
Matter and Anti-Matter in Membrane Systems

Artiom Alhazov¹, Bogdan Aman², Rudolf Freund³, Gheorghe Păun⁴

¹ Institute of Mathematics and Computer Science, Academy of Sciences of Moldova
Academiei 5, MD-2028, Chişinău, Moldova
artiom@math.md

² Institute of Computer Science, Romanian Academy, Iaşi, Romania
bogdan.aman@iit.academiaromana-is.ro

³ Faculty of Informatics, Vienna University of Technology
Favoritenstr. 9, 1040 Vienna, Austria
rudi@emcc.at

⁴ Institute of Mathematics, Romanian Academy, Bucharest, Romania
gpaun@us.es

Summary. The concept of a matter object being annihilated when meeting its corresponding anti-matter object is investigated in the context of membrane systems, i.e., of (distributed) multiset rewriting systems applying rules in the maximally parallel way. Computational completeness can be obtained with using only non-cooperative rules besides these matter/anti-matter annihilation rules if these annihilation rules have priority over the other rules. Without this priority condition, in addition catalytic rules with one single catalyst are needed to get computational completeness. Even deterministic systems are obtained in the accepting case. Universal P systems with a rather small number of rules – 57 for computing systems, 59 for generating and 52 for accepting systems – can be constructed when using non-cooperative rules together with matter/anti-matter annihilation rules having weak priority. Allowing anti-matter objects as input and/or output, we even get a computationally complete computing model for computations on integer numbers. Interpreting sequences of symbols taken in from and/or sent out to the environment as strings, we get a model for computations on strings, which can even be interpreted as representations of elements of a group based on a computable finite presentation.

1 Introduction

Membrane systems (usually called *P systems*) can be considered as distributed multiset rewriting systems, where all objects – if possible – evolve in parallel in the membrane regions and may be communicated through the membranes. Membrane systems were introduced in [15] and since then have become an emerging field of research. Overviews can be found in the monograph [16] and the handbook of membrane systems [17]; for actual news and results we refer to the P systems

webpage [19]. Computational completeness (computing any partial recursive relation on non-negative integers) can be obtained with using cooperative rules or with catalytic rules (eventually) together with non-cooperative rules. In this paper, we use another concept to avoid cooperative rules in general: for any object a (*matter*), we consider its anti-object (*anti-matter*) a^- as well as the corresponding *annihilation rule* $aa^- \rightarrow \lambda$, which is assumed to exist in all membranes; this annihilation rule could be assumed to remove a pair a, a^- in zero time, but here we use these annihilation rules as special non-cooperative rules having priority over all other rules in the sense of weak priority (e.g., see [1], i.e., other rules then also may be applied if objects cannot be bound by some annihilation rule any more). The idea of anti-matter has already been considered in another special variant of P systems with motivation coming from modeling neural activities, which are known as spiking neural P systems; for example, spiking neural P systems with anti-matter (*anti-spikes*) were already investigated in [14]. Moreover, in [5] the power of anti-matter for solving NP-complete problems is exhibited.

As expected (for example, compare with the Geffert normal forms, see [18]), the annihilation rules are rather powerful. Yet it is still surprising that using matter/anti-matter annihilation rules as the only non-cooperative rules, with the annihilation rules having priority, we already get computational completeness without using any catalyst; without giving the annihilation rules priority, we need one single catalyst. Even more surprising is the result that with priorities we obtain *deterministic* systems in the case of accepting P systems. Moreover, we show how rather small universal P systems with anti-matter can be obtained based on the universal register machine U_{32} constructed by Korec, see [12]. Allowing anti-matter objects as input and/or output, we even get a computationally complete computing model for computations on integer numbers. Finally, by interpreting sequences of symbols taken in from and/or sent out to the environment as strings, we also consider P systems with anti-matter as computing/accepting/generating devices for string languages or even for languages over a group based on a computable finite presentation.

2 Prerequisites

The set of integers is denoted by \mathbb{Z} , while the set of non-negative integers by \mathbb{N} . Given an alphabet V , a finite non-empty set of abstract symbols, the free monoid generated by V under the operation of concatenation is denoted by V^* . The elements of V^* are called strings, the empty string is denoted by λ , and $V^* \setminus \{\lambda\}$ is denoted by V^+ . For an arbitrary alphabet $\{a_1, \dots, a_n\}$, the number of occurrences of a symbol a_i in a string x is denoted by $|x|_{a_i}$, while the length of a string x is denoted by $|x| = \sum_{a_i} |x|_{a_i}$. The Parikh vector associated with x with respect to a_1, \dots, a_n is $(|x|_{a_1}, \dots, |x|_{a_n})$. The Parikh image of an arbitrary language L over $\{a_1, \dots, a_n\}$ is the set of all Parikh vectors of strings in L , and is denoted by $Ps(L)$. For a family of languages FL , the family of Parikh images of languages in FL is denoted by $PsFL$, while for families of languages over a

one-letter (d -letter) alphabet, the corresponding sets of non-negative integers are denoted by NFL (N^dFL).

A (finite) multiset over a (finite) alphabet $V = \{a_1, \dots, a_n\}$, is a mapping $f : V \rightarrow \mathbb{N}$ and can be represented by $\langle a_1^{f(a_1)}, \dots, a_n^{f(a_n)} \rangle$ or by any string x for which $(|x|_{a_1}, \dots, |x|_{a_n}) = (f(a_1), \dots, f(a_n))$. In the following we will not distinguish between a vector (m_1, \dots, m_n) , a multiset $\langle a_1^{m_1}, \dots, a_n^{m_n} \rangle$ or a string x having $(|x|_{a_1}, \dots, |x|_{a_n}) = (m_1, \dots, m_n)$. Fixing the sequence of symbols a_1, \dots, a_n in an alphabet V in advance, the representation of the multiset $\langle a_1^{m_1}, \dots, a_n^{m_n} \rangle$ by the string $a_1^{m_1} \dots a_n^{m_n}$ is unique. The set of all finite multisets over an alphabet V is denoted by V° .

The family of regular and recursively enumerable string languages is denoted by REG and RE , respectively. For more details of formal language theory the reader is referred to the monographs and handbooks in this area as [4] and [18].

Register machines.

A *register machine* is a tuple $M = (m, B, l_0, l_h, P)$, where m is the number of registers, B is a set of labels, $l_0 \in B$ is the initial label, $l_h \in B$ is the final label, and P is the set of instructions bijectively labeled by elements of B . The instructions of M can be of the following forms:

- $l_1 : (ADD(j), l_2, l_3)$, with $l_1 \in B \setminus \{l_h\}$, $l_2, l_3 \in B$, $1 \leq j \leq m$.
Increases the value of register j by one, followed by a non-deterministic jump to instruction l_2 or l_3 . This instruction is usually called *increment*.
- $l_1 : (SUB(j), l_2, l_3)$, with $l_1 \in B \setminus \{l_h\}$, $l_2, l_3 \in B$, $1 \leq j \leq m$.
If the value of register j is zero then jump to instruction l_3 ; otherwise, the value of register j is decreased by one, followed by a jump to instruction l_2 . The two cases of this instruction are usually called *zero-test* and *decrement*, respectively.
- $l_h : HALT$. Stops the execution of the register machine.

A *configuration* of a register machine is described by the contents of each register and by the value of the current label, which indicates the next instruction to be executed. Computations start by executing the instruction l_0 of P , and terminate with reaching the HALT-instruction l_h .

3 P Systems

The basic ingredients of a (cell-like) P system are the membrane structure, the multisets of objects placed in the membrane regions, and the evolution rules. The *membrane structure* is a hierarchical arrangement of membranes, in which the space between a membrane and the immediately inner membranes defines a *region/compartiment*. The outermost membrane is called the *skin membrane*, the region outside is the *environment*. Each membrane can be labeled, and the label

(from a set Lab) will identify both the membrane and its region; the skin membrane is identified by (the label) 1. The membrane structure can be represented by an expression of correctly nested labeled parentheses, and also by a rooted tree (with the label of a membrane in each node and the skin in the root). The *multisets of objects* are placed in the compartments of the membrane structure and usually represented by strings of the form $a_1^{m_1} \dots a_n^{m_n}$.

The *evolution rules* are multiset rewriting rules of the form $u \rightarrow v$, where $u \in O^\circ$ and $v = (b_1, tar_1) \dots (b_k, tar_k)$ with $b_i \in O^\circ$ and $tar_i \in \{here, out, in\}$ or $tar_i \in \{here, out\} \cup \{in_j \mid j \in Lab\}$, $1 \leq i \leq k$. Using such a rule means “consuming” the objects of u and “producing” the objects from b_1, \dots, b_k of v , where the target *here* means that the objects remain in the same region where the rule is applied, *out* means that they are sent out of the respective membrane (in this way, objects can also be sent to the environment, when the rule is applied in the skin region), *in* means that they are sent to one of the immediately inner membranes, chosen in a non-deterministic way, and in_j means that they are sent into the specified inner membrane. In general, the target indication *here* is omitted.

Formally, a (cell-like) P system is a construct

$$\Pi = (O, \mu, w_1, \dots, w_m, R_1, \dots, R_m, l_{in}, l_{out})$$

where O is the alphabet of objects, μ is the membrane structure (with m membranes), w_1, \dots, w_m are multisets of objects present in the m regions of μ at the beginning of a computation, R_1, \dots, R_m are finite sets of evolution rules, associated with the regions of μ , l_{in} is the label of the membrane region where the inputs are put at the beginning of a computation, and l_{out} indicates the region from which the outputs are taken; l_{out}/l_{in} being 0 indicates that the output/input is taken from the environment.

If a rule $u \rightarrow v$ has $|u| > 1$, then it is called *cooperative* (abbreviated *coo*); otherwise, it is called *non-cooperative* (abbreviated *ncoo*). In *catalytic P systems* non-cooperative as well as *catalytic rules* of the form $ca \rightarrow cv$ are used, where c is a *catalyst* – a special object that never evolves and never passes through a membrane, but it just assists object a to evolve to the multiset v . In a *purely catalytic P system* only catalytic rules are allowed. In both catalytic and purely catalytic P systems, in their description O is replaced by O, C in order to specify those objects from O that are the catalysts in the set C .

The evolution rules are used in the non-deterministic maximally parallel way, i.e., in any computation step of Π a multiset of rules is chosen from the sets R_1, \dots, R_m in such a way that no further rule can be added to it so that the obtained multiset would still be applicable to the existing objects in the membrane regions $1, \dots, m$. A *configuration* of a system is given by the membranes and the objects present in the compartments of the system. Starting from a given *initial configuration* and applying evolution rules as described above, we get *transitions* among configurations; a sequence of transitions forms a computation. A computation is *halting* if it reaches a configuration where no rule can be applied any more.

In the *generative case*, a halting computation has associated a result, in the form of the number of objects present in membrane l_{out} in the halting configuration (l_{in} can be omitted). The set of non-negative integers and the set of (Parikh) vectors of non-negative integers obtained as results of halting computations in Π are denoted by $N_{gen}(\Pi)$ and $Ps_{gen}(\Pi)$, respectively.

In the *accepting case*, for $l_{in} \neq 0$, we accept all (vectors of) non-negative integers whose input, given as the corresponding numbers of objects in membrane l_{in} , leads to a halting computation (l_{out} can be omitted); the set of non-negative integers and the set of (Parikh) vectors of non-negative integers accepted in that way by halting computations in Π are denoted by $N_{acc}(\Pi)$ and $Ps_{acc}(\Pi)$, respectively.

For the input being taken from the environment, i.e., for $l_{in} = 0$, we need an additional target indication *come*; $(a, come)$ means that the object a is taken into the skin from the environment (all objects there are assumed to be available in an unbounded number). The multiset of all objects taken from the environment during a halting computation then is the multiset accepted by this accepting P system, which in this case we shall call a *P automaton* [3]; the set of non-negative integers and the set of (Parikh) vectors of non-negative integers accepted by halting computations in Π are denoted by $N_{aut}(\Pi)$ and $Ps_{aut}(\Pi)$, respectively.

A P system Π can also be considered as a system computing a partial recursive function (in the deterministic case) or even a partial recursive relation (in the non-deterministic case), with the input being given in a membrane region $l_{in} \neq 0$ as in the accepting case or being taken from the environment as in the automaton case. The corresponding functions/relations computed by halting computations in Π are denoted by $ZY_{\alpha}(\Pi)$, $Z \in \{Fun, Rel\}$, $Y \in \{N, Ps\}$, $\alpha \in \{acc, aut\}$.

Computational completeness for (generating) catalytic P systems can be achieved when using two catalysts or with three catalysts in purely catalytic P systems, and the same number of catalysts is needed for P automata; in accepting P systems, the number of catalysts increases with the number of components in the vectors of natural numbers to be analyzed [7]. It is a long-time open problem how to characterize the families of sets of (vectors of) natural numbers generated by (purely) catalytic P systems with only one (two) catalysts. Using additional control mechanisms as, for example, priorities or promoters/inhibitors, P systems with only one (two) catalyst(s) can be shown to be computationally complete, e.g., see Chapter 4 in [17]. Last year several other variants of control mechanism have been shown to lead to computational completeness in (purely) catalytic P systems using only one (two) catalyst(s), see [6], [9], and [10]. In this paper we are going to investigate the power of using matter/antimatter annihilation rules – with the astonishing result, that no catalysts are needed any more in case the annihilation rules have weak priority over the other rules.

The family of sets $Y_{\delta}(\Pi)$, $Y \in \{N, Ps\}$, $\delta \in \{gen, acc, aut\}$, computed by P systems with at most m membranes and cooperative rules and with non-cooperative rules is denoted by $Y_{\delta}OP_m(coo)$ and $Y_{\delta}OP_m(ncoo)$, respectively. The family of sets $Y_{\delta}(\Pi)$, $Y \in \{N, Ps\}$, $\delta \in \{gen, acc, aut\}$, computed by (purely)

catalytic P systems with at most m membranes and at most k catalysts is denoted by $Y_\delta OP_m(cat_k)$ ($Y_\delta OP_m(pcat_k)$). The following characterizations are known:

Theorem 1. For any $m \geq 1$ and any $Y \in \{N, Ps\}$,

$$YREG = Y_{gen}OP_m(ncoo) \subset Y_{gen}OP_m(coo) = YRE.$$

Theorem 2. For any $m \geq 1$, $d \geq 1$, $\delta \in \{gen, aut\}$,

$$\begin{aligned} Ps_{acc}OP_m(cat_{d+2}) &= Ps_{acc}OP_m(pcat_{d+3}) = N^dRE. \\ Ps_\delta OP_m(cat_2) &= Ps_\delta OP_m(pcat_3) = PsRE. \end{aligned}$$

4 Using Matter and Anti-Matter

This concept to be used in (catalytic) P systems is a direct generalization of the idea of anti-spikes from spiking neural P systems (see [14]): for each object a we introduce the anti-matter object a^- . We can look at these anti-matter objects a^- as objects of their own or else we may extend the notion of a (finite) multiset over the (finite) alphabet V , $V = \{a_1, \dots, a_n\}$, as a mapping $f : V \rightarrow \mathbb{N}$ to a mapping $f : V \rightarrow \mathbb{Z}$ now also allowing negative values. In a usual way, such an extended multiset on \mathbb{Z} is represented by $\langle a_1^{f(a_1)}, \dots, a_n^{f(a_n)} \rangle$. A unique string representation for such an extended multiset is obtained by assigning a string over the (ordered) alphabet $\langle a_1, a_1^-, \dots, a_n, a_n^- \rangle$ as $a_1^{f(a_1)} \dots a_n^{f(a_n)}$ such that $(a_i)^{-m}$, $m > 0$, is represented by $(a_i^-)^m$, $1 \leq i \leq n$. Any other string having the same Parikh vector with respect to the (ordered) alphabet $\langle a_1, a_1^-, \dots, a_n, a_n^- \rangle$ can be used for representing the multiset given by f as well.

As in spiking neural P systems with anti-spikes, also in cell-like P systems we might consider the annihilation of matter and anti-matter objects to happen in zero-time or in an intermediate step between normal derivation steps in the maximally parallel mode. Whenever related matter a and anti-matter a^- meet, they annihilate each other, as, for example, in an extended multiset on \mathbb{Z} matter a and anti-matter a^- cannot exist at the same moment, hence, also not in a string representing an extended multiset on \mathbb{Z} . Yet in this paper we consider both objects and anti-objects to be handled by usual evolution rules; the annihilation of matter and anti-matter objects then corresponds to an application of the (non-context-free!) rule $aa^- \rightarrow \lambda$. (Purely) catalytic P systems thus can be extended to

- allow for annihilation rules of the form $aa^- \rightarrow \lambda$ (and for catalytic annihilation rules $caa^- \rightarrow c$, where c is a catalyst) over an (ordered) alphabet $\langle a_1, a_1^-, \dots, a_n, a_n^- \rangle$, or
- to work on extended multisets over the (ordered) alphabet $\langle a_1, \dots, a_n \rangle$.

In contrast to the case described above, now in an instantaneous description of a configuration of a P system both matter and anti-matter objects may appear. When working with context-free or catalytic rules over an (ordered) alphabet $\langle a_1, a_1^-, \dots, a_n, a_n^- \rangle$, we may give the matter/anti-matter annihilation rules

weak priority over all other rules – in order to not have matter a and anti-matter a^- in some configuration at the same moment and let them “survive” for longer.

We now consider catalytic P systems extended by also allowing for annihilation rules $aa^- \rightarrow \lambda$, with these rules having weak priority over all other rules, i.e., other rules can only be applied if no annihilation rule could still bind the corresponding objects. The family of sets $Y_\delta(\Pi)$, $Y \in \{N, Ps\}$, $\delta \in \{gen, acc, aut\}$, and the family of functions/relations $ZY_\alpha(\Pi)$, $Z \in \{Fun, Rel\}$, $\alpha \in \{acc, aut\}$, computed by such extended P systems with at most m membranes and k catalysts is denoted by $Y_\delta OP_m(cat(k), antim/pri)$ and $ZY_\alpha OP_m(cat(k), antim/pri)$; we omit $/pri$ for the families without priorities.

The matter/anti-matter annihilation rules are so powerful that we only need the minimum number of catalysts, i.e., zero!

Theorem 3. *For any $n \geq 1$, $k \geq 0$, $Y \in \{N, Ps\}$, $\delta \in \{gen, acc, aut\}$, $\alpha \in \{acc, aut\}$, and $Z \in \{Fun, Rel\}$,*

$$\begin{aligned} Y_\delta OP_n(cat(k), antim/pri) &= YRE \text{ and} \\ ZY_\alpha OP_n(cat(k), antim/pri) &= ZYRE. \end{aligned}$$

Proof. Let $M = (m, B, l_0, l_h, P)$ be a register machine. We now construct a one-membrane P system, initially containing only the object l_0 , which simulates M . The contents of register r is represented by the number of copies of the object a_r , $1 \leq r \leq m$, and for each object a_r we also consider the corresponding anti-object a_r^- . The instructions of M are simulated as follows:

- $l_1 : (ADD(j), l_2, l_3)$, with $l_1 \in B \setminus \{l_h\}$, $l_2, l_3 \in B$, $1 \leq j \leq m$.
Simulated by the rules

$$l_1 \rightarrow a_r l_2 \text{ and } l_1 \rightarrow a_r l_3.$$

- $l_1 : (SUB(r), l_2, l_3)$, with $l_1 \in B \setminus \{l_h\}$, $l_2, l_3 \in B$, $1 \leq r \leq m$.
As rules common for all simulations of SUB-instructions, we have

$$a_r^- \rightarrow \#^-, \quad 1 \leq r \leq m,$$

and the annihilation rules

$$a_r a_r^- \rightarrow \lambda, \quad 1 \leq r \leq m, \text{ and } \#\#^- \rightarrow \lambda$$

as well as the trap rules

$$\#^- \rightarrow \#\# \text{ and } \# \rightarrow \#\#;$$

these last two rules lead the system into an infinite computation whenever a trap symbol is left without being annihilated.

The *zero test* for instruction l_1 is simulated by the rules

$$l_1 \rightarrow l_1' a_r^- \text{ and } l_1' \rightarrow \# l_3.$$

The symbol $\#$ generated by the second rule $l_1' \rightarrow \#l_3$ can only be eliminated if the anti-matter a_r^- generated by the first rule $l_1 \rightarrow l_1'a_r^-$ is not annihilated by a_r , i.e., only if register r is empty.

The *decrement case* for instruction l_1 is simulated by the rule

$$l_1 \rightarrow l_2a_r^-.$$

The anti-matter a_r^- either correctly annihilates one matter a_r thus decrementing the register r or else traps an incorrect guess by forcing the symbol a_r^- to evolve to $\#^-$ and then to $\#\#$ in the next two steps in case register r is empty.

- $l_h : HALT$. Simulated by $l_h \rightarrow \lambda$.

When the computation in M halts, the object l_h is removed, and no further rules can be applied provided the simulation has been carried out correctly, i.e., if no trap symbols $\#$ are present in this situation. The remaining objects in the system represent the result computed by M . \square

Without this priority of the annihilation rules, the construction is not working, hence, a characterization of the families $Y_\delta OP_n(ncoo, antim)$ as well as $ZY_\alpha OP_n(ncoo, antim)$ remains as an open problem. Yet in addition using catalytic rules with one catalyst again allows us to obtain computational completeness:

Theorem 4. For any $n \geq 1$, $k \geq 1$, $Y \in \{N, Ps\}$, $\delta \in \{gen, acc, aut\}$, $\alpha \in \{acc, aut\}$, and $Z \in \{Fun, Rel\}$,

$$\begin{aligned} Y_\delta OP_n(cat(k), antim) &= YRE \quad \text{and} \\ ZY_\alpha OP_n(cat(k), antim) &= ZYRE. \end{aligned}$$

Proof. We again consider a register machine $M = (m, B, l_0, l_h, P)$ as in the previous proof, and construct the catalytic P system

$$\begin{aligned} \Pi &= (O, \{c\}, []_1, cl_0, R_1, l_{in}, 1) \text{ with} \\ O &= \{a_r, a_r^- \mid 1 \leq r \leq m\} \cup \{l, l', l'' \mid l \in B\} \cup \{\#, \#^-, d\}, \end{aligned}$$

with the single catalyst c in the skin membrane. The results now are sent to the environment, in order not to have to count the catalyst in the skin membrane; for that purpose, we simply use the rule $a_i \rightarrow (a_i, out)$ for the output symbols a_i (we assume that output registers of M are only incremented).

For each ADD-instruction $l_1 : (ADD(j), l_2, l_3)$ in P , we again take the rules

$$l_1 \rightarrow a_r l_2 \text{ and } l_1 \rightarrow a_r l_3.$$

For each SUB-instruction $l_1 : (SUB(r), l_2, l_3)$, we now consider the four rules

$$\begin{aligned} l_1 &\rightarrow l_2 a_r^-, \\ l_1 &\rightarrow l_1' d a_r^-, \\ l_1' &\rightarrow l_1', \text{ and} \\ l_1' &\rightarrow \# l_3. \end{aligned}$$

As rules common for all SUB-instructions, we again add the matter/antimatter annihilation rules

$$a_r a_r^- \rightarrow \lambda \text{ and } \#\#^- \rightarrow \lambda$$

as well as the trap rules

$$\# \rightarrow \#\# \text{ and } \#^- \rightarrow \#\#,$$

but in addition, also

$$d \rightarrow \#\#$$

as well as the catalytic rules

$$cd \rightarrow c \text{ and } ca_r^- \rightarrow c\#^-, 1 \leq r \leq m.$$

The decrement case is simulated as in the previous proof, by using the rule $l_1 \rightarrow l_2 a_r^-$ and then applying the annihilation rule $a_r a_r^- \rightarrow \lambda$. The zero-test now is initiated with the rule $l_i \rightarrow l'_i d a_r^-$ thus introducing the (dummy) symbol d which keeps the catalyst busy for one step, where the catalytic rule $cd \rightarrow c$ has to be applied in order to avoid the application of the trap rule $d \rightarrow \#\#$. If register r is empty, then a_r^- cannot be annihilated and therefore evolves to $\#^-$ in the third step by the application of the catalytic rule $ca_r^- \rightarrow c\#^-$, which symbol $\#^-$ afterwards annihilates the symbol $\#$ generated by the rule $l'_i \rightarrow \#l_k$ in the same step; if register r is not empty, a_r^- is annihilated by some copy of a_r already in the first step, hence, the trap symbol $\#$ generated by the rule $l'_i \rightarrow \#l_k$ does not find its anti-matter $\#^-$ and therefore evolves to $\#\#$, thus leading to an infinite computation. Although the annihilation rule $a_r a_r^- \rightarrow \lambda$ now does not have priority over the catalytic rule $ca_r^- \rightarrow c\#^-$, maximal parallelism enforces $a_r a_r^- \rightarrow \lambda$ to be applied, if possible, already in the first step instead of $ca_r^- \rightarrow c\#^-$, as in a successful derivation the catalyst c first has to eliminate the dummy symbol d .

The rule $l_h \rightarrow \lambda$ is applied at the end of a successful simulation of the instructions of the register machine M , and the computation halts if no trap symbol $\#$ is present; the symbols sent out to the environment during the computation represent the result of this halting computation. \square

In the accepting case, with priorities, we can even simulate the actions of a deterministic register machine in a deterministic way, i.e., for each configuration of the system, there can be at most one multiset of rules applicable to it.

Theorem 5. *For any $n \geq 1$, $k \geq 0$, and $Y \in \{N, Ps\}$,*

$$Y_{detacc}OP_n(cat(k), antim/pri) = YRE \text{ and} \\ FunY_{detacc}OP_n(cat(k), antim/pri) = FunYRE.$$

Proof. We only show how the SUB-instructions of a register machine $M = (m, B', l_0, l_h, P)$ can be simulated in a deterministic way without introducing a trap symbol and therefore causing infinite loops by them:

Let $B = \{l \mid l : (SUB(r), l', l'') \in P\}$ and, for every register r ,

$$\begin{aligned} \tilde{M}_r &= \{\tilde{l} \mid l : (SUB(r), l', l'') \in P\}, & \hat{M}_r &= \{\hat{l} \mid l : (SUB(r), l', l'') \in P\}, \\ \tilde{M}_r^- &= \{\tilde{l}^- \mid l : (SUB(r), l', l'') \in P\}, & \hat{M}_r^- &= \{\hat{l}^- \mid l : (SUB(r), l', l'') \in P\}. \end{aligned}$$

We now take the rules

$$a_r^- \rightarrow \tilde{M}_r^- \hat{M}_r$$

and the annihilation rules $a_r a_r^- \rightarrow \lambda$ for every register r as well as $\hat{l}^- \rightarrow \lambda$ and $\tilde{l}^- \rightarrow \lambda$ for all $l \in B$. Then a SUB-instruction $l_1 : (SUB(r), l_2, l_3)$, with $l_1 \in B$, $l_2, l_3 \in B'$, $1 \leq r \leq m$, is simulated by the rules

$$\begin{aligned} l_1 &\rightarrow \bar{l}_1 a_r^-, \\ \bar{l}_1 &\rightarrow \hat{l}_1^- (\tilde{M}_r \setminus \{\tilde{l}_1\}), \\ \hat{l}_1^- &\rightarrow l_2 (\tilde{M}_r^- \setminus \{\tilde{l}_1^-\}), \text{ and} \\ \tilde{l}_1^- &\rightarrow l_3 (\hat{M}_r^- \setminus \{\hat{l}_1^-\}). \end{aligned}$$

The symbol \hat{l}_1^- generated by the second rule is eliminated again and replaced by \tilde{l}_1^- if a_r^- is not annihilated (which indicates that the register is empty). \square

5 Small Universal P Systems with Anti-Matter

In [12], several variants of universal register machines were exploited. The main interesting variant for the results presented in this paper is shown in Figure 1.

In the diagram of the universal register machine U_{32} in Figure 1, the operations used on the registers are: the *zero-test* on register i is indicated by a rhomboid inclosing the encryption Ri , and in the case that the contents of register i is zero, the next operation is the one to be reached with the arc labeled by z ; the *increment* operation is depicted by a rectangle with the encryption RiP , and the *decrement* operation by a rectangle with the encryption RiM (as the decrement operation RiM is always preceded by the corresponding zero-test, it can always be carried out). The states are depicted directly at the corresponding operations; q_1 is the initial state, and the state where the U_{32} stops is indicated by $STOP$ in Figure 1.

Remark 1. The universal register machine U_{32} uses a very sophisticated number-theoretic encoding of the enumeration of a specific variant of register machines. The code of the register machine to be simulated is put into register 1, the input number into register 2 (where also the output will be computed). The instructions are decoded in the *Instruction reader* part and the *Decoder* part (which essentially performs a division by three), and these instructions work on the registers 0, 2, and 3 (as we know, e.g., see [13], three registers are sufficient to simulate any other register machine). Thus, U_{32} can compute any partial recursive function $f : \mathbb{N} \rightarrow \mathbb{N}$ (with input and output number in register 2) in the same way as the register machine encoded by the number in register 1 computes f . For the following

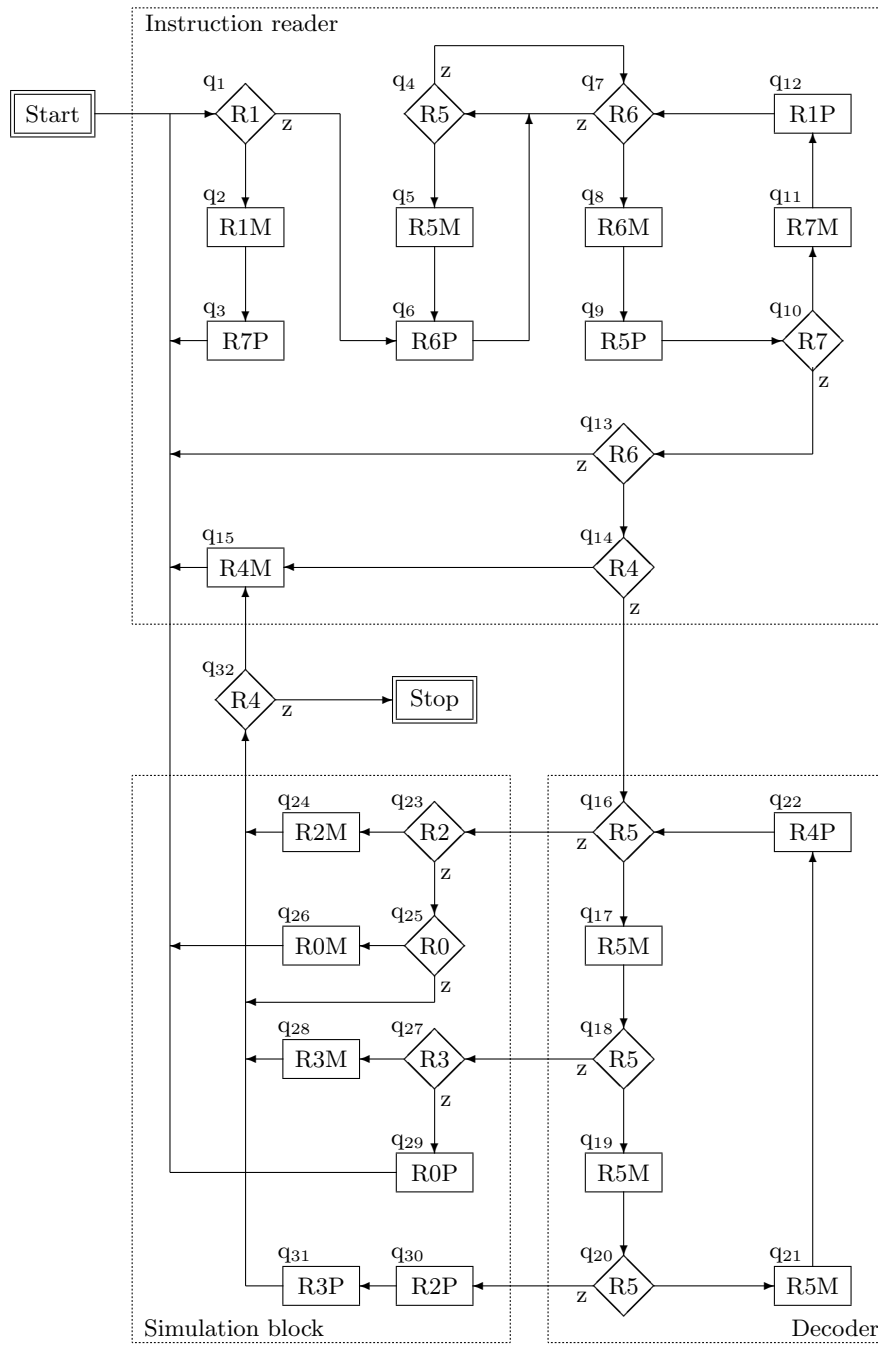


Fig. 1. The universal register machine U_{32} .

constructions it is important to note that U_{32} only stops when having finished the simulation of the register machine encoded in register 1 with the input in register 2, but enters an infinite computation otherwise.

A thorough analysis of the universal register machine U_{32} shows that when it halts not only register 2 as the output register, but also register 6 and register 1 (still containing the code of the register machine to be simulated) may be non-empty. On the other hand, due to the features of the 3-register machine as constructed in [13] and simulated by U_{32} in registers 0, 2, and 3, at the end of a computation registers 0 and 3 are empty (observe that the emptiness of these registers cannot be inferred from the program of U_{32}); in the accepting case, register 2 is empty, too.

In order to produce better descriptiveness complexity results with respect to the number of rules than those we would immediately get when applying the constructions given in the proof of Theorem 3, we introduce a generalization of register machines or counter automata.

Generalized counter automata.

For a register machine $M = (m, B, l_0, l_h, P)$ consider the more general type of instructions $i : (q, M_-, N, M_+, q')$, where $q, q' \in Q$ are states, $N \subseteq R$ is a set of registers, and M_-, M_+ are multisets of registers. Such a register machine applies instruction i as follows: first, multiset M_- is subtracted from the register values (i.e., for each register $j \in R$, $M_-(j)$ is subtracted from the contents of register j ; if at least one resulting value would be negative, the machine is blocked without producing any result); second, the subset N of registers is checked to be zero (if at least one of them is found to be non-zero, the machine is blocked without producing any result); third, the multiset M_+ is added to the register values (i.e., for each register $j \in R$, $M_+(j)$ is added to the contents of register j), and finally the state changes to q' .

The work of such a register machine, now also called a *generalized counter automaton* and written $M = (m, B, l_0, q_h, P)$, consists of derivation steps applying instructions, chosen in a non-deterministic way, associated with the current state. The computation starts in the initial state q_0 , and we say that it halts if the final state q_h has been reached (which replaces the condition of reaching the final HALT-instruction labeled by l_h).

Theorem 6. *There exist small universal P systems with non-cooperative rules and matter/anti-matter annihilation rules – with 9 annihilation rules and, in total, 52 rules in the accepting case, 59 rules in the generating case, and 57 rules in the computing case.*

Proof. We start with a translation of the P system from Theorem 4 in [8] (obtained from the universal register machine U_{32} machine in [12]). This sequential

antiport P system with forbidden contexts can be written with the instructions of a generalized counter automaton as follows:

$$\begin{array}{ll}
1 : (q_1, \langle 1 \rangle, \{\}, \langle 7 \rangle, q_1), & 10 : (q_{18}, \langle 5^3 \rangle, \{\}, \langle 4 \rangle, q_{18}), \\
2 : (q_1, \langle \rangle, \{1\}, \langle 6 \rangle, q_4), & 11 : (q_{18}, \langle \rangle, \{5, 3\}, \langle 0 \rangle, q_1), \\
3 : (q_4, \langle 5 \rangle, \{\}, \langle 6 \rangle, q_4), & 12 : (q_{18}, \langle 5^2, 0 \rangle, \{5, 2\}, \langle \rangle, q_1), \\
4 : (q_4, \langle 6 \rangle, \{5\}, \langle 5 \rangle, q_{10}), & 13 : (q_{18}, \langle 5^2, 2 \rangle, \{5\}, \langle \rangle, q_1), \\
5 : (q_{10}, \langle 7, 6 \rangle, \{\}, \langle 1, 5 \rangle, q_{10}), & 14 : (q_{18}, \langle 5^2 \rangle, \{5, 2, 0\}, \langle \rangle, q_1) \\
6 : (q_{10}, \langle 7 \rangle, \{6\}, \langle 1 \rangle, q_4), & 15 : (q_{18}, \langle 3, 4 \rangle, \{5\}, \langle \rangle, q_1), \\
7 : (q_{10}, \langle \rangle, \{6, 7\}, \langle \rangle, q_1), & 16 : (q_{18}, \langle 5, 4 \rangle, \{5\}, \langle 2, 3 \rangle, q_1). \\
8 : (q_{10}, \langle 6, 4 \rangle, \{7\}, \langle \rangle, q_1), & \\
9 : (q_{10}, \langle 6, 5 \rangle, \{7, 4\}, \langle \rangle, q_{18}), &
\end{array}$$

The system constructed in [8] halts when none of the rules 10 to 16 can be applied any more. For halting with the generalized counter automaton we have to define a halting state q_h and instructions how to reach this halting state. The details of this halting in the generalized counter automaton and how to halt in the P system to be constructed in the following, depending on the mode the automaton is used for – generating, accepting, computing – will be discussed later in the proof. We first show how these 16 instructions of the generalized counter automaton listed above can be simulated by a P system with anti-matter.

For a generalized counter automaton $M = (m, B, l_0, q_h, P)$, let

$$k = 1 + \max_{i:(q, M_-, N, M_+, q') \in P} (|M_-|, |N|).$$

We consider the following rules (common for different instructions of M):

$$\#^- \rightarrow \#^k, \# \rightarrow \#^k, \#\#^- \rightarrow \lambda, a_r \rightarrow \#^-, a_r a_r^- \rightarrow \lambda, r \in R.$$

Now we present the simulation of instruction $i : (q, M_-, N, M_+, q') \in P$. First we consider the case when M_- and N have no common elements, and moreover, we also assume that M_- does not overlap with M_+ (otherwise such an instruction can be split into two instructions; notice that this condition is already satisfied in the rules given above).

$$q \rightarrow l_i \prod_{r \in N} a_r^-, l_i \rightarrow q' (\prod_{r \in N} \#) (\prod_{r \in M_-} a_r^-) \prod_{r \in M_+} a_r.$$

Indeed, the zero-test is successful if *none* of the objects a_r^- generated in the first step annihilates with the corresponding register symbols a_r ; they have to change into objects $\#^-$ to annihilate with the same number of objects $\#$ produced in the next step. The decrement is successful if *all* objects a_r^- generated in the second step annihilate with the corresponding register symbols a_r . If either decrement or zero-test fail, then at least either one $\#$ or one $\#^-$ will be produced without its annihilation partner, leading to producing objects $\#$ in a geometric progression, ensuring that such computations do not produce any result (notice that no objects $\#$ or $\#^-$ are produced in the first step of the simulation of any instruction).

If the zero-test set N is empty, then the first step is a simple renaming, and thus can be combined with the second step, yielding just one rule

$$q \rightarrow q' \left(\prod_{r \in M_-} a_r^- \right) \prod_{r \in M_+} a_r.$$

Clearly, if M_- and N overlap, such an instruction can be broken down into two subsequent instructions of the generalized counter automaton. However, a more efficient solution with only three rules exists:

$$q \rightarrow l_i \prod_{r \in M_-} a_r^-, l_i \rightarrow l'_i \prod_{r \in N} a_r^-, l'_i \rightarrow q \left(\prod_{r \in N} \#^- \right) \prod_{r \in M_+} a_r.$$

The direct translation of the instructions of the generalized counter automaton given in the table at the beginning of the proof yields the following rules:

$$\begin{array}{ll} 1 : q_1 \rightarrow q_1 a_1^- a_7, & \\ 2 : q_1 \rightarrow l_2 a_1^-, & l_2 \rightarrow q_4 \# a_6, \\ 3 : q_4 \rightarrow q_4 a_5^- a_6, & \\ 4 : q_4 \rightarrow l_4 a_5^-, & l_4 \rightarrow q_{10} \# a_6^- a_5, \\ 5 : q_{10} \rightarrow q_{10} a_7^- a_6^- a_1 a_5, & \\ 6 : q_{10} \rightarrow l_6 a_6^-, & l_6 \rightarrow q_4 \# a_7^- a_1, \\ 7 : q_{10} \rightarrow l_7 a_6^- a_7^-, & l_7 \rightarrow q_1 \# \#, \\ 8 : q_{10} \rightarrow l_8 a_7^-, & l_8 \rightarrow q_1 \# a_6^- a_4^-, \\ 9 : q_{10} \rightarrow l_9 a_7^- a_4^-, & l_9 \rightarrow q_{18} \# \# a_6^- a_5^-, \\ 10 : q_{18} \rightarrow q_{18} a_5^- a_5^- a_5^- a_4, & \\ 11 : q_{18} \rightarrow l_{11} a_5^- a_3^-, & l_{11} \rightarrow q_1 \# \# a_0, \\ 12 : q_{18} \rightarrow l_{12} a_5^- a_5^- a_0^-, & l_{12} \rightarrow l'_{12} a_5^- a_2^-, \quad l'_{12} \rightarrow q_1 \# \#, \\ 13 : q_{18} \rightarrow l_{13} a_5^- a_5^- a_2^-, & l_{13} \rightarrow l'_{13} a_5^-, \quad l'_{13} \rightarrow q_1 \#, \\ 14 : q_{18} \rightarrow l_{14} a_5^- a_5^-, & l_{14} \rightarrow l'_{14} a_5^- a_2^- a_0^-, \quad l'_{14} \rightarrow q_1 \# \# \#, \\ 15 : q_{18} \rightarrow l_{15} a_5^-, & l_{15} \rightarrow q_1 \# a_3^- a_4^-, \\ 16 : q_{18} \rightarrow l_{16} a_5^- a_4^-, & l_{16} \rightarrow l'_{16} a_5^-, \quad l'_{16} \rightarrow q_1 \# a_2 a_3, \end{array}$$

In the *accepting case*, the input/output register 2 as well as the registers 0 and 3 are empty (see Remark 1); in order to reach the STOP in Figure 1, starting from q_{10} , register 6 must be non-empty, but registers 4, 5, 2, and 0 must be empty. Hence, we add the additional rule

$$17 : (q_{10}, \langle 6 \rangle, \{4, 5, 2, 0\}, \langle \rangle, q_h),$$

which is simulated by the rules

$$17 : q_{10} \rightarrow l_{17} a_4^- a_5^- a_2^- a_0^-, l_{17} \rightarrow q_h \# \# \# \# a_6^-.$$

As the rules with l_7 and l'_{12} on the left side have the same right side, we can replace l'_{12} by l_7 , thus decreasing the number of non-cooperative rules by one.

On the other hand, we have to add the rules

$$a_r \rightarrow \#^-, 0 \leq r \leq 7,$$

and the trap rules

$$\#^- \rightarrow \#^5 \text{ and } \# \rightarrow \#^5$$

as well as the annihilation rules

$$(\#\#^- \rightarrow \lambda) \text{ and } (a_r a_r^- \rightarrow \lambda), 0 \leq r \leq 7.$$

In sum, we obtain the universal P system with anti-matter

$$\begin{aligned} \Pi &= (O, []_1, q_1, R_1, 1) \text{ where} \\ O &= \{l_2, l_4, l_6, l_7, l_8, l_9, l_{11}, l_{12}, l_{13}, l'_{13}, l_{14}, l'_{14}, l_{15}, l_{16}, l'_{16}, l_{17}\} \\ &\cup \{q_1, q_4, q_{10}, q_{18}, q_h\} \cup \{a, a^- \mid a \in \{a_j \mid 0 \leq j \leq 7\} \cup \{\#\}\} \end{aligned}$$

with the rules in R_1 as described above; in total, the number of rules is 52, where 9 of them are the model-defined annihilation rules.

For the *computing case*, we have to re-introduce the state q_{32} of U_{32} , i.e., we replace instructions 15, 16, and 17 in the table of instructions for the generalized counter automaton by the instructions 15', 16', 17', and 18'; moreover we have to “clean” registers 1 and 6 (see Remark 1) and therefore to add the state q'_h and another three instructions:

$$\begin{aligned} 15' &: (q_{18}, \langle 3 \rangle, \{5\}, \langle \rangle, q_{32}), \\ 16' &: (q_{18}, \langle 5 \rangle, \{5\}, \langle 2, 3 \rangle, q_{32}), \\ 17' &: (q_{32}, \langle 4 \rangle, \{\}, \langle \rangle, q_1), \\ 18' &: (q_{32}, \langle \rangle, \{4\}, \langle \rangle, q'_h), \\ 19' &: (q'_h, \langle 1 \rangle, \{\}, \langle \rangle, q'_h), \\ 20' &: (q'_h, \langle 6 \rangle, \{\}, \langle \rangle, q'_h), \\ 21' &: (q'_h, \langle \rangle, \{1, 6\}, \langle \rangle, q_h). \end{aligned}$$

These instructions can be simulated by the following rules:

$$\begin{aligned} 15' &: q_{18} \rightarrow l_{15} a_5^-, \quad l_{15} \rightarrow q_{32} \# a_3^-, \\ 16' &: q_{18} \rightarrow l_{16} a_5^-, \quad l_{16} \rightarrow l'_{16} a_5^-, \quad l'_{16} \rightarrow q_{32} \# a_2 a_3, \\ 17' &: q_{32} \rightarrow q_1 a_4^-, \\ 18' &: q_{32} \rightarrow l_{18} a_4^-, \quad l_{18} \rightarrow q'_h \#, \\ 19' &: q'_h \rightarrow q'_h a_1^-, \\ 20' &: q'_h \rightarrow q'_h a_6^-, \\ 21' &: q'_h \rightarrow q_h a_1^- a_6^-, \quad q_h \rightarrow \#\#. \end{aligned}$$

In that way, we obtain a universal P system Π' with anti-matter having 57 rules, i.e., 49 non-cooperative rules and 9 model-defined annihilation rules:

$$\begin{aligned} \Pi' &= (O', []_1, q_1, R'_1, 1, 1) \text{ where} \\ O' &= \{l_2, l_4, l_6, l_7, l_8, l_9, l_{11}, l_{12}, l_{13}, l'_{13}, l_{14}, l'_{14}, l_{15}, l_{16}, l'_{16}, l_{18}\} \\ &\cup \{q_1, q_4, q_{10}, q_{18}, q_{32}, q'_h, q_h\} \cup \{a, a^- \mid a \in \{a_j \mid 0 \leq j \leq 7\} \cup \{\#\}\} \end{aligned}$$

and R'_1 contains the rules 1 to 14, 15' to 21' as well as the rules $a_r \rightarrow \#^-$, $0 \leq r \leq 7$, the trap rules $\#^- \rightarrow \#^4$ and $\# \rightarrow \#^4$ as well as the annihilation rules

$(\#\#^- \rightarrow \lambda)$ and $(a_r a_r^- \rightarrow \lambda)$, $0 \leq r \leq 7$. The P system now halts with the skin membrane only containing copies of the symbol a_2 representing the output value.

In fact, this construction could also be used for the accepting case, where we could already stop with q'_h , as we need not “clean” registers 1 and 6; in total this would yield only one rule more, i.e., 53 instead of 52 rules.

Finally, in the *generating case*, we start with the new initial state q_0 and add the two rules $q_0 \rightarrow a_2 q_0$ and $q_0 \rightarrow q_1$, which allows us to produce, in a non-deterministic way, an input for U_{32} simulating the identity function on the domain of the set to be generated by the P system, i.e., we get a P system with 59 rules. \square

6 When Matter/Anti-Matter Annihilation Generates Energy

The matter/anti-matter annihilation may also be assumed to result in the generation of a specific amount of “energy”, which is also well motivated by physics. In the definitions of these systems, the matter/anti-matter annihilation rules $a_r a_r^- \rightarrow \lambda$ are replaced by $a_r a_r^- \rightarrow e$ where e is a symbol denoting this special amount of energy.

The family of sets $Y_\delta(\Pi)$, $Y \in \{N, Ps\}$, $\delta \in \{gen, acc, aut\}$, and the set of functions/relations $ZY_\alpha(\Pi)$, $Z \in \{Fun, Rel\}$, $\alpha \in \{acc, aut\}$, computed by such P systems with at most m membranes and k catalysts is denoted by $Y_\delta OP_m(cat(k), antimen/pri)$ and $ZY_\alpha OP_m(cat(k), antimen/pri)$; we omit */pri* for the families without priorities.

The following results are immediate consequences of the corresponding Theorems 3 and 5 – in both cases, each matter/anti-matter annihilation rule $xx^- \rightarrow \lambda$ is replaced by $xx^- \rightarrow e$ where e is this symbol denoting a special amount of energy, and, in addition, we add the rule $e \rightarrow \lambda$:

Corollary 1. *For any $n \geq 1$, $k \geq 0$, $Y \in \{N, Ps\}$, $\delta \in \{gen, acc, aut\}$, $\alpha \in \{acc, aut\}$, and $Z \in \{Fun, Rel\}$,*

$$Y_\delta OP_n(cat(k), antimen/pri) = YRE \text{ and} \\ ZY_\alpha OP_n(cat(k), antimen/pri) = ZYRE.$$

Corollary 2. *For any $n \geq 1$, $k \geq 0$, and $Y \in \{N, Ps\}$,*

$$Y_{detacc} OP_n(cat(k), antimen/pri) = YRE \text{ and} \\ FunY_{detacc} OP_n(cat(k), antimen/pri) = FunYRE.$$

But we can even show more, i.e., omitting the rule $e \rightarrow \lambda$ and leaving the amount of energy represented by the number of copies of e in the system, the energy inside the system at the end of a successful computation is a direct measure for the number of SUB-instructions simulated by the P system or even a measure for the number of all instructions which were simulated.

Corollary 3. *The construction in the proof of Theorem 3 can be adapted in such a way that the simulation of each instruction of the register machine takes exactly three steps (including the annihilation rules), and moreover, the number of energy objects e at the end of a successful computation exactly equals the number of instructions simulated.*

Proof. Let $M = (m, B, l_0, l_h, P)$ be a register machine. Following the construction given in the proof of Theorem 3, the instructions of M now can be simulated as follows:

- $l_1 : (ADD(j), l_2, l_3)$, with $l_1 \in B \setminus \{l_h\}$, $l_2, l_3 \in B$, $1 \leq j \leq m$.
Simulated by the rules

$$\begin{aligned} l_1 &\rightarrow l_1', \\ l_1' &\rightarrow l_1'', \\ l_1'' &\rightarrow ea_r l_2, \\ l_1'' &\rightarrow ea_r l_3. \end{aligned}$$

- $l_1 : (SUB(r), l_2, l_3)$, with $l_1 \in B \setminus \{l_h\}$, $l_2, l_3 \in B$, $1 \leq r \leq m$.
As rules common for all simulations of SUB-instructions, we have

$$\begin{aligned} a_r^- &\rightarrow \#^-, \quad 1 \leq r \leq m, \\ a_r a_r^- &\rightarrow e, \quad 1 \leq r \leq m, \\ \#\#^- &\rightarrow e, \\ \#^- &\rightarrow \#\#, \\ \# &\rightarrow \#\#. \end{aligned}$$

The *zero test* for instruction l_1 is simulated by the rules

$$\begin{aligned} l_1 &\rightarrow l_1' a_r^-, \\ l_1' &\rightarrow \# l_1'', \text{ and} \\ l_1'' &\rightarrow l_3; \end{aligned}$$

- the symbol $\#$ generated by the second rule $l_1' \rightarrow \# l_1''$ can only be eliminated if the anti-matter a_r^- generated by the first rule $l_1 \rightarrow l_1' a_r^-$ is not annihilated by a_r , i.e., only if register r is empty; e is generated by $\#\#^- \rightarrow e$.

The *decrement case* for instruction l_1 is simulated by the rules

$$\begin{aligned} l_1 &\rightarrow \tilde{l}_1 a_r^-, \\ \tilde{l}_1 &\rightarrow \tilde{l}_1', \\ \tilde{l}_1' &\rightarrow l_2; \end{aligned}$$

- here, e is generated by $a_r a_r^- \rightarrow e$.

- $l_h : HALT$. Simulated by the rules

$$\begin{aligned} l_h &\rightarrow l_h', \\ l_h' &\rightarrow l_h'', \\ l_h'' &\rightarrow e. \end{aligned}$$

In each case, exactly one symbol e is generated during each cycle of three steps simulating an instruction of M . \square

Remark 2. Let M be a register machine and

$$RS(M) = \{(n, m) \mid n \in L(M), n \text{ is computed by } M \text{ in } m \text{ steps}\}.$$

Then, according to [2], RS is recursive. Hence, although $L(M)$ may not be recursive, $RS(M)$ is recursive in any case.

Now let $L \in NRE$ and $L = L(M)$ for a register machine M . Following the construction given in the proof of Corollary 3, we can construct a P system with energy Π such that $Ps(\Pi) = RS(M)$.

7 Computing with Integers

As already discussed in Section 4, given an alphabet $V = \{a_1, \dots, a_d\}$ we may extend the notion of a (finite) multiset over V as a mapping $f : V \rightarrow \mathbb{N}$ to a mapping $f : V \rightarrow \mathbb{Z}$ now also allowing negative values, with a unique string representation for such an extended multiset being obtained by assigning a string over the (ordered) alphabet $\langle a_1, a_1^-, \dots, a_d, a_d^- \rangle$ as $a_1^{f(a_1)} \dots a_d^{f(a_d)}$ such that $(a_i)^{-m}$, $m > 0$, is represented by $(a_i^-)^m$, $1 \leq i \leq d$. Besides this canonical representation of f by the string $a_1^{f(a_1)} \dots a_d^{f(a_d)}$, any other string having the same Parikh vector with respect to the (ordered) alphabet $\langle a_1, a_1^-, \dots, a_d, a_d^- \rangle$ can be used for representing the multiset given by f as well. According to these definitions, matter and related anti-matter cannot be present in the same string or multiset over the alphabet $\{a_1, a_1^-, \dots, a_d, a_d^-\}$. Obviously, there is a one-to-one correspondence between vectors from \mathbb{Z}^d and the corresponding Parikh vectors over $\langle a_1, a_1^-, \dots, a_d, a_d^- \rangle$, which can also be viewed as vectors over \mathbb{Z}^{2d} : for any of these vectors $v = (v_1, v_2, \dots, v_{2d-1}, v_{2d})$, we have either $v_{2i-1} = 0$ or $v_{2i} = 0$ (or both), for all $1 \leq i \leq d$.

In order to specify that now we are dealing with d -dimensional vectors of integer numbers, we use the notation $Ps^{\mathbb{Z}^d}$: the family of sets of integer numbers $Ps_{\delta}^{\mathbb{Z}^d}(\Pi)$, $\delta \in \{gen, acc, aut\}$, and the family of functions/relations $ZPs_{\alpha}^{\mathbb{Z}^d}(\Pi)$, $Z \in \{Fun, Rel\}$, $\alpha \in \{acc, aut\}$, computed by such P systems with at most m membranes and k catalysts is denoted by $Ps_{\delta}^{\mathbb{Z}^d}OP_m(cat(k), antim/pri)$ and $ZPs_{\alpha}^{\mathbb{Z}^d}OP_m(cat(k), antim/pri)$; we omit $/pri$ for the families without priorities. Moreover, the family of recursively enumerable sets of d -dimensional vectors of integer numbers is denoted by $Ps^{\mathbb{Z}^d}RE$, the corresponding functions/relations by $ZPs^{\mathbb{Z}^d}RE$.

Theorem 7. *For any $d \geq 1$ we have that:*

- for any $n \geq 1$, $k \geq 0$, $\delta \in \{gen, acc, aut\}$, $\alpha \in \{acc, aut\}$, and $Z \in \{Fun, Rel\}$,

$$Ps_{\delta}^{\mathbb{Z}^d}OP_n(cat(k), antim/pri) = Ps^{\mathbb{Z}^d}RE \text{ and} \\ ZPs_{\alpha}^{\mathbb{Z}^d}OP_n(cat(k), antim/pri) = ZPs^{\mathbb{Z}^d}RE;$$

- for any $n \geq 1$, $k \geq 1$, $\delta \in \{gen, acc, aut\}$, $\alpha \in \{acc, aut\}$, and $Z \in \{Fun, Rel\}$,

$$Ps_{\delta}^{\mathbb{Z}^d} OP_n(cat(k), antim) = Ps^{\mathbb{Z}^d} RE \text{ and} \\ ZPs_{\alpha}^{\mathbb{Z}^d} OP_n(cat(k), antim) = ZPs^{\mathbb{Z}^d} RE;$$

- for any $n \geq 1$, and $k \geq 0$,

$$Ps_{detacc}^{\mathbb{Z}^d} OP_n(cat(k), antim/pri) = Ps^{\mathbb{Z}^d} RE \text{ and} \\ FunPs_{detacc}^{\mathbb{Z}^d} OP_n(cat(k), antim/pri) = FunZPs^{\mathbb{Z}^d} RE.$$

Proof. As we have shown in Section 4, all variants of P systems with anti-matter mentioned in the theorem are computationally complete when dealing with multisets over any arbitrary alphabet, being able to simulate the actions of a register machine. Hence, as any d -dimensional vector of integer numbers can be represented by a $2d$ -dimensional vector of non-negative integers, which can be processed in the usual way by register machines and thus simulated by all the variants of P systems with anti-matter mentioned in the theorem, we only have to solve the technical detail how to get this $2d$ -dimensional vector of non-negative integers from a given d -dimensional vector of integer numbers represented by symbols over the (ordered) alphabet $\langle a_1, a_1^-, \dots, a_d, a_d^- \rangle$: given the input in an input membrane $\neq 0$, we there just make a first step using in parallel the non-cooperative rules $a_i \rightarrow [a_i, +]$ and $a_i^- \rightarrow [a_i, -]$, $1 \leq i \leq d$. Then the multisets over these symbols can be handled in the usual way, now both of them having the corresponding anti-matter objects $[a_i, +]^-$ and $[a_i, -]^-$. In a similar way, we can take the input from the environment by using rules of the form $q \rightarrow p(a_i, come)[a_i, +]$ or $q \rightarrow p(a_i^-, come)[a_i, -]$ where q, p represent states of the register machine. The symbols a_i and a_i^- then are not needed any more and can be eliminated by the rules $a_i \rightarrow \lambda$ and $a_i^- \rightarrow \lambda$. The remaining computations in the respective P system then can be carried out by simulating the actions of a register machine. \square

8 Computing with Languages

P systems with anti-matter, as most of the computationally complete variants of P systems, can also be considered as language generating devices – the objects sent out can be concatenated to strings over a given alphabet, and the objects taken in during a halting computation can be assumed to form a string. For sake of simplicity, we may assume that in each computation step, at most one symbol is sent out or taken in; otherwise, as usual, e.g., see [3], we may take any permutation of the symbols sent out or taken in to be part of a string to be considered as output or input, respectively. Obviously, according to this method of getting an input string, for the accepting case only the automaton variant is to be considered now, as otherwise we would have to take an encoding of the input string by a multiset.

8.1 Languages over Strings

Let V be a finite alphabet. The set of strings (over V) generated or accepted (in the sense of automata) by a P system with anti-matter Π is denoted by $L_\delta^V(\Pi)$, $\delta \in \{gen, aut\}$, the function/relation computed by Π is denoted by $ZL_{aut}^V(\Pi)$, $Z \in \{Fun, Rel\}$. The family of sets $L_\delta^V(\Pi)$, $\delta \in \{gen, aut\}$, and the family of functions/relations $ZL_{aut}^V(\Pi)$, $Z \in \{Fun, Rel\}$, computed by such P systems with at most m membranes and k catalysts is denoted by $L_\delta^V OP_m(cat(k), antim/pri)$ and $ZL_{aut}^V OP_m(cat(k), antim/pri)$, respectively; we omit */pri* for the families without priorities; $cat(0)$ is used as a synonym for *ncoo*. If the alphabet is arbitrary, we omit the superscript V in these notations. Moreover, the languages over V in RE are denoted by RE^V , the corresponding family of functions/relations by ZRE^V .

The use of anti-matter and of matter/anti-matter annihilation rules (having priority over other rules) allows us to give a simple example how to generate an even non-context-free string language:

Example 1. Consider the P system with anti-matter

$$\begin{aligned} \Pi &= (O, []_1, q_1, R_1, 1) \text{ where} \\ O &= \{a, b, c\} \cup \{b^-, c^-\} \cup \{q_1, q_2, q_3\}, \\ R_1 &= \{q_1 \rightarrow q_2, q_2 \rightarrow q_3, q_3 \rightarrow \lambda, q_1 \rightarrow q_1(a, come) b^- c^-\} \\ &\quad \cup \{q_2 \rightarrow q_2(b, come), q_3 \rightarrow q_3(c, come)\} \\ &\quad \cup \{a \rightarrow \lambda\} \cup \{x \rightarrow x, x^- \rightarrow x^-, xx^- \rightarrow \lambda \mid x \in \{b, c\}\}. \end{aligned}$$

The reader may easily verify that

$$L_{aut}^{\{a,b,c\}}(\Pi) = \{a^n b^n c^n \mid n \geq 0\}.$$

For each symbol a taken in with state q_1 (which is eliminated in the next step by $a \rightarrow \lambda$) using the rule $q_1 \rightarrow q_1(a, come) b^- c^-$, an anti-matter object for both b and c is generated. The anti-matter objects b^- are eliminated in state q_2 , and afterwards the anti-matter objects c^- are eliminated in state q_3 . The computation only halts (with empty skin membrane) after having used the rule $q_3 \rightarrow \lambda$ if and only if an equal number of objects a , b , and c has been taken in, as otherwise, the rules $x \rightarrow x$ or $x^- \rightarrow x^-$, $x \in \{b, c\}$, keep the system in an infinite loop if too many x or not enough x have been taken in, respectively. Observe that this system also works if we do not require priority of the annihilation rules, but then, for each successful computation accepting the string $a^n b^n c^n$, $n \geq 1$, there exist infinite computations where we use one of the rules $x^- \rightarrow x^-$ again and again instead of letting x^- being annihilated by $xx^- \rightarrow \lambda$. Hence, we may say that

$$\{a^n b^n c^n \mid n \geq 0\} \in L_{aut}^{\{a,b,c\}} OP_1(ncoo).$$

Theorem 8. *For any arbitrary alphabet V we have that:*

- for any $n \geq 1$, $k \geq 0$, $\delta \in \{gen, aut\}$, and $Z \in \{Fun, Rel\}$,

$$L_{\delta}^V OP_n(cat(k), antim/pri) = RE^V \text{ and} \\ ZL_{aut}^V OP_n(cat(k), antim/pri) = ZRE^V;$$

- for any $n \geq 1$, $k \geq 1$, $\delta \in \{gen, aut\}$, and $Z \in \{Fun, Rel\}$,

$$L_{\delta}^V OP_n(cat(k), antim) = RE^V \text{ and} \\ ZL_{aut}^V OP_n(cat(k), antim) = ZRE^V.$$

Proof. As we have shown in Section 4, all variants of P systems with anti-matter mentioned in the theorem are computationally complete when dealing with multisets, being able to simulate the actions of a register machine. Hence, by well-known techniques, input symbols composing an input string can be encoded as numbers in an input register and thus as a multiset in the simulating P system with anti-matter. In the same way, the results of a computation in the P system can be decoded from the multiset representing the output register of the underlying register machine. An input symbol $a \in V$ is taken in by rules of the form $q \rightarrow p(a, come)$ where q, p represent states of the register machine, and sent out by rules of the form $q \rightarrow p(a, out)$. \square

8.2 Languages over Computable Finite Presentations of Groups

Strings may be used in a wider sense as representations of group elements. In order to establish these more general results, we first need some definitions and examples from group theory, e.g., see [11].

Groups and Group Presentations

Let $G = (G', \circ)$ be a group with group operation \circ . As is well-known, the group axioms are

- *closure*: for any $a, b \in G'$, $a \circ b \in G'$,
- *associativity*: for any $a, b, c \in G'$, $(a \circ b) \circ c = a \circ (b \circ c)$,
- *identity*: there exists a (unique) element $e \in G'$, called the *identity*, such that $e \circ a = a \circ e$ for all $a \in G'$, and
- *invertibility*: for any $a \in G'$, there exists a (unique) element a^{-1} , called the *inverse* of a , such that $a \circ a^{-1} = a^{-1} \circ a = e$.

Moreover, the group is called *commutative*, if for any $a, b \in G'$, $a \circ b = b \circ a$. In the following, we will not distinguish between G' and G if the group operation is obvious from the context.

For any element $b \in G'$, the order of b is the smallest number $n \in \mathbb{N}$ such that $b^n = e$ provided such an n exists, and then we write $ord(b) = n$; if no such n exists, $\{b^n \mid n \geq 1\}$ is an infinite subset of G' and we write $ord(b) = \infty$.

For any set B , B^{-1} is defined as the set of symbols representing the inverses of the elements of B , i.e., $B^{-1} = \{b^{-1} \mid b \in B\}$. We now consider the strings in $(B \cup B^{-1})^*$ and two strings as different unless their equality follows from the group axioms, i.e., for any $a, b, c \in (B \cup B^{-1})^*$, $a \circ b \circ b^{-1} \circ c = a \circ c$; using these reductions, we obtain a set of irreducible strings from those in $(B \cup B^{-1})^*$, the set of which we denote by $I(B)$. Then the *free group* generated by B is $F(B) = (I(B), \circ)$ with the elements being the irreducible strings over $B \cup B^{-1}$ and the group operation to be interpreted as the usual string concatenation, yet, obviously, if we concatenate two elements from $I(B)$, the resulting string eventually has to be reduced again. The identity in $F(B)$ is the empty string.

In general, B (not containing the identity) is called a *generator* of the group G if every element a from G can be written as a finite product/sum of elements from B , i.e., $a = b_1 \circ \dots \circ b_m$ for $b_1, \dots, b_m \in B$. In this paper, we restrict ourselves to finitely presented groups, i.e., having a finite presentation $\langle B \mid R \rangle$ with B being a finite generator set and moreover, R being a finite set of relations among these generators. In a similar way as in the definition of the free group generated by B , we here consider the strings in B^* reduced according to the group axioms and the relations given in R . Informally, the group $G = \langle B \mid R \rangle$ is the largest one generated by B subject only to the group axioms and the relations in R . Formally, we will restrict ourselves to relations of the form $b_1 \circ \dots \circ b_m = c^{-1}$ with $b_1, \dots, b_m, c \in B$, which equivalently may be written as $b_1 \circ \dots \circ b_m \circ c = e$; hence, instead of such relations we may specify R by strings over B yielding the group identity, i.e., instead of $b_1 \circ \dots \circ b_m = c^{-1}$ we take $b_1 \circ \dots \circ b_m \circ c$ (these strings then are called *relators*).

Example 2. The free group $F(B) = (I(B), \circ)$ can be written as $\langle B \mid \emptyset \rangle$ (or even simpler as $\langle B \rangle$) because it has no restricting relations.

Example 3. The *cyclic group* of order n has the presentation $\langle \{a\} \mid \{a^n\} \rangle$ (or, omitting the set brackets, written as $\langle a \mid a^n \rangle$); it is also known as \mathbb{Z}_n or as the quotient group \mathbb{Z}/\mathbb{Z}_n .

Example 4. \mathbb{Z} is a special case of an Abelian group generated by (1) and its inverse (-1) , i.e., \mathbb{Z} is the free group generated by (1). \mathbb{Z}^d is an Abelian group generated by the unit vectors $(0, \dots, 1, \dots, 0)$ and their inverses $(0, \dots, -1, \dots, 0)$. It is well known that every finitely generated Abelian group is a direct sum of a torsion group and a free Abelian group where the torsion group may be written as a direct sum of finitely many groups of the form $\mathbb{Z}/p^k\mathbb{Z}$ for p being a prime, and the free Abelian group is a direct sum of finitely many copies of \mathbb{Z} .

Example 5. A very well-known example for a non-Abelian group is the hexagonal group with the finite presentation $\langle a, b, c \mid a^2, b^2, c^2 \rangle$. All three generators a, b, c are self-inverse.

Remark 3. Unfortunately, given a finite presentation of a group $\langle B \mid R \rangle$, in general it is not even decidable whether the group presented in that way is finite or

infinite. Hence, in this paper we restrict ourselves to infinite groups where the word equivalence problem $u = v$ is decidable, or equivalently, there is a decision procedure telling us whether, given two strings u and v , $u \circ v^{-1} = e$. In that case, we call $\langle B \mid R \rangle$ a *recursive* or *computable* finite group presentation.

As a first example we now consider the set (“language”) of all one-dimensional vectors:

Example 6. Consider the P system

$$\begin{aligned} \Pi &= (\{q_0, q_+, q_-, q_h\}, []_1, q_0, R_1, 1) \text{ where} \\ R_1 &= \{q_0 \rightarrow q_h, q_+ \rightarrow q_h, q_- \rightarrow q_h\} \\ &\cup \{q_0 \rightarrow (+1)q_+, q_+ \rightarrow (+1)q_+, q_0 \rightarrow (-1)q_-, q_- \rightarrow (-1)q_-\}. \end{aligned}$$

In order to generate the empty string, corresponding with the zero-vector (0), we simply apply $q_0 \rightarrow q_h$. We may also choose to generate a positive or a negative vector, i.e., we start with $q_0 \rightarrow (+1)q_+$ or $q_0 \rightarrow (-1)q_-$, respectively. After $n - 1$ applications of the rules $q_+ \rightarrow (+1)q_+$ and $q_- \rightarrow (-1)q_-$ as well as of the final rule $q_+ \rightarrow q_h$ or $q_- \rightarrow q_h$, respectively, we have sent out a string representing the unique irreducible representation of the vector $(+n)$ or $(-n)$, respectively.

Remark 4. The reader may easily verify that, given any finitely generated Abelian group, such a regular P system exists which generates all strings representing the (unique, with respect to a complete order on the positive generators) irreducible representations of the group elements. For non-commutative groups with relators, such trivial representations are not possible.

If we do not require irreducibility of the string sent out to the environment, then of course, for any finitely generated group, we can generate representations of all its elements very easily:

Example 7. Given a finite presentation of a group $\langle B \mid R \rangle$, with $B^- = B$, consider the P system

$$\begin{aligned} \Pi &= (\{q_0\}, []_1, q_0, R_1, 1) \text{ where} \\ R_1 &= \{q_0 \rightarrow \lambda\} \cup \{q_0 \rightarrow gq_0 \mid g \in B\}. \end{aligned}$$

Most of the strings sent out now will not be reduced.

Remark 5. In general, as long as we have given the group by a computable finite presentation, for a mechanism having the full power of Turing computability, we can require that the “strings” sent out to the environment are irreducible ones. Hence, for a given recursively enumerable set L of elements over the computable finite presentation $\langle B \mid R \rangle$ of a group, such a mechanism can generate the irreducible string representations of the elements in L . Thus, the results collected in the following theorem are obvious consequences of the results stated in Theorem 8.

Let $\langle B \mid R \rangle$ be the computable finite presentation of a group. The set of string representations (of elements of this group with respect to this finite presentation $\langle B \mid R \rangle$) generated or accepted (in the sense of automata) by a P system with anti-matter Π is denoted by $L_\delta^{\langle B \mid R \rangle}(\Pi)$, $\delta \in \{gen, aut\}$, the function/relation computed by Π is denoted by $ZL_{aut}^{\langle B \mid R \rangle}(\Pi)$, $Z \in \{Fun, Rel\}$. The family of sets $L_\delta^{\langle B \mid R \rangle}(\Pi)$, $\delta \in \{gen, aut\}$, and the family of functions/relations $ZL_{aut}^{\langle B \mid R \rangle}(\Pi)$, $Z \in \{Fun, Rel\}$, computed by such P systems with at most m membranes and k catalysts is denoted by $L_\delta^{\langle B \mid R \rangle}OP_m(cat(k), antim/pri)$ and $ZL_{aut}^{\langle B \mid R \rangle}OP_m(cat(k), antim/pri)$, respectively; we omit $/pri$ for the families without priorities. If the computable finite group presentation may be an arbitrary one, we omit the superscript $\langle B \mid R \rangle$ in these notations. The family of recursively enumerable sets of elements over the computable finite presentation $\langle B \mid R \rangle$ of a group is denoted by $RE^{\langle B \mid R \rangle}$, the corresponding family of recursively enumerable functions/relations by $ZRE^{\langle B \mid R \rangle}$, $Z \in \{Fun, Rel\}$.

Theorem 9. *Let $\langle B \mid R \rangle$ be the computable finite presentation of a group. Then we have that:*

- for any $n \geq 1$, $k \geq 0$, $\delta \in \{gen, aut\}$, and $Z \in \{Fun, Rel\}$,

$$L_\delta^{\langle B \mid R \rangle}OP_n(cat(k), antim/pri) = RE^{\langle B \mid R \rangle} \text{ and} \\ ZL_{aut}^{\langle B \mid R \rangle}OP_n(cat(k), antim/pri) = ZRE^{\langle B \mid R \rangle};$$

- for any $n \geq 1$, $k \geq 1$, $\delta \in \{gen, aut\}$, and $Z \in \{Fun, Rel\}$,

$$L_\delta^{\langle B \mid R \rangle}OP_n(cat(k), antim) = RE^{\langle B \mid R \rangle} \text{ and} \\ ZL_{aut}^{\langle B \mid R \rangle}OP_n(cat(k), antim) = ZRE^{\langle B \mid R \rangle}.$$

Proof. As for string languages, all computations can be carried out by simulating register machines, hence, again the results from Section 4 apply. Moreover, as already mentioned in Remark 5, the additional computations can also be carried out in this way, as $\langle B \mid R \rangle$ is computable. \square

Remark 6. Let us mention that the results obtained in Theorem 9 for arbitrary computable finite presentations $\langle B \mid R \rangle$ of a group can also be applied to the infinite Abelian groups \mathbb{Z}^d with their canonical group presentations by the unit vectors $(0, \dots, 1, \dots, 0)$ and their inverses $(0, \dots, -1, \dots, 0)$. Keeping in mind that there is a one-to-one correspondence between the representation of a vector in \mathbb{Z}^n by a multiset of symbols and the corresponding string representing this multiset, most of the results shown in Theorem 7 are special cases of the respective results stated in Theorem 9.

9 Summary

We have shown that only non-cooperative rules together with matter/anti-matter annihilation rules are needed to obtain computational completeness in P systems

working in the maximally parallel derivation mode if annihilation rules have weak priority; without priorities, one catalyst is needed. In the case of accepting P systems we were able to even get deterministic systems. Allowing anti-matter objects as input and/or output, we have even obtained a computationally complete computing model for computations on integer numbers. Interpreting sequences of symbols taken in from and/or sent out to the environment, we have also got a model for computations on strings, where strings can even be interpreted as representations of elements of a group based on a computable finite presentation.

There may be a lot of other interesting models of P systems allowing for introducing anti-matter objects and matter/anti-matter annihilation rules. Several problems remain open even for the models presented here, for example, can we avoid both catalysts and priorities. Moreover, the number of rules needed for universal P systems with anti-matter might still be reduced. Finally, the variants of P systems with anti-matter computing on sets of integer numbers and on languages of strings, even considered as representations of elements of a group based on a computable finite presentation, deserve more detailed investigations.

References

1. A. Alhazov, D. Sburlan: Static Sorting P Systems. In: G. Ciobanu, Gh. Păun, M.J. Pérez-Jiménez (Eds.): Applications of Membrane Computing. *Natural Computing Series*, Springer, 2005, pp. 215–252.
2. M. Cavaliere, R. Freund, A. Leitsch, Gh. Păun: Event-Related Outputs of Computations in P Systems. *Journal of Automata, Languages and Combinatorics* **11** (3), 263–278 (2006).
3. E. Csuhaj-Varjú, Gy. Vaszil: P Automata or Purely Communicating Accepting P Systems. In: Gh. Păun, G. Rozenberg, A. Salomaa, C. Zandron (Eds.): *Membrane Computing. International Workshop, WMC-CdeA 2002 Curtea de Argeș, Romania, August 19–23, 2002. Revised Papers*. Lecture Notes in Computer Science **2597**, Springer, 2003, pp. 219–233.
4. J. Dassow, Gh. Păun: *Regulated Rewriting in Formal Language Theory*. Springer, 1989.
5. D. Díaz-Pernil, F. Peña-Cantillana, M. A. Gutiérrez-Naranjo: Antimatter as a Frontier of Tractability in Membrane Computing. *Brainstorming Week in Membrane Computing*, Sevilla, February 2014.
6. R. Freund: Purely Catalytic P Systems: Two Catalysts Can Be Sufficient for Computational Completeness. In: A. Alhazov, S. Cojocaru, M. Gheorghe, Yu. Rogozhin (Eds.): *CMC14 Proceedings – The 14th International Conference on Membrane Computing, Chișinău, August 20–23, 2013*. Institute of Mathematics and Computer Science, Academy of Sciences of Moldova, 2013, pp. 153–166.
7. R. Freund, L. Kari, M. Oswald, P. Sosík: Computationally Universal P Systems without Priorities: Two Catalysts Are Sufficient. *Theoretical Computer Science* **330**, 251–266 (2005).
8. R. Freund, M. Oswald: A Small Universal Antiport P System with Forbidden Context. In: H. Leung, G. Pighizzini (Eds.): *8th International Workshop on Descriptive Complexity of Formal Systems - DCFs 2006*, Las Cruces, New Mexico, USA, June

- 21 - 23, 2006. Proceedings DCFSS, New Mexico State University, Las Cruces, New Mexico, USA, 2006, pp. 259–266.
9. R. Freund, M. Oswald: Catalytic and Purely Catalytic P Automata: Control Mechanisms for Obtaining Computational Completeness. In: S. Bensch, F. Drewes, R. Freund, F. Otto (Eds.): *Fifth Workshop on Non-Classical Models of Automata and Applications (NCMA 2013)*, OCG, Wien, 2013, pp. 133–150.
 10. R. Freund, Gh. Păun: How to Obtain Computational Completeness in P Systems with One Catalyst. In: T. Neary and M. Cook: *Proceedings Machines, Computations and Universality 2013, MCU 2013*, Zürich, Switzerland, September 9–11, 2013, *EPTCS* **128**, 47–61 (2013).
 11. D. F. Holt, B. Eick, E. A. O'Brien: *Handbook of Computational Group Theory*. CRC Press, 2005.
 12. I. Korec: Small Universal Register Machines. *Theoretical Computer Science* **168**, 267–301 (1996).
 13. M. L. Minsky: *Computation: Finite and Infinite Machines*. Prentice Hall, Englewood Cliffs, New Jersey, USA, 1967.
 14. L. Pan, Gh. Păun: Spiking Neural P Systems with Anti-Matter. *International Journal of Computers, Communications & Control* **4** (3), 273–282 (2009).
 15. Gh. Păun: Computing with Membranes. *Journal of Computer and System Sciences* **61** (1) (2000), 108–143 (and Turku Center for Computer Science-TUCS Report 208, November 1998, www.tucs.fi).
 16. Gh. Păun: *Membrane Computing. An Introduction*. Springer, 2002.
 17. Gh. Păun, G. Rozenberg, A. Salomaa (Eds.): *The Oxford Handbook of Membrane Computing*. Oxford University Press, 2010.
 18. G. Rozenberg, A. Salomaa (Eds.): *Handbook of Formal Languages*, 3 volumes. Springer, 1997.
 19. The P Systems Website: www.ppage.psystems.eu.