Energy-based Models of P Systems

Giancarlo Mauri, Alberto Leporati, Claudio Zandron

Dipartimento di Informatica, Sistemistica e Comunicazione
Università degli Studi di Milano – Bicocca
Viale Sarca 336/14, 20126 Milano, Italy

{mauri,leporati,zandron}@disco.unimib.it

Summary. Energy plays an important role in many theoretical computational models. In this paper we review some results we have obtained in the last few years concerning the computational power of two variants of P systems that manipulate energy while performing their computations: energy-based and UREM P systems. In the former, a fixed amount of energy is associated to each object, and the rules transform objects by manipulating their energy. We show that if we assign local priorities to the rules, then energy–based P systems are as powerful as Turing machines, otherwise they can be simulated by vector addition systems and hence are not universal. We also discuss the simulation of conservative and reversible circuits of Fredkin gates by means of (self)–reversible energy–based P systems. On the other side, UREM P systems are membrane systems in which a given amount of energy is associated to each membrane. The rules transform and move single objects among the regions. When an object crosses a membrane, it may modify the associated energy value. Also in this case, we show that UREM P systems reach the power of Turing machines if we assign a sort of local priorities to the rules, whereas without priorities they characterize the class $PsMAT^{\lambda}$, and hence are not universal.

1 Introduction

Membrane systems (also known as *P systems*) have been introduced in [13] as a parallel, nondeterministic, synchronous and distributed model of computation inspired by the structure and functioning of living cells. The basic model consists of a hierarchical structure composed by several membranes, embedded into a main membrane called the *skin*. Membranes divide the Euclidean space into *regions*, that contain multisets of *objects* (represented by symbols of an alphabet) and *evolution rules*. Using these rules, the objects may evolve and/or move from a region to a neighboring one. Usually, the rules are applied in a nondeterministic and maximally parallel way. A *computation* starts from an initial configuration of the system and terminates when no evolution rule can be applied. The result of a computation is the multiset of objects contained into an *output membrane*, or

emitted from the skin of the system. For a systematic introduction to P systems we refer the reader to [14], whereas the latest information can be found in [17].

Since the introduction of P systems, many investigations have been performed on their computational properties: in particular, many variants have been proposed in order to study the contribution of various ingredients (associated with the membranes and/or with the rules of the system) to the achievement of the computational power of these systems. In this paper we review some computational features of two models of membrane systems that manipulate *energy* while performing their computations: energy-based P systems and UREM P systems.

In *energy–based P systems*, a given amount of energy is associated to each object. Moreover, instances of a special symbol are used to denote free energy units occurring inside the system. These energy units can be used to transform objects, through appropriate rules that manipulate energy, while satisfying the principle of energy conservation. In particular, if the object to which the rule is applied contains less (more) energy than the one which has to be produced, then the necessary free energy units can be taken from (released to) the region where the rule is applied. We assume that the application of rules consumes no energy: in particular, objects can be moved between adjacent regions of the system without energy consumption. Rules are applied in a sequential manner: at each computation step, one of the enabled rules is nondeterministically selected and applied. We show that, if a potentially infinite amount of free energy units is available, then energy–based P systems are able to simulate register machines (hence, the model is universal). This is done by assigning a form of local priorities to the rules: if two or more rules can be applied in a given region, then the one which consumes or releases the largest amount of free energy units is applied (if two or more of the enabled rules manipulate exactly the same maximal amount of free energy, then one of them is nondeterministically chosen). Instead, if we disregard priorities, then energy–based P systems can be simulated by vector addition systems, and hence are not universal. On the other hand, if we do not allow the presence of an infinite amount of energy, then the power of energy–based P systems reduces to that of finite state automata, both when considering priorities associated with the rules and when disregarding them. We also show that energy–based P systems can be used to simulate reversible and conservative (that is, energy–preserving) boolean circuits composed of Fredkin gates; the simulating P systems are themselves reversible and logically complete, and so we have the possibility to compute *any* boolean function by energy–based P systems in a reversible way.

The second model of membrane systems we consider are *P systems with unit rules and energy assigned to membranes* (UREM P systems, for short). In these systems, the rules are directly assigned to membranes (and not to the regions, as it is usually done in membrane computing). Every membrane carries an energy value that can be changed during a computation by objects passing through the membrane. Also in this case, rules are applied in the sequential way. The input, as well as the result of a successful computation, are considered to be the distributions of energy values carried by the membranes in the initial and in the halting

configuration, respectively. We show that UREM P systems using a sort of local priority relation on the rules are Turing–complete. On the contrary, by omitting the priority relation we obtain a characterization of $PsMAT^{\lambda}$, the family of Parikh sets generated by context–free matrix grammars (with λ-rules and without occurrence checking). Alternatively, we can obtain Turing–completeness without using priorities, by applying rules in the maximally parallel mode.

The paper is organized as follows. In section 2 we recall the definition of three computational models that will be used throughout the paper, to study the computational power of energy–based and UREM P systems: register machines, vector addition systems, and Fredkin circuits. In sections 3 and 4 we review the computational power of energy–based and of UREM P systems, respectively. Section 5 concludes the paper and gives some directions for further research.

2 Preliminaries

In the following subsections we briefly recall the definition of three computational models that will be used in the rest of the paper to study the computational power of UREM and energy–based P systems.

2.1 Deterministic register machines

A *deterministic n–register machine* is a construct $M = (n, P, m)$, where $n > 0$ is the number of registers, P is a finite sequence of instructions (program) bijectively labelled with the elements of the set $\{1, 2, \ldots, m\}$, 1 is the label of the first instruction to be executed, and m is the label of the last instruction of P. Registers contain non–negative integer values. The instructions of P have the following forms:

- $j : (INC(r), k)$, with $j, k \in \{1, 2, \ldots, m\}$ and $r \in \{1, 2, \ldots, n\}$
  This instruction, labelled with j, increments (by 1) the value contained in register r, and then jumps to instruction k.
- $j : (DEC(r), k, l)$, with $j, k, l \in \{1, 2, \ldots, m\}$ and $r \in \{1, 2, \ldots, n\}$
  If the value contained in register r is positive, then decrement it (by 1) and jump to instruction k. If the value of r is zero, then jump to instruction l (without altering the contents of the register).
- $m : HALT$
  Stop the execution of the program. Note that, without loss of generality, we may assume that this instruction always appears exactly once in P, with label m.

Computations start by executing the first instruction of P (labelled with 1), and terminate when they reach instruction m. Register machines provide a simple universal computational model [12]. In particular, the results proved in [5] immediately lead to the following proposition.

Proposition 1. *For any partial recursive function* $f : \mathbb{N}^\alpha \to \mathbb{N}^\beta$ *there exists a deterministic* $(\max\{\alpha, \beta\} + 2)$*–register machine* M *computing* f *in such a way that, when starting with* $(n_1, \ldots, n_\alpha) \in \mathbb{N}^\alpha$ *in registers* 1 *to* α*,* M *has computed* $f(n_1, \ldots, n_\alpha) = (r_1, \ldots, r_\beta)$ *if it halts in the final label* m *with registers* 1 *to* β *containing* r_1 *to* r_β*, and all other registers being empty. If the final label cannot be reached, then* $f(n_1, \ldots, n_\alpha)$ *remains undefined.*

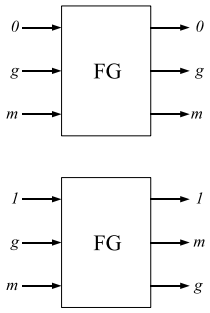
2.2 Vector addition systems

Vector addition systems were introduced in [7] as a mathematical tool for analyzing systems of parallel processes. It is known that they are not Turing–complete, as they are equivalent to self–loop–free Petri nets [16]. Formally, a vector addition system (VAS, for short) is a pair $V = (B, s)$, where $B = \{b_1, b_2, \ldots, b_m\}$ is a set of m vectors, called *basis* or *displacement* vectors, and s is the *start* vector. All vectors consist of n integer values. The elements of s are non–negative (in what follows, we denote this as $s \geq 0$). The *reachability set* $R(V)$ for a VAS V is the smallest set of vectors such that: (1) $s \in R(V)$, and (2) if $x \in R(V)$, $b_j \in B$ and $x + b_j \geq 0$, then $x + b_j \in R(V)$. By considering a subset of $\beta \geq 1$ components as the output places, we can generate a set of vectors of β components by means of a VAS as follows. The VAS is started in the initial configuration. At each computation step the VAS, being in a configuration described by a vector $x \in R(V)$, chooses in a nondeterministic way a basis vector $b_j \in B$ such that $x + b_j \geq 0$ and goes to the resulting configuration $x + b_j$. The computation halts when no basis vector b_j satisfies the condition $x + b_j \geq 0$, for the current configuration x. In such a case, the values occurring at the output places of x constitute the output of the computation. Non–halting computations produce no output.

2.3 Fredkin gates and circuits

The *Fredkin gate* is a three–input/three–output boolean gate, whose input/output map $\mathrm{FG} : \{0,1\}^3 \to \{0,1\}^3$ is logically reversible (that is, its inputs can always be deduced from its outputs) and preserves the number of 1's given as input. The map FG associates any input triple $(\alpha_i, \beta_i, \gamma_i)$ with its corresponding output triple $(\alpha_o, \beta_o, \gamma_o)$ according to the following relations: $\alpha_o = \alpha_i$, $\beta_o = (\neg\alpha_i \wedge \beta_i) \vee (\alpha_i \wedge \gamma_i)$, $\gamma_o = (\alpha_i \wedge \beta_i) \vee (\neg\alpha_i \wedge \gamma_i)$ (see the truth table in Figure 1). It is worth noting that the Fredkin gate behaves as a conditional switch, since α_i can be considered as a control line whose value determines whether the input values β_i and γ_i have to be exchanged or not: $\mathrm{FG}(1, \beta_i, \gamma_i) = (1, \gamma_i, \beta_i)$ and $\mathrm{FG}(0, \beta_i, \gamma_i) = (0, \beta_i, \gamma_i)$ for every $\beta_i, \gamma_i \in \{0, 1\}$.

The Fredkin gate is functionally complete for boolean logic: by fixing $\gamma_i = 0$ we obtain $\gamma_o = \alpha_i \wedge \beta_i$, whereas by fixing $\beta_i = 1$ and $\gamma_i = 0$ we obtain $\beta_o = \neg\alpha_i$. By inspecting the truth table, we can see that the Fredkin gate is also logically reversible, since the map FG is a bijection on $\{0,1\}^3$. Moreover, it is conservative: for every input/output pair the number of 1's in the input triple is the same as the



α_i β_i γ_i	$\mapsto$	α_o β_o γ_o
0 0 0		0 0 0
0 0 1		0 0 1
0 1 0		0 1 0
0 1 1		0 1 1
1 0 0		1 0 0
1 0 1		1 1 0
1 1 0		1 0 1
1 1 1		1 1 1

Fig. 1. The Fredkin gate: its behavior as a conditional switch (left) and its truth table (right)

number of 1's in the output triple. In other words, the output triple is obtained by applying an appropriate (input–dependent) permutation to the input triple.

The Fredkin gate is the basis of the model of conservative logic introduced in [2], which describes computations by considering some notable properties of microdynamical laws of physics, such as reversibility and the conservation of the internal energy of the physical system by which computations are performed. Within that model, computations are performed by reversible *Fredkin circuits*, which are acyclic and connected directed graphs made up of *layers* of Fredkin gates. Figure 2 depicts an example of Fredkin circuit having three gates arranged in two layers. The evaluation of a Fredkin circuit in topological order (i.e. layer by layer) defines

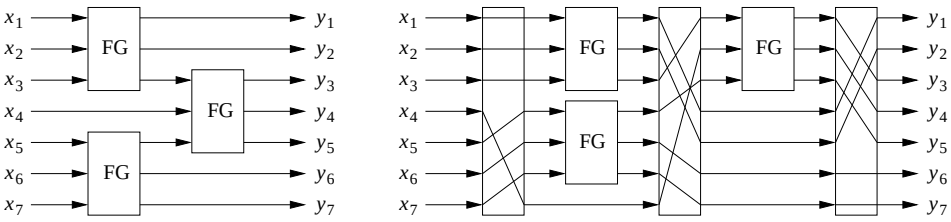


Fig. 2. A reversible Fredkin circuit (on the left) and its normalized version

the boolean function computed by the circuit, which is obtained as the composition of the functions computed by each layer. The conservativeness of the circuit (preservation of the number of 1's) is equivalent to the requirement that the output n-tuple is obtained by applying an appropriate (input–dependent) permutation to the corresponding input n-tuple.

A *reversible* n–input Fredkin circuit is a Fredkin circuit FC_n which computes a bijective map $f_{FC_n} : \{0,1\}^n \to \{0,1\}^n$. Note that the function computed by a reversible Fredkin circuit is also conservative: in fact, every layer of FC_n is

composed by Fredkin gates, which are conservative, and by wires, which obviously preserve the number of 1's given as input.

3 Energy–based P systems

In this section we consider *energy–based P systems* [11, 10], a model of membrane systems whose computations occur by manipulating the energy associated to the objects, as well as the free energy units occurring inside the regions of the system. These energy units can be used to transform objects, using appropriate rules, which are defined according to conservativeness considerations. Formally, an energy–based P system of degree $m \geq 1$, as defined in [10], is a construct $\Pi = (A, \varepsilon, \mu, e, w_1, \ldots, w_m, R_1, \ldots, R_m, i_{\text{in}}, i_{\text{out}})$ where:

- A is an alphabet; its elements are called *objects*;
- $\varepsilon : A \to \mathbb{N}$ is a mapping that associates to each object $a \in A$ the value $\varepsilon(a)$ (also denoted by ε_a), which can be viewed as the "energy value of a". If $\varepsilon(a) = \ell$, we also say that object a *embeds* ℓ units of energy;
- μ is a hierarchical membrane structure consisting of m membranes, each labelled in a unique way with a number in the set $\{1, \ldots, m\}$;
- $e \notin A$ is a special symbol that denotes one *free energy* unit, that is, one unit of energy which is not embedded into any object;
- w_i, with $i \in \{1, \ldots, m\}$, specifies the multiset (over $A \cup \{e\}$) of objects initially present in region i. In what follows we will sometimes assume that the number of e's in some regions of the system is unbounded. In any case, the number of objects from A will always be bounded;
- R_i, with $i \in \{1, \ldots, m\}$, is a finite set of multiset rewriting rules over $A \cup \{e\}$ associated with region i. Rules can be of the following types:

$$ae^k \to (b, p) \ , \qquad a \to (b, p)e^k \ , \qquad e \to (e, p) \ , \qquad a \to (b, p)$$

  where $a, b \in A$, $p \in \{\text{here}, \text{in}(name), \text{out}\}$ and k is a non negative integer. Rules satisfy the *conservativeness condition*, whereby the sum of all (free and embedded) energy values appearing in the left hand side of each rule equals the sum of all (free and embedded) energy values in the corresponding right hand side;
- i_{in} is an integer between 1 and m and specifies the input membrane of Π;
- i_{out} is an integer between 0 and m and specifies the output membrane of Π. If $i_{\text{out}} = 0$ then the environment is used for the output, that is, the output value is the multiset of objects over $A \cup \{e\}$ ejected from the skin.

When a rule of the type $ae^k \to (b, p)$ is applied, the object a, in presence of k free energy units, is allowed to be transformed into object b (note that $\varepsilon_a + k = \varepsilon_b$, for the conservativeness condition). If $p = \text{here}$, then the new object b remains in the same region; if $p = \text{out}$, then b exits from the current membrane. Finally, if $p = \text{in}(name)$, then b enters into the membrane labelled with $name$, which must

be directly contained inside the current membrane in the membrane hierarchy. The meaning of rule $a \to (b, p)e^k$, where k is a positive integer number, is similar: the object a is allowed to be transformed into object b by releasing k units of free energy (here, $\varepsilon_a = \varepsilon_b + k$). As above, the new object b may optionally move one level up or down into the membrane structure. The k free energy units might then be used by another rule to produce "more energetic" objects from "less energetic" ones. When $k = 0$ the rule $ae^k \to (b, p)$, also written as $a \to (b, p)$, transforms the object a into the object b (note that in this case $\varepsilon_b = \varepsilon_a$) and moves it (if $p \neq$ here) upward or downward into the membrane hierarchy, without acquiring or releasing any free energy unit. Analogously, rules $e \to (e, p)$ simply move (if $p \neq$ here) one unit of free energy upward or downward into the membrane structure.

An important observation concerns the application of rules. In the original definition of energy–based P systems, given in [11], the rules were applied in the maximally parallel way, as it usually happens in membrane systems. In the next section we will assume instead that the rules are applied in the *sequential* manner: at each computation step (a global clock is assumed), exactly one among the enabled rules is nondeterministically chosen and applied in the system. We will return to the maximally parallel mode of application in the subsequent section, where we will simulate Fredkin gates and circuits.

A configuration of Π is the tuple $(M_1, \ldots, M_m)$ of multisets (over $A \cup \{e\}$) of objects contained in each region of the system; $(w_1, \ldots, w_m)$ is the *initial* configuration. A configuration where no rule can be further applied is said to be *final*. A computation is a sequence of transitions between configurations of Π, starting from the initial one. A computation is *successful* if and only if it reaches a final configuration or, in other words, it *halts*. The multiset $w_{i_{\text{in}}}$ of objects occurring inside the input membrane is the *input* for the computation, whereas the multiset of objects occurring inside the output membrane (or ejected from the skin, if $i_{\text{out}} = 0$) in the final configuration is the *output* of the computation. A non–halting computation produces no output. As an alternative, we can consider the Parikh vectors associated with the multisets, and see energy–based P systems as computing devices that transform (input) Parikh vectors to (output) Parikh vectors. Optionally, we can disregard the number of free energy units that occur in the input and in the output region of the system, when defining the input and the output multisets (or Parikh vectors).

Since energy is an additive quantity, it is natural to define the *energy of a multiset* as the sum of the amounts of energy associated to each instance of the objects which occur into the multiset. Similarly, the energy of a configuration is the sum of the amounts of energy associated to each multiset which occurs into the configuration. A *conservative computation* is a computation where each configuration has the same amount of energy. A *conservative energy–based P system* is an energy–based P system that performs only conservative computations.

In what follows we will sometimes consider a slightly modified version of energy–based P systems as defined above, in which there are $\alpha \geq 1$ input membranes and $\beta \geq 1$ output membranes. As it will become clear in the following,

this modification does not increase the computational power of energy–based P systems; this is due to the fact that, for any fixed value of $\alpha \geq 1$ (resp., $\beta \geq 1$), the set $\mathbb{N}^\alpha$ (resp., $\mathbb{N}^\beta$) is isomorphic to $\mathbb{N}$, as it is easily shown by using the Cantor mapping. Sometimes we will also use energy–based P systems as generating devices: we will disregard the input membrane, and will consider the multisets (or Parikh vectors) produced in the output membrane at the end of the (halting) nondeterministic computations of the system.

3.1 Computational power

In this section we recall some results, taken from [8], concerning the computational power of energy–based P systems.

Let Π be an energy–based P system as formally defined above. First of all we observe that if we assume that the number of free energy units is *bounded* in each region of Π, then only a finite number of distinct configurations can be obtained, starting from the initial configuration. In fact, each object of Π can only be transformed into another object (it can never be created or destroyed), and possibly moved to another region, according to the rules listed in the definition of the system. In the "worst" case, every object can be transformed into any other object, and can be sent to any region of Π; however, also in this case the number of possible combinations is finite, and thus we obtain a finite number of configurations. By associating a state to each possible configuration of Π, it is not difficult to see that bounded energy–based P systems can be simulated by finite state automata: an arc of the state diagram connects two vertices u and v if and only if the configuration of Π that corresponds to v can be obtained in one step (that is, by applying one rule) from the configuration that corresponds to u.

In order to compare the computational power of energy–based P systems with that of Turing machines, from now on we assume that, in the initial configuration, some regions of the system contain an unlimited number of free energy units. Moreover, we define the following *local* priorities associated to the rules of the system: in each region, if two or more rules can be applied at a given computation step, then one of the rules that manipulate the *maximum* amount of free energy units is nondeterministically chosen and applied. Clearly, even if we impose this policy on energy–based P systems that have a bounded amount of free energy units in each region, we cannot go beyond the computational power of finite state automata.

Assuming an infinite amount of free energy units in the initial configuration, energy–based P systems with priorities assigned to the rules are universal, as stated in the following theorem.

Theorem 1. *Every partial recursive function $f : \mathbb{N}^\alpha \to \mathbb{N}^\beta$ can be computed by an energy–based P system with an infinite supply of free energy units and priorities assigned to rules, with (at most)* $\max\{\alpha, \beta\} + 3$ *membranes.*

Proof. We prove this proposition by simulating deterministic register machines. Let $M = (n, P, m)$ be a deterministic n–register machine that computes f. Observe that, according to Proposition 1, $n = \max\{\alpha, \beta\} + 2$ is enough.

The input values $x_1, \ldots, x_\alpha$ are expected to be in the first α registers of M, and the output values are expected to be in registers 1 to β at the end of a successful computation. Moreover, without loss of generality, we may assume that at the beginning of a computation all registers except (possibly) registers 1 to α contain zero. We construct the energy–based P system $\Pi = (A, \varepsilon, \mu, e, w_s, w_1, \ldots, w_n, R_s, R_1, \ldots, R_n)$ where:

- $A = \{p_j : j \in \{1, 2, \ldots, m\}\} \cup \{\widetilde{p}_j : j \in \{1, 2, \ldots, m-1\}$ and j is the label of an INC instruction$\} \cup \{p'_j : j \in \{1, 2, \ldots, m-1\}$ and j is the label of a DEC instruction$\}$;
- $\varepsilon : A \to \mathbb{N}$ is defined as follows:
  - $\varepsilon(p_j) = 2$ for all $j \in \{1, 2, \ldots, m\}$;
  - $\varepsilon(\widetilde{p}_j) = 1$ for all $j \in \{1, 2, \ldots, m-1\}$ such that j is the label of an INC instruction;
  - $\varepsilon(p'_j) = 3$ for all $j \in \{1, 2, \ldots, m-1\}$ such that j is the label of a DEC instruction;
- $\mu = [_s[_1]_1 \cdots [_\alpha]_\alpha \cdots [_n]_n]_s$ (note that label s denotes the skin membrane);
- $w_s = \{p_1\}$, plus an infinite supply of free energy units;
- $w_i = \begin{cases} \{e^{x_i}\} & \text{if } 1 \leq i \leq \alpha \\ \emptyset & \text{if } \alpha + 1 \leq i \leq n \end{cases}$
- $R_s = \{p_j \to (p_j, in(r)) : j \in \{1, 2, \ldots, m-1\}$ and the j-th instruction of P operates on register $r\} \cup \{\widetilde{p}_j e \to (p_\ell, here) : j \in \{1, 2, \ldots, m-1\}$ and j is the label of an INC instruction that jumps to label $\ell\} \cup \{p'_j \to (p_{\ell_1}, here)e : j \in \{1, 2, \ldots, m-1\}$ and j is the label of a DEC instruction whose first jump label is $\ell_1\}$;
- $R_i = \{p_j \to (\widetilde{p}_j, out)e : j \in \{1, 2, \ldots, m-1\}$ and j is the label of an INC instruction that affects register $i\} \cup \{p_j e \to (p'_j, out) : j \in \{1, 2, \ldots, m-1\}$ and j is the label of a DEC instruction that affects register $i\} \cup \{p_j \to (p_{\ell_2}, out) : j \in \{1, 2, \ldots, m-1\}$ and j is the label of a DEC instruction that affects register i and whose second jump label is $\ell_2\}$, for all $i \in \{1, 2, \ldots, n\}$.

Informally, the system is composed of the skin membrane, that contains one elementary membrane for each register of M. At each moment during the computation, the value r_i contained in register i, $1 \leq i \leq n$, is represented by the number of free energy units contained in the i-th elementary membrane. Hence, the elementary membranes from 1 to α contain the input at the beginning of the computation, whereas the elementary membranes from 1 to β contain the output if and when the computation halts. The region enclosed by the skin contains one object of the kind p_j, $j \in \{1, 2, \ldots, m\}$, which represents the value j (that is, the instruction labelled with j) of the program counter of M. To simulate the instruction $j : (INC(r), \ell)$, the object p_j enters into the region r thanks to the rule $p_j \to (p_j, in(r))$. In this region, p_j is transformed into $\widetilde{p}_j$ by means of the

rule $p_j \rightarrow (\widetilde{p}_j, out)e$, thus releasing one free energy unit, while the resulting object $\widetilde{p}_j$ is sent back to the region enclosed by the skin. There, a rule of the kind $\widetilde{p}_j e \rightarrow (p_\ell, here)$ produces the object which represents the label of the next instruction to be executed. As we can see, the application of this rule requires the presence of a free energy unit in the region enclosed by the skin.

To simulate the instruction $j : (DEC(r), \ell_1, \ell_2)$, the object p_j, which occurs in the region enclosed by the skin, enters into region r by means of the rule $p_j \rightarrow (p_j, in(r))$. Assuming that there is at least one free energy unit inside region r, the object p_j can be transformed into p'_j thanks to the rule $p_j e \rightarrow (p'_j, out)$. One free energy unit is thus consumed in region r, and the resulting object is sent back to the region enclosed by the skin. There, it is transformed into p_{ℓ_1} thanks to the rule $p'_j \rightarrow (p_{\ell_1}, here)e$, by releasing one unit of free energy. On the other hand, if membrane r does not contain free energy units (and *only* in this case) then object p_j – just arrived from the region enclosed by the skin – is transformed into p_{ℓ_2} by means of the rule: $p_j \rightarrow (p_{\ell_2}, out)$. In this case no free energy units are involved in the transformation, and the resulting object is immediately sent to the region enclosed by the skin. Note that the correct simulation of the DEC instruction is guaranteed by the priorities associated with the rules: when object p_j enters into membrane r, then the rule $p_j e \rightarrow (p'_j, out)$ has priority over the rule $p_j \rightarrow (p_{\ell_2}, out)$, since it manipulates more free energy units than the other.

The halt instruction is simply simulated by doing nothing with the object p_m when it appears in region s. It is apparent from the description given above that, after the simulation of each instruction, the number of free energy units contained into membrane i equals the value contained in register i, with $1 \leq i \leq n$. Hence, when the halting symbol p_m appears in region s, the contents of membranes 1 to β equal the output of the program P. □

The following corollary is an immediate consequence of Theorem 1, by taking $\beta = 0$.

Corollary 1. *Let $L \subseteq \mathbb{N}^\alpha$, $\alpha \geq 1$, be a recursively enumerable set of (vectors of) non–negative integers. Then L can be accepted by an energy–based P system with an infinite supply of free energy units and priorities assigned to rules, with (at most) $\alpha + 3$ membranes.*

For the generating case we have to simulate *nondeterministic* register machines, which are defined exactly as the deterministic version, the only difference being in the INC instruction, that now has the form $j : (INC(r), k, \ell)$; when executing this instruction, after incrementing register r, the computation continues nondeterministically either with the instruction labelled by k or with the instruction labelled by ℓ. The necessary changes in the above simulation are obvious, and hence are here omitted. Under this setting, the following corollary is also an immediate consequence of Theorem 1, by taking $\alpha = 0$.

Corollary 2. *Let $L \subseteq \mathbb{N}^\beta$, $\beta \geq 1$, be a recursively enumerable set of (vectors of) non–negative integers. Then L can be generated by an energy–based P system with*

an infinite supply of free energy units and priorities assigned to rules, with (at most) $\beta + 3$ membranes.

On the other hand, if we assume that an infinite amount of free energy units occurs in the initial configuration but no priorities are assigned to the rules, then energy–based P systems are *not* universal, as proved in the following theorem.

Theorem 2. *Energy–based P systems with an infinite supply of free energy units, and* without *priorities assigned to the rules, can be simulated by vector addition systems.*

Proof. Let Π be an energy–based P system that contains an infinite supply of free energy units in its initial configuration. Denoted by m the degree of Π, by n the cardinality of the alphabet A, and by R the total number of rules in Π, we define a vector addition system $V = (B, s)$, with $B = \{b_1, b_2, \ldots, b_R\}$, as follows. The vectors $s, b_1, b_2, \ldots, b_R$ have one component for each possible object/region pair (a, i) of Π, that is, for all $a \in A \cup \{e\}$ and $i \in \{1, 2, \ldots, m\}$ (note that here we treat e just like the objects of A). The start vector s reflects the initial configuration of Π: for all $a \in A \cup \{e\}$ and for all $i \in \{1, 2, \ldots, m\}$, the component of s associated with the pair (a, i) is set to the number of copies of a in the i-th region of Π. The only exception is given for those regions of Π where an infinite number of free energy units occur: the corresponding components of s are initialized with E, which is defined as the maximum number of free energy units which are necessary to execute *any* rule of Π (formally, $E = \max\{k \mid ae^k \to (b, p)$ is a rule of $\Pi\}$). So doing, we are able to initialize every component of s with a finite value.

Each rule of the kind $ae^k \to (b, p) \in R_i$ is translated into a basis vector $b_l \in B$, $l \in \{1, \ldots, R\}$, as follows: since one copy of a and k copies of e are removed from region i, the component of b_l that corresponds to the pair (a, i) will be equal to -1, and the component that corresponds to (e, i) will be equal to $-k$. Similarly, denoted by j the region determined by the target p, since one copy of b will be sent to region j, the corresponding component of b_l will be equal to 1. Rules of the kind $a \to (b, p)e^k$, as well as rules of the kind $a \to (b, p)$ and $e \to (e, p)$, are translated into appropriate basis vectors in a similar way. An important observation is that each component of the basis vectors that corresponds to a pair (e, i), such that region i of Π contains an infinite supply of free energy units in its initial configuration, is set equal to E. So doing, at each computation step E copies of e are added to those components of the VAS which correspond to the regions of Π that contain an infinite amount of e. Thus, at the beginning of the next computation step, such components have a value which is finite but sufficiently high to simulate any rule of Π.

It is clear that any feasible sequential computation of Π corresponds to a sequence of applications of basis vectors of V, and that for each pair (a, i), with $a \in A \cup \{e\}$ and $i \in \{1, 2, \ldots, m\}$, the number of copies of object a in the region i of Π after the application of a rule matches the value of the component of the state vector that corresponds to (a, i), with the exception of the pairs (e, i) for those regions i of Π that contain an infinite number of free energy units in the initial

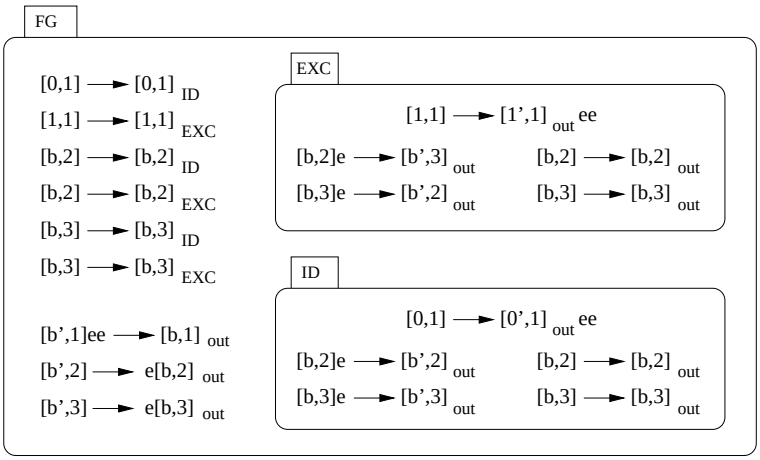


Fig. 3. An energy–based P system which simulates the Fredkin gate

configuration. However, any multiset (or its corresponding Parikh set) generated by Π can also be generated by V by means of the above simulation. □

3.2 Simulating the Fredkin gate

Let us now describe an energy–based P system which simulates the Fredkin gate. The results contained in this section are taken from [11, 10]; as stated above, we switch to the maximally parallel mode of applying the rules.

The system, illustrated in Figure 3, is defined as follows. The alphabet contains 12 kinds of objects. For the sake of clarity, we denote these objects by $[b, j]$ and $[c, j]$, with $b \in \{0, 1\}$, $c \in \{0', 1'\}$ and $j \in \{1, 2, 3\}$. Intuitively, $[b, j]$ and $[c, j]$ indicate the boolean value which occurs in the j-th line of the Fredkin gate. It will be clear from the simulation that we need two different symbols to represent each of these boolean values. Every object of the kind $[b, j]$, with $b \in \{0, 1\}$ and $j \in \{1, 2, 3\}$, has energy equal to 3, whereas the objects $[c, 1]$ have energy equal to 1 and the objects $[c, 2]$ and $[c, 3]$ (with $c \in \{0', 1'\}$) have energies equal to 4.

The simulation works as follows. The input values $[x_1, 1], [x_2, 2], [x_3, 3]$, with $x_1, x_2, x_3 \in \{0, 1\}$, are injected into the skin. If $x_1 = 0$ then the object $[0, 1]$ enters into membrane ID, where it is transformed to the object $[0', 1]$ by releasing 2 units of energy. The object $[0', 1]$ leaves membrane ID and waits for 2 energy units to transform back to $[0, 1]$ and leave the system. The objects $[x_2, 2]$ and $[x_3, 3]$, with $x_2, x_3 \in \{0, 1\}$, may enter nondeterministically either into membrane ID or into membrane EXC; however, if they enter into EXC they cannot be transformed to $[x'_2, 3]$ and $[x'_3, 2]$ since in EXC there are no free energy units. Thus the only possibility for objects $[x_2, 2]$ and $[x_3, 3]$ is to leave EXC and choose again between membranes ID and EXC in a nondeterministic way. Eventually, after some time they

enter (one at the time or simultaneously) into membrane ID. Here they have the possibility to be transformed into $[x'_2, 2]$ and $[x'_3, 3]$ respectively, using the 2 units of free energy which occur into the region enclosed by ID (alternatively, they have the possibility to leave ID and choose nondeterministically between membranes ID and EXC once again). When the objects $[x'_2, 2]$ and $[x'_3, 3]$ are produced they immediately leave ID, and are only allowed to transform back to $[x_2, 2]$ and $[x_3, 3]$ respectively, releasing 2 units of energy. The objects $[x_2, 2]$ and $[x_3, 3]$ just produced leave the system, and the 2 units of energy can only be used to transform $[0', 1]$ back to $[0, 1]$ and expel it from the skin.

On the other hand, if $x_1 = 1$ then the object $[1, 1]$ enters into membrane EXC where it is transformed into the object $[1', 1]$ by releasing 2 units of energy. The object $[1', 1]$ leaves membrane EXC and waits for 2 energy units to transform back to $[1, 1]$ and leave the system. Once again the objects $[x_2, 2]$ and $[x_3, 3]$, with $x_2, x_3 \in \{0, 1\}$, may choose nondeterministically to enter either into membrane ID or into membrane EXC. If they enter into ID they can only exit again since in ID there are no free energy units. When they enter into EXC they can be transformed to $[x'_2, 3]$ and $[x'_3, 2]$ respectively, using the 2 free energy units which occur into the region, and leave EXC. Now objects $[x'_2, 3]$ and $[x'_3, 2]$ can only be transformed into $[x_2, 3]$ and $[x_3, 2]$ respectively, and leave the system. During this transformation 2 free energy units are produced; these can only be used to transform $[1', 1]$ back to $[1, 1]$, which leaves the system.

It is apparent from the simulation that the system can be defined to work on any triple of lines of a circuit, by simply modifying the values of the second component of the objects manipulated by the system.

The proposed P system is conservative: the number of energy units present into the system (both free and embedded into objects) during computations is constantly equal to 9. At the end of the computation, all these energy units are embedded into the output values. The system is also reversible: it is immediately seen that if we inject into the skin the output triple just produced as the result of a computation, the system will expel the corresponding input triple. This behavior is trivially due to the fact that the Fredkin gate is *self–reversible*, meaning that $\text{FG} \circ \text{FG} = \text{ID}_3$ (equivalently, $\text{FG} = \text{FG}^{-1}$), where ID_3 is the identity function on $\{0, 1\}^3$. Notice that, in general, this property does not hold for the functions $f : \{0, 1\}^n \to \{0, 1\}^n$ computed by n–input reversible Fredkin circuits. This means that in general the P system that simulates a given Fredkin circuit must be appropriately designed in order to be self–reversible.

3.3 Simulation of reversible Fredkin circuits

Basing upon the simulation of the Fredkin gate we have exposed in the previous section, in [10] we have shown that any reversible Fredkin circuit can be simulated by an appropriate energy–based P system. Since the construction is quite involved, in what follows we just give a few details.

Let FC_n be an n–input reversible Fredkin circuit of depth d, and let $L_1, L_2, \ldots, L_d$ denote the layers of FC_n. As we can see on the left side of Figure 2, each

layer is composed by some number of Fredkin gates and some non–intersecting wires. Let k_j, with $j \in \{1, 2, \ldots, d\}$, be the number of Fredkin gates occurring in layer L_j. First of all we define the P systems $G_{j,i}$, for $j \in \{1, 2, \ldots, d\}$ and $i \in \{1, 2, \ldots, k_j\}$, by modifying the P system FG exposed in the previous section as follows. The objects of $G_{j,i}$ are denoted by $[b, \ell, j]$ and $[c, \ell, j]$, with $b \in \{0, 1\}$, $c \in \{0', 1'\}$, $\ell \in \{\ell_1, \ell_2, \ell_3\} \subseteq \{1, 2, \ldots, n\}$ such that $\ell_1 \neq \ell_2 \neq \ell_3$, and $j \in \{1, 2, \ldots, d\}$. Intuitively, $G_{j,i}$ simulates the i-th Fredkin gate occurring in layer L_j of FC_n, and $[b, \ell, j]$, $[c, \ell, j]$ indicate the boolean value which occurs in the ℓ-th line of L_j. The values ℓ_1, ℓ_2 and ℓ_3 correspond to the three lines of the circuit upon which the Fredkin gate operates. The objects $[b, \ell, j]$ have energy equal to 3, whereas the energy of objects $[c, \ell_1, j]$ is 1 and the energy of $[c, \ell_2, j]$ and $[c, \ell_3, j]$ is equal to 4. The system $G_{j,i}$ processes the objects $[b, \ell, j]$ given as input exactly as FC would process the corresponding objects $[b, \ell]$, with the only difference that, when it expels the results of the computation in its enviroment, it changes objects $[b, \ell, j]$ to $[b, \ell, j + 1]$. This is done in order to indicate that the simulation of FC_n can continue with the next layer.

We can now build an energy–based P system P_n which simulates FC_n as follows. To simplify the exposition, we will consider the P systems $G_{j,i}$ defined above as black boxes that, when fed with input values (represented as appropriate objects), after some time produce their results. The objects of P_n are denoted by $[b, i, j]$, with $b \in \{0, 1\}$, $i \in \{1, 2, \ldots, n\}$ and $j \in \{1, 2, \ldots, d+1\}$. The energy of all these objects is equal to 3. As before, $[b, i, j]$ indicates the presence of the boolean value b on the i-th input line of the j-th layer of FC_n. Note that some of these objects are also used in subsystems $G_{j,i}$. The system P_n, illustrated in Figure 4, is composed by a main membrane (the skin) that contains a subsystem F_j for each layer L_j of FC_n. Every subsystem F_j simulates the corresponding layer L_j of the circuit, using the subsystems $G_{j,1}, G_{j,2}, \ldots, G_{j,k_j}$ to simulate the Fredkin gates which occur in L_j. The region associated to the skin membrane contains the rules:

$$[b, i, j] \rightarrow [b, i, j]_{F_j} \tag{1}$$

and the rules:

$$[b, i, d + 1] \rightarrow [b, i, d + 1]_{out} \tag{2}$$

for every $b \in \{0, 1\}$, $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, d\}$. The application of rules (1) makes the objects representing the boolean values occurring in the i-th input line of layer L_j move into subsystem F_j, whereas rules (2) expel the result of the simulation to the environment. The region associated to membrane F_j, for $j \in \{1, 2, \ldots, d\}$, contains the rules:

$$[b, i, j] \rightarrow [b, i, j]_{G_{j,r_i}} \tag{3}$$

where $r_i \in \{1, 2, \ldots, k_j\}$ is the number of the Fredkin gate in L_j which has i as an input line, as well as the rules:

$$[b, i, j + 1] \rightarrow [b, i, j + 1]_{out} \tag{4}$$

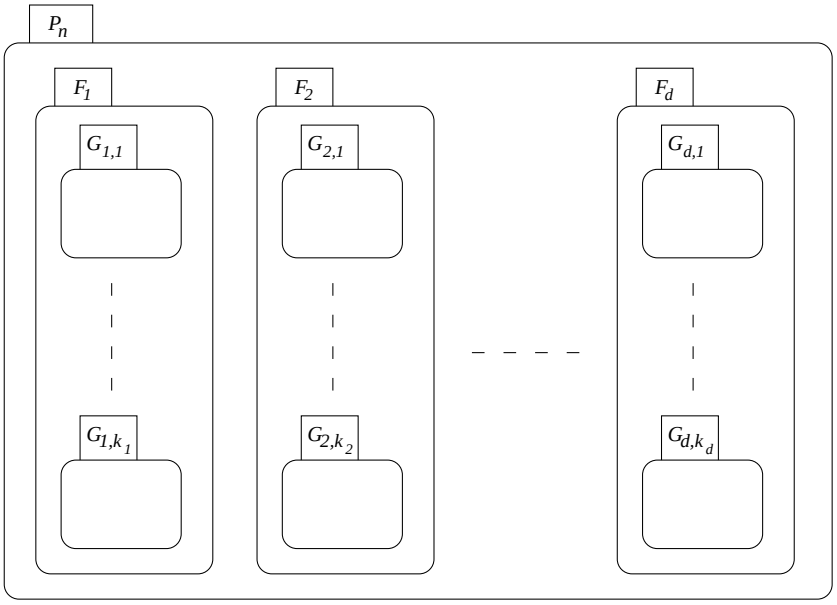


Fig. 4. Structure of the P system P_n which simulates an n–input reversible Fredkin circuit FC_n. Every subsystem F_j simulates the corresponding layer L_j of FC_n, whereas the subsystems $G_{j,i}$ simulate the Fredkin gates occurring in L_j

which expel the results towards the skin membrane when they appear. For all the objects $[b, i, j]$ which have not to be processed by a Fredkin gate (since the i-th line of L_j is a wire) the region enclosed by membrane F_j contains the rules:

$$[b, i, j] \rightarrow [b, i, j + 1]_{out} \tag{5}$$

Hence, the simulation of FC_n works as follows. At the beginning of the computation the objects $[x_1, 1, 1], [x_2, 2, 1], \ldots, [x_n, n, 1]$, representing the input n-tuple $(x_1, x_2, \ldots, x_n)$ of FC_n, are injected into the skin. The application of rules (1) makes these objects move into subsystem F_1. If a given object $[b, i, 1]$ hasn't to be processed by a Fredkin gate (since the i-th line of L_1 is a wire) then the corresponding rule from (5) expels the object $[b, i, 2]$ to the region enclosed by membrane F_1. On the other hand, using rules (3), the objects $[b, i, 1]$ that must be processed by a Fredkin gate are dispatched to the correct subsystems G_{1,r_i}. Eventually, after some time the objects corresponding to the result of the computation performed by each gate of L_1 leave the corresponding systems $G_{1,1}, G_{1,2}, \ldots, G_{1,k_1}$, with the third component incremented by 1. These objects are expelled from F_1 using rules (4). As objects $[b, i, 2]$ are expelled from F_1, rules (1) dispatch them to subsystem F_2. The simulation of FC_n continues in this way until the objects $[b, i, d + 1]$ leave the subsystem F_d. Here they activate rules (2), that expel them into the environment as the result of the computation performed by P_n.

The formal definition of P_n can be found in [10]. Let us note that the system is conservative, since the amount of energy units present into the system (both free and embedded into objects) during computations is constantly equal to $3n$. The number of rules and the number of membranes in the system are directly proportional to the number of gates in FC_n. Differently from the other approaches seen in literature, the depth of hierarchy μ in system P_n is constant; in particular, it does not depend upon the number of gates occurring in FC_n.

Reverse computations

If a Fredkin circuit FC_n is reversible, then there exists a Fredkin circuit FC'_n which computes the inverse function $f^{-1}_{FC_n} : \{0,1\}^n \to \{0,1\}^n$. This circuit can be easily obtained from FC_n by reversing the order of all layers. Actually, in [10] we have shown that the P system P_n that simulates FC_n can be modified in order to become *self–reversible*, that is, able to compute both f_{FC_n} and $f^{-1}_{FC_n}$. To this aim, we add a further component $k \in \{0,1\}$ to the objects of P_n, which is used to distinguish between "forward" and "backward" computations. Precisely, the objects which are used to compute f_{FC_n} have $k = 0$, and those used to compute $f^{-1}_{FC_n}$ have $k = 1$. A forward computation starts by injecting the objects $[x_1, 1, 1, 0], [x_2, 2, 1, 0], \ldots, [x_n, n, 1, 0]$ into the skin of P_n. The computation proceeds as described above, with the rules modified in order to consider the presence of the new component $k = 0$. The objects produced in output are $[y_1, 1, d+1, 0], \ldots, [y_n, n, d+1, 0]$, where $(y_1, \ldots, y_n) = f_{FC_n}(x_1, \ldots, x_n)$.

Analogously, a "backward" computation should start by injecting the objects $[y_1, 1, 1, 1], [y_2, 2, 1, 1], \ldots, [y_n, n, 1, 1]$ into the skin. The computation of $f^{-1}_{FC_n}$ can be accomplished by incorporating the rules of the region enclosed by the skin and the subsystems of P'_n (both modified in order to take into account the presence of the new component $k = 1$) into P_n. Interferences between the rules concerning forward and backward computations do not occur since they act on different kinds of objects.

A further improvement is obtained by observing that each layer of FC_n is self–reversible, and that the layers of FC'_n are the same as the layers of FC_n, in reverse order. Hence we can merge each subsystem F_j, which simulates layer L_j of FC_n, with the subsystem F'_{d-j+1}, which simulates layer L'_{d-j+1} of FC'_n. The merge operation consists in putting the rules and the subsystems of F'_{d-j+1} into F_j. Of course we have also to modify the rules in the region enclosed by the skin so that the objects that were previously moved to F'_{d-j+1} are now dispatched to F_j. Recursively, since each Fredkin gate is self–reversible, we can merge also subsystems $G_{j,1}, \ldots, G_{j,k_j}$ occurring into F_j with the corresponding subsystems $G'_{d-j+1,1}, \ldots, G'_{d-j+1,k_j}$ which occur into F'_{d-j+1}. In this way, we obtain a self–reversible P system which is able to compute both f_{FC_n} and $f^{-1}_{FC_n}$. The new system has the same number of membranes as P_n, and the double of rules.

Reducing the number of subsystems

As we have seen in the previous sections, the number of membranes and the number of rules of the P system P_n that simulates the reversible Fredkin circuit FC_n grow linearly *with respect to the number of gates occurring in the circuit*. Actually, the number of membranes in P_n can be made linear with respect to n, *independently of the number of gates* occurring in the simulated Fredkin circuit FC_n. To compensate the reduced number of membranes, the number of rules in the system will grow accordingly. For the sake of simplicity, let us consider only forward computations, involving objects of the kind $[b, i, j]$, with $b \in \{0, 1\}$, $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, d + 1\}$.

First of all, every n–input reversible Fredkin circuit FC_n can be "normalized" by moving the Fredkin gates contained into each layer as upward as possible, as illustrated on the right side of Figure 2. The resulting layers are called *normalized* layers. In order to keep track of which input value goes into which gate, we precede each normalized layer by a fixed (that is, non input–dependent) permutation, which is realized by rearranging the wires as required. A final fixed permutation, occurring after the last normalized layer, allows the output values of FC_n to appear on the correct output lines. Observe that the number of possible n–input normalized layers of Fredkin gates is $\lfloor \frac{n}{3} \rfloor$. We can thus number all possible normalized layers with an index $\ell \in \{1, \ldots, \lfloor \frac{n}{3} \rfloor\}$, and describe a normalized Fredkin circuit by a sequence of indexes $\ell_1, \ell_2, \ldots, \ell_d$ together with a corresponding sequence of fixed permutations $\pi_1, \pi_2, \ldots, \pi_{d+1}$.

The normalization of every layer L_j of FC_n can be performed in linear time with respect to n, as described in [10]. The time needed to normalize the entire circuit is thus bounded by $O(n \cdot d)$, the size of the circuit.

An energy–based P system that simulates a normalized Fredkin circuit can be built by composing (at most) the $\lfloor \frac{n}{3} \rfloor$ subsystems $F_1, \ldots, F_{\lfloor n/3 \rfloor}$, each one capable to simulate a fixed normalized layer of Fredkin gates. The region enclosed by the skin contains the rules $[b, i, j] \to [b, \pi_j(i), j]_{F_{\ell_j}}$ for all $b \in \{0, 1\}$, $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, d\}$, as well as the rules $[b, i, d + 1] \to [b, \pi_{d+1}(i), d + 1]_{out}$. These rules implement the fixed permutations, move the objects to the subsystem that simulates the next normalized layer, and expel the results of the computation into the environment. The simulation of each normalized layer is analogous to the simulation of the layers of a non–normalized Fredkin circuit, as described above. Note that the objects emerge from subsystems $F_1, \ldots, F_{\lfloor n/3 \rfloor}$ with the j component incremented by 1, so that they are ready for the next computation step. If the same normalized layer occurs in two or more positions in the normalized Fredkin circuit, then the corresponding subsystem must contain the rules which allow to process all the objects which appear in these positions.

A further transformation of the Fredkin circuit allows to perform the simulation with just *one* subsystem. Starting from a normalized n–input Fredkin circuit NFC_n, we transform each normalized layer so that in the resulting circuit every layer contains the same number of gates. Figure 5 shows the result of this transformation, applied to the normalized Fredkin circuit illustrated in Figure 2.

Informally, the transformation is performed as follows. Considering one normal-

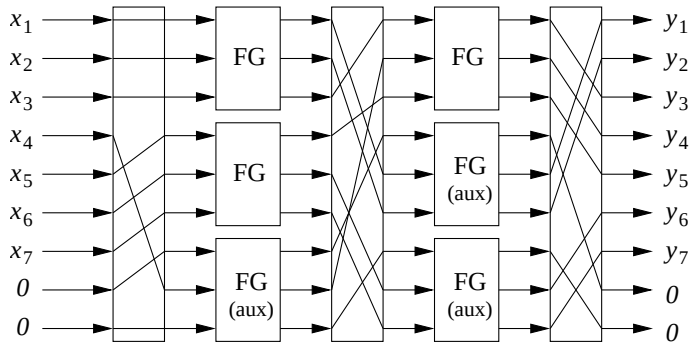


Fig. 5. A normalized Fredkin circuit with auxiliary lines and gates. The number of gates is the same in each layer

ized layer at a time, we first add a number of auxiliary lines, fed with the boolean constant 0. The number of auxiliary lines added depends upon the number of *free* lines (that is, lines not affected by any gate) in the given layer. As a result, the total number of lines is a multiple of 3. We can thus add an appropriate number of auxiliary Fredkin gates (denoted by "FG (aux)" in Figure 5) to the layer, each one taking an auxiliary line as its first input, so that every auxiliary gate computes the identity function. At the end of this process, we add (if needed) to each layer further auxiliary lines, in order to obtain the same number of input/output lines for all the layers. Since the auxiliary lines have been added at the bottom of the circuit, we have to permute them together with the original free lines to feed them correctly to the transformed layer. The details can be found in [10]. The energy–based P system that simulates a transformed Fredkin circuit is the same as described in the previous section, but now it contains only the subsystem which simulates a full layer of Fredkin gates. If desired, also the membrane which encloses such subsystem can be removed, thus lowering the depth of the membrane hierarchy by 1. The new system has again $\lfloor n/3 \rfloor$ subsystems, each one simulating a Fredkin gate. Of course, the rules in the skin must be modified so that they dispatch the objects directly to the correct subsystem.

4 UREM P Systems

Let us now consider *UREM P systems* [4], that is, P systems with unit rules and energy assigned to membranes. A UREM P system of degree $d+1$ is a construct Π of the form $\Pi = (A, \mu, e_0, \ldots, e_d, w_0, \ldots, w_d, R_0, \ldots, R_d)$, where:

- A is an alphabet of *objects*;

- μ is a *membrane structure*, with the membranes labelled by numbers $0, \ldots, d$ in a one-to-one manner;
- $e_0, \ldots, e_d$ are the initial energy values assigned to the membranes $0, \ldots, d$. In what follows we assume that $e_0, \ldots, e_d$ are non–negative integers;
- $w_0, \ldots, w_d$ are multisets over A associated with the regions $0, \ldots, d$ of μ;
- $R_0, \ldots, R_d$ are finite sets of *unit rules* associated with the membranes $0, \ldots, d$. Each rule or R_i has the form $(\alpha_i : a, \Delta e, b)$, where $\alpha \in \{in, out\}$, $a, b \in A$, and $|\Delta e|$ is the amount of energy that — for $\Delta e \geq 0$ — is added to or — for $\Delta e < 0$ — is subtracted from e_i (the energy assigned to membrane i) by the application of the rule.

The *initial configuration* of Π consists of $e_0, \ldots, e_d$ and $w_0, \ldots, w_d$. The transition from a configuration to another one is performed by nondeterministically choosing one rule from some R_i and applying it (hence we consider the *sequential* mode of applying the rules). Applying $(in_i : a, \Delta e, b)$ means that an object a (being in the membrane immediately outside of i) is changed into b while entering membrane i, thereby changing the energy value e_i of membrane i by Δe. On the other hand, the application of a rule $(out_i : a, \Delta e, b)$ changes object a into b while leaving membrane i, and changes the energy value e_i by Δe. The rules can be applied only if the amount e_i of energy assigned to membrane i fulfills the requirement $e_i + \Delta e \geq 0$. Moreover, we use a sort of *local priorities*: if there are two or more applicable rules in membrane i, then one of the rules with $\max |\Delta e|$ has to be used.

A sequence of transitions is called a *computation*; it is *successful* if and only if it halts. The *result* of a successful computation is considered to be the distribution of energies among the membranes in the halting configuration. A non–halting computation does not produce a result. If we consider the energy distribution of the membrane structure as the input to be analysed, we obtain a model for accepting sets of (vectors of) non–negative integers.

4.1 Computational power

The following result, proved in [4], establishes computational completeness for this model of P systems.

Theorem 3. *Every partial recursive function $f : \mathbb{N}^\alpha \to \mathbb{N}^\beta$ ($\alpha \geq 1$, $\beta \geq 1$) can be computed by a UREM P system with (at most)* $\max\{\alpha, \beta\} + 3$ *membranes.*

As in the case of energy–based P systems, the proof of this proposition is obtained by simulating register machines. In the simulation, a P system is defined which contains one subsystem for each register of the simulated machine. The contents of the register are expressed as the energy value e_i assigned to the i-th subsystem. A single object is present in the system at every computation step, which stores the label of the instruction of the program P currently simulated. Increment instructions are simulated in two steps by using the rules $(in_i : p_j, 1, \widetilde{p_j})$ and $(out_i : \widetilde{p_j}, 0, p_k)$. Decrement instructions are also simulated in two steps, by

using the rules $(in_i : p_j, 0, \widetilde{p_j})$ and $(out_i : \widetilde{p_j}, -1, p_k)$ or $(out_i : \widetilde{p_j}, 0, p_l)$. The use of priorities associated to these last rules is crucial to correctly simulate a decrement instruction. For the details of the proof we refer the reader to [4].

When taking $\beta = 0$ in the proof of the above proposition, we get the *accepting* variant of P systems with unit rules and energy assigned to membranes:

Corollary 3. *Let $L \subseteq \mathbb{N}^\alpha$, $\alpha \geq 1$, be a recursively enumerable set of (vectors of) non–negative integers. Then L can be accepted by a UREM P system having (at most) $\alpha + 3$ membranes.*

The above results were obtained by simulating deterministic register machines by means of *deterministic* UREM P systems, where at each step only one rule is enabled and can be applied. As we did with energy–based P systems, for the *generative* case we have to pass to a *nondeterministic* choice of rules, and simulate nondeterministic register machines. Under this setting, the following corollary is also a simple consequence of Theorem 3, by taking $\alpha = 0$. As a technical detail we mention that the nondeterministic INC instruction $j : (INC(i), k, \ell)$ is simulated in two steps by using the rules $(in_i : p_j, 1, \widetilde{p_j})$ and then $(out_i : \widetilde{p_j}, 0, p_k)$ or $(out_i : \widetilde{p_j}, 0, p_\ell)$.

Corollary 4. *Let $L \subseteq \mathbb{N}^\beta$, $\beta \geq 1$, be a recursively enumerable set of (vectors of) non–negative integers. Then L can be generated by a UREM P system having (at most) $\beta + 3$ membranes.*

Once again, when omitting the priority feature we do not get systems with universal computational power. This time, however, we obtain a characterization of the family $PsMAT^\lambda$ of Parikh sets generated by context–free matrix grammars, without occurrence checking and with λ-rules. The proof is quite involved, and hence we refer the reader to [4, 10].

However, even without the priority feature UREM P systems can obtain universal computational power, provided that their rules are applied in the maximally parallel mode instead of the sequential mode:

Theorem 4. *Each partial recursive function $f : \mathbb{N}^\alpha \to \mathbb{N}^\beta$ ($\alpha \geq 1$, $\beta \geq 1$) can be computed by a UREM P system with (at most) $\max\{\alpha, \beta\} + 4$ membranes when working in the maximally parallel mode without priorities on the rules.*

Once again, the proof is obtained by simulating register machines. This time, however, the simulation is more complicated, and requires the use of an auxiliary membrane which is used as a "pacemaker" to drive the correct simulation of INC and DEC instructions. We refer the reader to [10] for the details.

The following results are immediate consequences of Theorem 4 as Corollaries 3 and 4 were immediate consequences of Theorem 3:

Corollary 5. *Let $L \subseteq \mathbb{N}^\alpha$, $\alpha \geq 1$, be a recursively enumerable set of (vectors of) non–negative integers. Then L can be accepted by a UREM P system with (at most) $\alpha + 4$ membranes in the maximally parallel mode without priorities on the rules.*

Corollary 6. *Let $L \subseteq \mathbb{N}^\beta$, $\beta \geq 1$, be a recursively enumerable set of (vectors of) non–negative integers. Then L can be generated by a UREM P system with (at most) $\beta + 4$ membranes in the maximally parallel mode without priorities on the rules.*

5 Conclusions

In this paper we have reviewed some results obtained in the last few years, concerning the computational power of two models of computation defined in the framework of membrane computing: energy–based P systems and UREM P systems. Such models are inspired from the functioning of some physical laws, that consider the computation devices as physical objects that manipulate energy during their computations.

We believe that these P systems have the potential to generate further stimulating research. Two spin–offs of UREM P systems we have not mentioned in this paper are *tissue–like* UREM P systems, whose study has begun in [10], and *quantum–like* UREM P systems, introduced in [9]. A tissue–like version of energy–based P systems is missing, as well as a comparison with other models of P systems that use energy in their computation steps (such as [15, 3, 6]).

References

1. A. Alhazov, R. Freund, A. Leporati, M. Oswald, C. Zandron. (Tissue) P Systems with Unit Rules and Energy Assigned to Membranes. *Fundamenta Informaticae*, **74**:391–408, 2006.
2. E. Fredkin, T. Toffoli. Conservative Logic. *International Journal of Theoretical Physics*, **21**(3-4):219–253, 1982.
3. R. Freund. Energy–Controlled P Systems. In *Membrane Computing, Proceedings of the International Workshop WMC–CdeA 2002*, LNCS 2597, Springer–Verlag, Berlin, 2003, 247–260.
4. R. Freund, A. Leporati, M. Oswald, C. Zandron. Sequential P Systems with Unit Rules and Energy Assigned to Membranes. In *Machines, Computations and Universality (MCU 2004)*, Saint–Petersburg, Russia, September 21–24, 2004, LNCS 3354, Spriger–Verlag, Berlin, 2005, pp. 200–210.
5. R. Freund, M. Oswald. GP Systems with Forbidding Context. *Fundamenta Informaticae*, **49**(1-3):81–102, 2002.
6. P. Frisco. The Conformon–P System: a Molecular and Cell Biology–inspired Computability Model. *Theoretical Computer Science*, **312**:295–319, 2004.
7. R. Karp, R. Miller. Parallel Program Schemata. *Journal of Computer and System Science*, **3**(4):167–195, 1969. Also RC2053, IBM T.J. Watson Research Center, New York, April 1968.
8. A. Leporati, D. Besozzi, P. Cazzaniga, D. Pescini, C. Ferretti. Computing with Energy and Chemical Reactions. *Natural Computing*, to appear.

9. A. Leporati, G. Mauri, C. Zandron. Quantum Sequential P Systems with Unit Rules and Energy Assigned to Membranes. In *Membrane Computing: 6th International Workshop (WMC 2005)*, Vienna, Austria, July 18–21, 2005, LNCS 3850, Springer–Verlag, Berlin, 2006, pp. 310–325.
10. A. Leporati, C. Zandron, G. Mauri. Reversible P Systems to Simulate Fredkin Circuits. *Fundamenta Informaticae*, **74**:529548, 2006.
11. A. Leporati, C. Zandron, G. Mauri. Simulating the Fredkin Gate with Energy-based P Systems. *Journal of Universal Computer Science*, **10**(5):600–619, 2004.
12. M.L. Minsky. *Finite and Infinite Machines*. Prentice Hall, Englewood Cliffs, New Jersey, 1967.
13. Gh. Păun: Computing with membranes. *Journal of Computer and System Sciences*, **1**(61):108–143, 2000. See also Turku Centre for Computer Science – TUCS Report No. 208, 1998.
14. Gh. Păun: *Membrane computing. An introduction*. Springer-Verlag, Berlin, 2002.
15. Gh. Păun, Y. Suzuki, H. Tanaka. P Systems with Energy Accounting. *International Journal Computer Math.*, **78**(3):343–364, 2001.
16. J.L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall, Englewood Cliffs, New Jersey, 1981.
17. The P systems Web page: `http://ppage.psystems.eu`