out)\},\\
\rho_1 &= \{r_1 > r_2, r_2 > r_3, r_3 > r_4\}.
\end{aligned}$$

- If there are more objects a than objects v, only the rules $es \to s(u, out)(c, out)$ and $e \to (u, out)$ are applicable. The (u, out) notation means that the object u is sent out to the parent membrane. Rule $es \to s(u, out)(c, out)$ sends out to membrane 0 a single c (restricted by the existence of a single s into this membrane) for each subtraction step. The number of objects c represents the quotient. On the other hand, both rules send out n_1 objects u (equal to the number of objects e). The evolution continues in the outer membrane by applying $bu \rightarrow b'|_{\neg v}$ of $n_1$times, meaning the objects b are converted into objects b' by consuming the objects u only in the absence of v ($|_{\neg v}$ denotes an inhibitor having an effect opposite to that of a promoter). 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 $es \rightarrow s(u, out)(c, out)$ and $e \rightarrow (u, out)$ are applied. Due to the fact that we have objects v in membrane 0, the rule $bu \rightarrow b'|_{\neg v}$ cannot be applied. Since n_2 is not divisible by n_1, the number of the left objects u in membrane 0 represents the remainder of the division. A final cleanup is required in this case, because an object c is sent out even if we have not a "complete" subtraction step; the rule $ctu \rightarrow u|_v$ removes that extra c from membrane 0 in the presence of v. This rule is applied only once because we have a unique t in this membrane.

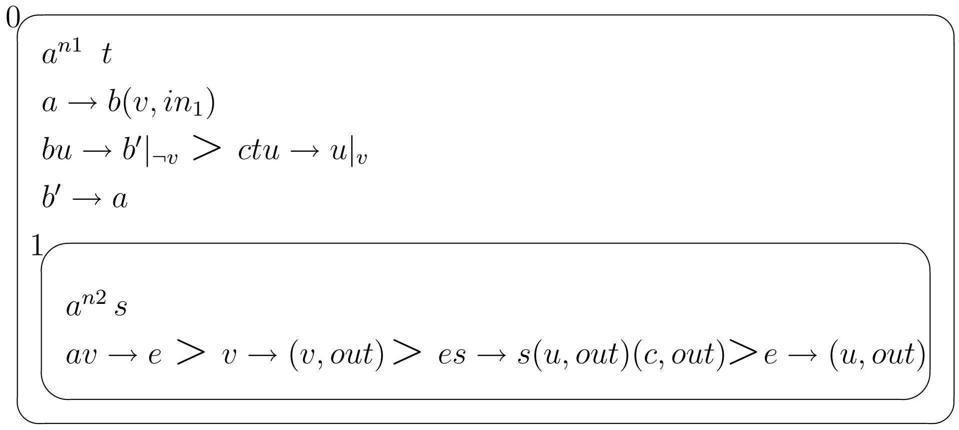
0
$a^{n1}\ t$
$a \rightarrow b(v, in_1)$
$bu \rightarrow b'|_{\neg v} > ctu \rightarrow u|_v$
$b' \rightarrow a$
1
$a^{n2}\ s$
$av \rightarrow e > v \rightarrow (v, out) > es \rightarrow s(u, out)(c, out) > e \rightarrow (u, out)$

Fig. 3. P system for division (natural encoding)

Simulator example:

```
<psystem>
  <membrane name="0">
    <object name="a" count="14" />
    <object name="s" count="1" />
    <rule body="a->b+v(1)" />
    <rule body="b+u->b’|!v" priority="2" />
    <rule body="u->|!v" priority="1"/>
```

```
      <rule body="b'->a" />
      <rule body="c+s+u->u|v" />
      <membrane name="1">
        <object name="a" count="26" />
        <object name="s" count="1" />
        <rule body="a+v->e" priority="3"/>
        <rule body="v->v(0)" priority="2" />
        <rule body="e+s->s+u(0)+c(0)" priority="1"/>
        <rule body="e->u(0)" />
      </membrane>
    </membrane>
    <query text='objects from *' />
    <query text='count of (objects from 0 where (objects u))' />
    <query text='count of (objects from 0 where (objects c))' />
  </psystem>
```

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

Successor MCE_2
Time complexity: $O(1)$

Complexity proof; by the evolution we can observe that:
- **If** an object 0 appears in the encoding then the rule $0s \to 1$ transforms a single 0 into an 1; **1 time unit**
- **else**, if the number is encoded using only objects 1, the rule $1 \to 0|_s$ transforms all objects 1 into 0, **1 time unit** (because of the maximal parallelism); and, the rule $s \to 0$ produces an additional 0, **1 time unit**.
Consequently, the time complexity of the successor is $O(1)$ because the evolution succeeds 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 $0s \to 1$ transforms this 0 into an 1, or we have a number encoded using only objects 1 and the rule $1 \to 0|_s$ transforms all 1s into 0s; moreover the rule $s \to 0$ produces an additional 0.

$$\begin{aligned}
\Pi &= (V, \mu, w_0, (R_0, \rho_0), 0),\\
V &= \{0, 1, s\},\\
\mu &= [_0]_0,\\
w_0 &= 0^{n-k}1^k s,\\
R_0 &= \{r_1 : 0s \to 1, r_2 : 1 \to 0|_s, r_3 : s \to 0\},\\
\rho_0 &= \{r_1 > r_2, r_2 > r_3\}.
\end{aligned}$$

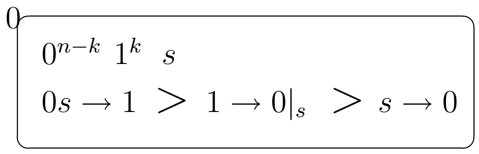

Fig. 4. Successor in MCE_2

Simulator example:

```
<psystem>
  <membrane name="0">
    <object name="0" count="4" />
    <object name="1" count="3" />
    <object name="s" />
    <rule body="0+s->1" priority="2" />
    <rule body="1->0|s" priority="1" />
    <rule body="s->0" />
  </membrane>
  <query text="objects from *" />
</psystem>
```

Predecessor MCE_2
Time complexity: $O(1)$

Complexity proof; by the evolution we can observe that:
- **If** an object 0 appears in the encoding then the rule $1s \to 0$ transforms a single 1 into an 0; **1 time unit**
- **else**, if the number is encoded using only objects 0, the rule $0s \to u$ transforms erase an objects 0, **1 time unit**; and, the rule $0 \to 1|_u$ transforms all the other objects 0 into 1, **1 time unit**(because of the maximal parallelism).
Consequently, the time complexity of the predecessor is $O(1)$ because the evolution succeeds 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 by the rule $0s \to u$, and transform all the other objects 0 into 1 by rule $0 \to 1|_u$.

$$\begin{aligned}
\Pi &= (V, \mu, w_0, (R_0, \rho_0), 0),\\
V &= \{0, 1, s\},\\
\mu &= [_0]_0,\\
w_0 &= 0^{n-k}1^k s,
\end{aligned}$$

$$R_0 = \{0 \to 1|_u, r_1 : 1s \to 0, r_2 : 0s \to u\},$$
$$\rho_0 = \{r_1 > r_2\}.$$

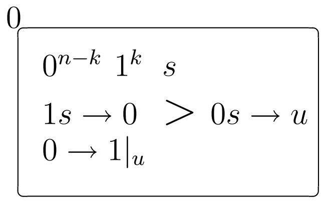

Fig. 5. Predecessor in MCE_2

Simulator example:

```
<psystem>
  <membrane name="0">
    <object name="0" count="4" />
    <object name="1" count="3" />
    <object name="s" />
    <rule body="1+s->0" priority="1" />
    <rule body="0+s->u" />
    <rule body="0->1|u" />
  </membrane>
  <query text="objects from *" />
</psystem>
```

Addition in MCE_2
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 $O(1)$ time complexity). Consequently, addition succeeds in $2n \cdot O(1) = O(2n) = O(n)$ time complexity.
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. 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.

$$\begin{aligned}
\Pi &= (V, \mu, w_0, (R_0, \rho_0), (R_1, \rho_1), 1),\\
V &= \{0, 1, s, p\},\\
\mu &= [_0[_1]_1]_0,\\
w_0 &= 0^{n_1-k_1}1^{k_1}s,\\
w_1 &= 0^{n_2-k_2}1^{k_2}p,\\
R_0 &= \{r_1 : 1s \to 0(s, in_1), r_2 : 0s \to u, r_3 : 0 \to 1|_u, r_4 : u \to (s, in_1)\},\\
\rho_0 &= \{r_1 > r_2, r_2 > r_3, r_3 > r_4\},\\
R_1 &= \{r_1 : ps \to (s, out), r_2 : 0s \to 1(s, out),\\
&\qquad r_3 : 1 \to 0|_s, r_4 : s \to 0(s, out)\},\\
\rho_1 &= \{r_1 > r_2, r_2 > r_3, r_3 > r_4\}.
\end{aligned}$$

Simulator example:

```
<psystem>
  <membrane name="0">
    <object name="1" count="1" />
    <object name="1" count="1" />
    <object name="s" />
    <rule body="1+s->0+s(1)" priority="3"/>
    <rule body="0+s->u" priority="2"/>
    <rule body="0->1|u" priority="1"/>
    <rule body="u->s(1)"/>
    <membrane name="1">
      <object name="0" count="1" />
      <object name="1" count="1" />
      <object name="p" />
      <rule body="p+s->s(0)" priority="3"/>
      <rule body="0+s->1+s(0)" priority="2"/>
      <rule body="1->0|s" priority="1"/>
      <rule body="s->0+s(0)"/>
    </membrane>
  </membrane>
  <query text="objects from *"/>
</psystem>
```

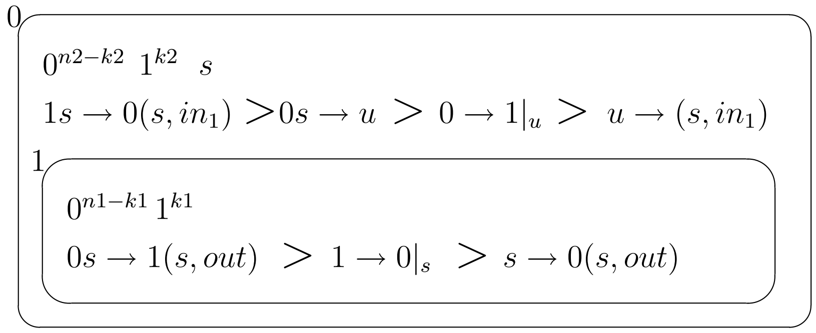
0
$0^{n2-k2}\ 1^{k2}\ \ s$
$1s \to 0(s, in_1) > 0s \to u > 0 \to 1|_u > u \to (s, in_1)$
1
$0^{n1-k1}\ 1^{k1}$
$0s \to 1(s, out) > 1 \to 0|_s > s \to 0(s, out)$

Fig. 6. Addition in MCE_2

Multiplication $\boldsymbol{MCE_2}$
Time complexity: $\boldsymbol{O(n1 \cdot n2) = O(n^2)}$, if $\boldsymbol{n1 = n2 = n}$.

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 succeeds in $n1 \cdot O(1) \cdot O(n2) = O(n1 \cdot n2)$ time complexity.
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.

$$\begin{aligned}
\Pi &= (V, \mu, w_0, (R_0, \rho_0), (R_1, \rho_1), (R_2, \rho_2), (R_{bk}, \rho_{bk}), 2),\\
V &= \{0, 1, p, q, s, u, w\},\\
\mu &= [_0[_1[_2]_2[_{bk}]_{bk}]_1]_0,\\
w_0 &= 0^{n_1-k_1}1^{k_1}s,
\end{aligned}$$

$$\begin{aligned}
w_1 &= q,\\
w_2 &= 0,\\
w_{bk} &= 0^{n_2-k_2}1^{k_2},\\
R_0 &= \{r_1 : 1s \to 0(w, in_1), r_2 : 0s \to u, r_3 : 0 \to 1|_u, r_4 : u \to (w, in_1)\},\\
\rho_0 &= \{r_1 > r_2, r_2 > r_3, r_3 > r_4\},\\
R_1 &= \{r_1 : 1s \to 0, r_2 : 0s \to u, r_3 : 0 \to 1|_u, r_4 : wq \to (s, out),\\
&\quad r_5 : w \to (w, in_{bk})(p, in_2)|_{\neg 0 \neg 1}, r_6 : s \to (s, out)|_{\neg 0 \neg 1},\\
&\quad u \to (s, in_2)\},\\
\rho_1 &= \{r_1 > r_2, r_1 > r_3, r_1 > r_4, r_1 > r_5, r_2 > r_6\},\\
R_2 &= \{r_1 : ps \to (s, out), r_2 : 0s \to 1(s, out),\\
&\quad r_3 : 1 \to 0|_s, r_4 : s \to 0(s, out)\},\\
\rho_2 &= \{r_1 > r_2, r_2 > r_3, r_3 > r_4\}\\
R_{bk} &= \{r_1 : 1 \to 1(1, out)|_w, r_2 : 0 \to (0, out), r_3 : w \to (s, out)\}\\
\rho_{bk} &= \{r_1 > r_3, r_2 > r_3\}.
\end{aligned}$$

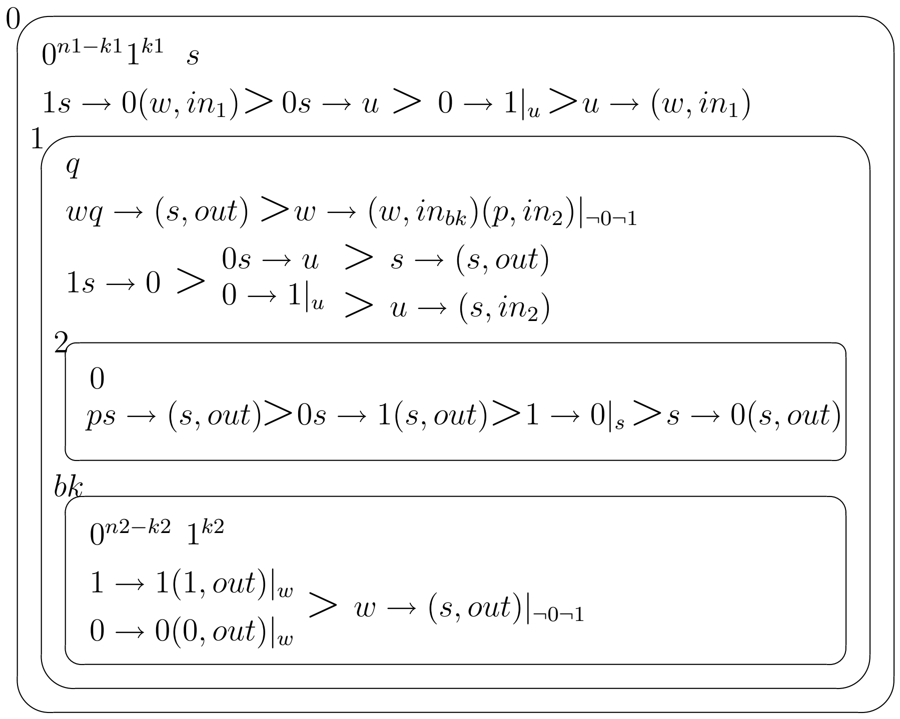
0
$0^{n1-k1}1^{k1}$ s
$1s \to 0(w, in_1) > 0s \to u > 0 \to 1|_u > u \to (w, in_1)$
1
q
$wq \to (s, out) > w \to (w, in_{bk})(p, in_2)|_{\neg 0 \neg 1}$
$1s \to 0 >$ $0s \to u > s \to (s, out)$
$0 \to 1|_u > u \to (s, in_2)$
2
0
$ps \to (s, out) > 0s \to 1(s, out) > 1 \to 0|_s > s \to 0(s, out)$
bk
0^{n2-k2} 1^{k2}
$1 \to 1(1, out)|_w$
$0 \to 0(0, out)|_w$
$> w \to (s, out)|_{\neg 0 \neg 1}$

Fig. 7. Multiplier in MCE_2

Simulator example:

```
<psystem>
  <membrane name="0">
    <object name="0" count="3"/>
    <object name="1" count="10"/>
    <object name="s"/>
    <rule body="1+s->0+w(1)" priority="3"/>
    <rule body="0+s->u" priority="2"/>
    <rule body="0->1|u" priority="1"/>
    <rule body="u->w(1)"/>
    <membrane name="1">
      <object name="q"/>
      <rule body="w+q->s(0)" priority="1"/>
      <rule body="w->w(bk)+p(2)||!0+!1"/>
      <rule body="s->s(0)||!0+!1"/>
      <rule body="1+s->0+s(2)" priority="3"/>
      <rule body="0+s->u" priority="2"/>
      <rule body="0->1|u" priority="1"/>
      <rule body="u->s(2)"/>
      <membrane name="bk">
        <object name="0" count="3"/>
        <object name="1" count="1"/>
        <rule body="1->1(1)+1|w" priority="1"/>
        <rule body="0->0(1)+0|w" priority="1"/>
        <rule body="w->s(1)"/>
      </membrane>
      <membrane name="2">
        <object name="0" count="1"/>
        <rule body="p+s->s(1)" priority="3"/>
        <rule body="0+s->1+s(1)" priority="2"/>
        <rule body="1->0|s" priority="1"/>
        <rule body="s->0+s(1)"/>
      </membrane>
    </membrane>
  </membrane>
  <query text="objects from 2"/>
  <query text="count of (objects from 2)"/>
  <query text="count of (objects from 2 where (objects 1))"/>
</psystem>
```

4.3 Multiple-iterations successor and predecessor

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

Complexity proof; by the P system evolution we observe:

- the rule $0s \to 1$ consumes as many s as possible (maximum m[= the length of n]); **1 time unit** (because of the maximal parallelism)

- the rule $su \to 0t$ generates an single 0 (the length of n is increasing by 1 $[m = m + 1]$); **1 time unit**

- the rule $1 \to 0|_t$ transforms all (m) objects 1 into 0; **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.

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 $p = \sum_{i=0}^{k}(m + 1) = k(m + 1)$; 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.

P system evolution

The multiple-iterations successor performs p successor iterations on the number n. The number 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.

$$
\begin{aligned}
\mathit{\Pi} &= (V, \mu, w_0, (R_0, \rho_0), 0),\\
V &= \{0, 1, s, t, u\},\\
\mu &= [_0]_0,\\
w_0 &= 0^{n-k}1^k s^p u,\\
R_0 &= \{r_1 : 0s \to 1, r_2 : su \to 0t, r_3 : 1 \to 0|_t, r_4 : t \to u\},\\
\rho_0 &= \{r_1 > r_2, r_2 > r_3, r_3 > r_4\}.
\end{aligned}
$$

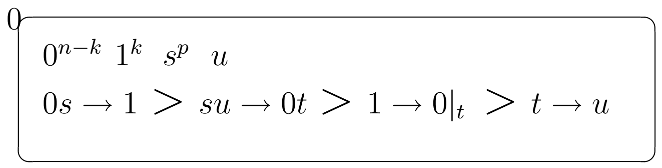

Fig. 8. Multiple-iterations successor

Simulator example:

```
<psystem>
 <membrane name="0">
   <object name="0" count="2"/>
   <object name="1" count="1"/>
   <object name="s" count="5"/>
   <object name="u"/>
   <rule body="0+s->1" priority="3"/>
   <rule body="s+u->0+t" priority="2"/>
   <rule body="1->0|t" priority="1"/>
   <rule body="t->u"/>
 </membrane>
 <query text="objects from *"/>
</psystem>
```

Multiple-iterations predecessor MCE_2
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 the maximal parallelism)
- 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 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, usualy $p > m$.

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, after which the rule $0 \to 1|_s$ transforms all 0s into 1s. The number of objects in the encoding is decreased by the rule $0su \to t$. 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.

$$\begin{aligned}
\Pi &= (V, \mu, w_0, (R_0, \rho_0), 0),\\
V &= \{0, 1, s, t, u\},\\
\mu &= [_0]_0,\\
w_0 &= 0^{n-k}1^k s^p u,\\
R_0 &= \{r_1 : 1s \to 0, r_2 : 0su \to t, r_3 : 0 \to 1|_s, r_4 : t \to u\},\\
\rho_0 &= \{r_1 > r_2, r_2 > r_3, r_3 > r_4\}.
\end{aligned}$$

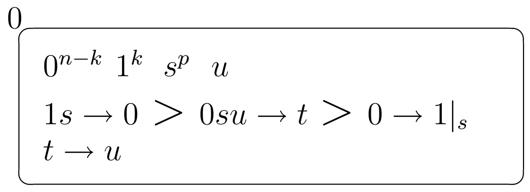
0
$0^{n-k}\ 1^k\ \ s^p\ \ u$
$1s \to 0 > 0su \to t > 0 \to 1|_s$
$t \to u$

Fig. 9. Multiple-iterations predecessor

Simulator example:

```
<psystem watchRules="false">
 <membrane name="0">
   <object name="0" count="1" />
   <object name="1" count="1" />
   <object name="s" count="5" />
   <object name="u" count="1" />
   <rule body="1+s->0" priority="2" />
```

```
    <rule body="0+s+u->t" priority="1"/>
    <rule body="0->1|s" />
    <rule body="t->u" />
  </membrane>
  <query text="objects from *" />
</system>
```

Decoder MCE_2
Time complexity: $O(m)$

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.

$$\begin{aligned}
\Pi &= (V, \mu, w_0, (R_0, \rho_0), 0),\\
V &= \{0, 1, d, t, u\},\\
\mu &= [_0]_0,\\
w_0 &= 0^{n-k}1^k u,\\
R_0 &= \{t \to u, r_1 : 1 \to 0d, r_2 : 0u \to td, r_3 : 0 \to 1\},\\
\rho_0 &= \{r_1 > r_2, r_2 > r_3\}.
\end{aligned}$$

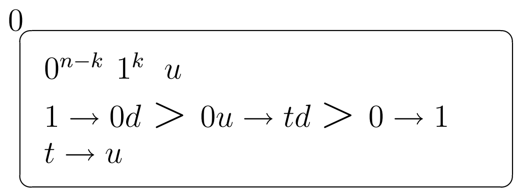

Fig. 10. MCE_2 decoding

Simulator example:

```
<psystem>
  <membrane name="0">
    <object name="0" count="1" />
    <object name="1" count="1" />
    <object name="u" count="1" />
    <rule body="1->0+d" priority="2" />
```

```
      <rule body="0+u->t+d" priority="1"/>
      <rule body="0->1" />
      <rule body="t->u" />
    </membrane>
    <query text="objects from *" />
  </system>
```

Optimized adder MCE_2
Time complexity: $O(m)$

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 ($O(m)$). Consequently, the time complexity of the optimized adder is $O(m)$.

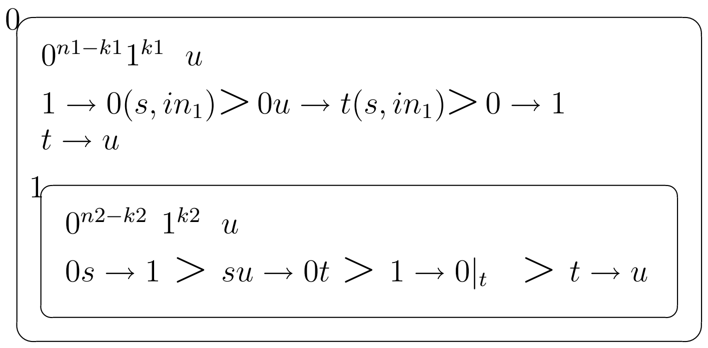
0
$0^{n1-k1}1^{k1}\ u$
$1 \to 0(s, in_1) > 0u \to t(s, in_1) > 0 \to 1$
$t \to u$
1
$0^{n2-k2}\ 1^{k2}\ u$
$0s \to 1 > su \to 0t > 1 \to 0|_t > t \to u$

Fig. 11. Optimized adder

P system evolution:
The optimized adder contains in membrane 0 a multiple-iteration predecessor, 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.

$$\Pi = (V, \mu, w_0, w_1, (R_0, \rho_0), (R_1, \rho_1), 1),$$

$$
\begin{aligned}
V &= \{0, 1, s, t, u\},\\
\mu &= [_0[_1]_1]_0,\\
w_0 &= 0^{n_1-k_1}1^{k_1}u,\\
w_1 &= 0^{n_2-k_2}1^{k_2}u,\\
R_0 &= \{t \to u, r_1 : 1 \to 0d, r_2 : 0u \to td, r_3 : 0 \to 1\},\\
\rho_0 &= \{r_1 > r_2, r_2 > r_3\},\\
R_1 &= \{r_1 : 0s \to 1, r_2 : su \to 0t, r_3 : 1 \to 0|_t, r_4 : t \to u\}\\
\rho_1 &= \{r_1 > r_2, r_2 > r_3, r_3 > r_4\}.
\end{aligned}
$$

Simulator example:

```
<psystem>
 <membrane name="1">
   <object name="0" count="2"/>
   <object name="0" count="4"/>
   <object name="u" count="1"/>
   <rule body="1->0+s(2)" priority="3"/>
   <rule body="0+u->t+s(2)" priority="2"/>
   <rule body="0->1" priority="1"/>
   <rule body="t->u"/>
   <membrane name="2">
     <object name="0" count="2"/>
     <object name="1" count="4"/>
     <object name="u"/>
     <rule body="0+s->1" priority="3"/>
     <rule body="s+u->0+t" priority="2"/>
     <rule body="1->0|t" priority="1"/>
     <rule body="t->u"/>
   </membrane>
 </membrane>
 <query text="objects from *"/>
</psystem>
```

4.4 Successor P systems for MCE_3
Time complexity: $O(1)$

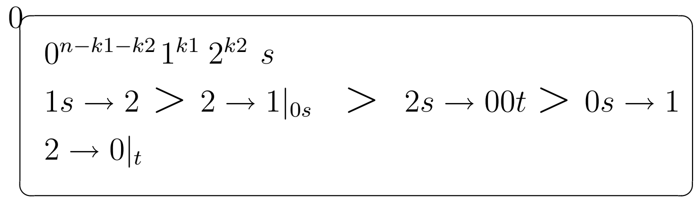
0
$0^{n-k1-k2}1^{k1}2^{k2}\ s$
$1s \to 2 > 2 \to 1|_{0s} > 2s \to 00t > 0s \to 1$
$2 \to 0|_t$

Fig. 12. Optimized adder

P system evolution
The successor of a number in this encoding is computed in the following manner, we have four possibilities: - either, we have an object 1 and the rule $1s \rightarrow 2$ transforms an 1 into an 2, - or, we have a number encoded using objects 0 and 2 then the rule $2 \rightarrow 1|_{0s}$ transforms all 2s into 1s, moreover the rule $0s \rightarrow 1$ erases an 0 and generates an 1 - or, the encoding contains only objects 2 then the rule $2s \rightarrow 00t$ transforms an object 2 into two objects 0 and a t which promotes the rule $2 \rightarrow 0|_t$. This rule transforms all others 2s into 0s. In this case the length of the encoding is increased. - or, the encoding contains only objects 0 and then the rule $0s \rightarrow 1$ transforms an object 0 into 1.

$$\begin{aligned}
\Pi &= (V, \mu, w_0, w_1, (R_0, \rho_0), 0),\\
V &= \{0, 1, 2, s\},\\
\mu &= [_0]_0,\\
w_0 &= 0^{n_1-k_1-k_2}1^{k_1}2^{k_2}s,\\
R_0 &= \{r_1 : 1s \rightarrow 2, r_2 : 2 \rightarrow 1|_{0s}, r_3 : 2s \rightarrow 00t, r_4 : 2 \rightarrow 0|_t, r_5 : 0s \rightarrow 1\},\\
\rho_0 &= \{r_1 > r_2, r_2 > r_3, r_2 > r_4, r_3 > r_5\}.
\end{aligned}$$

Simulator example:

```
<psystem>
 <membrane name="0">
   <object name="0" count="3"/>
   <object name="2" count="3"/>
   <object name="2" count="3"/>
   <object name="s"/>
   <rule body="1+s->2" priority="4"/>
   <rule body="2->1|s+0" priority="2"/>
   <rule body="2+s->0+0+t" priority="1"/>
   <rule body="2->0|t" priority="1"/>
   <rule body="0+s->1"/>
 </membrane>
 <query text="objects from *"/>
</psystem>
```

5 Conclusion

The idea to encode numbers using multisets and to define arithmetical operations in membrane computing is introduced in [2]. In this paper we extend the presentation including more details and complexity proofs, mentioning some new encodings, and adding arithmetical operations in more complex encodings.

The most compact encodings over the 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 $O(b^m)$.

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

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). 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.

We provide the XML code for each of the arithmetical operations defined in the paper, and use the web-based simulator of the P systems available at `http://psystems.ieat.ro/` to implement the arithmetical operations, and test each operation.

References

1. A. Atanasiu: Arithmetic with membranes. *Pre-proceedings of the Workshop on Multiset Processing (Curtea de Argeş)*, 2000, 1–17.
2. C. Bonchiş, G. Ciobanu, C. Izbaşa: Encodings and arithmetic operations in membrane computing. In *Theory and Applications of Models of Computation* (J.-Y. Cai, S. Barry Cooper, A. Li, eds.), LNCS 3959, Springer, 2006, 618–627.
3. C. Bonchiş, G.Ciobanu, C. Izbaşa, D. Petcu: A Web-based P systems simulator and its parallelization. In *Unconventional Computing* (C. Calude et al., eds.), LNCS 3699, Springer, 2005, 58–69.
4. B. Chen: Permutations and combinations. Electronic materials on *Combinations on Multisets*, Hong Kong University of Science and Technology, 2005.
5. G. Ciobanu, W. Guo: P systems running on a cluster of computers. In *Proceedings 4th Workshop on Membrane Computing*, LNCS 2933, Springer, 2004, 123–139.
6. G. Ciobanu, D. Paraschiv: P system software simulator. *Fundamenta Informaticae*, 49 (2002), 61–66.
7. G. Ciobanu, Gh. Păun, Gh. Ştefănescu: P transducers. *New Generation Computing*, 24 (2006), 1–28.
8. R.L. Graham, D.E. Knuth, O. Patashnik: *Concrete Mathematics.* Addison-Wesley, Reading, MA, 1990.
9. Gh. Păun: *Membrane Computing. An Introduction.* Springer, Berlin, 2002.
10. E.W. Weisstein et al.: "Pairing Function." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/PairingFunction.html