P systems with active membranes: Trading time for space

Antonio E. Porreca, Alberto Leporati, Giancarlo Mauri, Claudio Zandron

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

e-mail: {porreca,leporati,mauri,zandron}@disco.unimib.it

Abstract

We consider recognizer P systems having three polarizations associated to the membranes, and we show that they are able to solve the **PSPACE**-complete problem Quantified 3SAT when working in polynomial space and exponential time. The solution is uniform (all the instances of a fixed size are solved by the same P system) and uses only communication rules: evolution rules, as well as membrane division and dissolution rules, are not used. Our result shows that, as it happens with Turing machines, this model of P systems can solve in exponential time and polynomial space problems that cannot be solved in polynomial time, unless **P** = **PSPACE**.

1 Introduction

Membrane systems (usually called *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 of several membranes, embedded into a main membrane called the *skin*. Membranes divide the Euclidean space into *regions*, that contain some *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. An interesting subclass of membrane systems is constituted by *confluent recognizer* P systems, in which: (*i*) all computations halt, (*ii*) only two possible outputs exist (usually named *yes* and *no*), and (*iii*) the result produced by the system only depends upon its input, and is not influenced by the particular sequence of computation steps taken to produce it. For a systematic introduction to P systems we refer the reader to [15], whereas the latest information can be found in [21].

Since the introduction of membrane 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 or efficiency of these systems. In this respect, it is known [17, 20, 9] that the class of all decision problems which can be solved in polynomial time by a family of recognizer P systems that use only evolution, communication and dissolution rules coincides with the standard complexity class **P**. Hence, in order to solve computationally hard (such as **NP**-complete or **PSPACE**-complete) problems in polynomial time by means of P systems it seems necessary to be able to produce an exponential number of membranes in a polynomial number of computation steps. To this aim, several variants of membrane division rules have been introduced.

Recognizer P systems with active membranes (using division rules and, possibly, polarizations associated to membranes) have thus been successfully used to solve **NP**-complete problems *efficiently* (from a time-complexity standpoint). The first solutions were given in the so called *semi-uniform* setting [14, 20, 10, 12], which means that we assume the existence of a deterministic Turing machine that, for every instance of the problem, produces in polynomial time a description of the P system that solves such an instance. The solution is computed in a *confluent* manner, meaning that the instance given in input is positive if and only if every computation of the P system associated with it is an accepting computation. Another way to solve **NP**-complete problems by means of P systems is by considering the *uniform* setting, in which any instance of the problem can be given as input — encoded in an appropriate way — to a specific P system and then solved by it. Sometimes (like in this paper), a uniform solution to a decision problem is provided by defining a family $\mathbf{\Pi} = \{\Pi_n : n \in \mathbb{N}\}$ of P systems such that for every $n \in \mathbb{N}$ the system Π_n reads in input an encoding of any possible instance of size n, and solves it. P systems with active membranes have thus been successfully used to design uniform polynomial-time solutions to some well-known **NP**-complete problems, such as SAT [16].

All the papers mentioned above deal with P systems having three polarizations, that use only division rules for elementary membranes (in [19] also division for non-elementary membranes is permitted, and in this way a semi-uniform solution to the **PSPACE**-complete problem QUANTIFIED SAT is provided), and working in the *maximally parallel* way. As shown in [1], the number of polarizations can be decreased to two without loss of efficiency. Also, in [8] the computational power of recognizer P systems with active membranes but without electrical charges and dissolution rules has been investigated, establishing that they characterize the complexity class **P**.

In this paper we trade time for space, that is, we allow an arbitrary number of computation steps but we impose that all computations performed by our P systems use at most a polynomial amount of space. As shown in [18], this constraint implies that the number of computation steps is at most exponential with respect to the input size. In what follows we will first recall from [18] the definition of space used by a computation performed by a P system. Then, we will focus our attention on recognizer P systems with active membranes (that, in the context of this paper, means associating three polarizations to the membranes, whereas division and dissolution rules are forbidden). We will show that these P systems are able to efficiently simulate deterministic register

machines, using only communication and evolution rules. Such a simulation will then be used to prove our main result: recognizer P systems with active membranes are able to solve, in a *uniform* way, all the instances of a given size of QUANTIFIED 3SAT (abbreviated as Q3SAT from now on), using a polynomial amount of space. This means that the complexity class **PSPACE** is contained into the class of decision problems which can be solved in polynomial space by the above kind of recognizer P systems; furthermore, such P systems can solve in arbitrary time (and polynomial space) problems which cannot be solved in polynomial time unless **P** = **PSPACE**.

The paper is organized as follows. In section 2 we first recall the definition of recognizer P systems with active membranes, and of space used by these P systems during computations. We also recall the definition and working of deterministic register machines. In section 3 we show that recognizer P systems with polarized membranes, communication and evolution rules are able to simulate deterministic register machines in an efficient way. We also show that, when an upper bound on the values contained in the registers of the simulated machines is known, we can indeed avoid evolution rules. In section 4 we show how register machines can be used to solve the **PSPACE**-complete decision problem Q3SAT in polynomial space. This is done by first writing a program that uniformly solves all the instances of Q3SAT of a specified size, then compiling this program into (extended) register machine instructions, which can then be translated easily into standard instructions. In section 5 we combine the results of the two previous sections, showing that the above model of recognizer P systems can solve Q3SAT by working in exponential time and polynomial space. Section 6 contains the conclusions and some directions for further research.

2 Mathematical preliminaries

Let us start by recalling the definition of the model of P systems we will use in the following.

Definition 1. A *P system with active membranes* (with polarized membranes and without division and dissolution rules), of degree $m \geq 1$, is a tuple

$$\Pi = (\Gamma, \Lambda, \mu, w_1, \ldots, w_m, R)$$

where:

- Γ is a finite alphabet of symbols, also called *objects*;
- Λ is a finite set of labels for the membranes;
- μ is a membrane structure (i.e., a rooted unordered tree) consisting of m membranes enumerated by $1, \ldots, m$. Furthermore, each membrane is labeled by an element of Λ. In this paper we use a one-to-one labeling, even if the original definition given in [14] does not necessarily require it.
- $w_1, \ldots, w_m$ are strings over Γ, describing the multisets of objects placed in the m initial regions of μ;
- R is a finite set of rules.

Each membrane possesses a further attribute, named *polarization* or *electrical charge*, which is either neutral (represented by 0), positive (+) or negative (−).

The rules are of the following kinds:

- *Object evolution rules*, of the form $[a \to w]_h^\alpha$

  They can be applied inside a membrane labeled by h, having polarization α and containing an occurrence of the object a; the object a is rewritten into the multiset w (i.e., a is removed from the multiset in h and replaced by every object in w).

- *Send-in communication rules*, of the form $a\,[\;]_h^\alpha \to [b]_h^\beta$

  They can be applied to a membrane labeled by h, having polarization α and such that the external region contains an occurrence of the object a; the object a is sent into h becoming b and, simultaneously, the polarization of h is changed to β.

- *Send-out communication rules*, of the form $[a]_h^\alpha \to [\;]_h^\beta\, b$

  They can be applied to a membrane labeled by h, having polarization α and containing an occurrence of the object a; the object a is sent out from h to the outside region becoming b and, simultaneously, the polarization of h is changed to β.

A *configuration* in a P system with active membranes is described by its membrane structure, together with its polarizations and the multisets of objects contained in its regions. The initial configuration is given by μ, all membranes having polarization 0 and the initial contents of the membranes being $w_1, \ldots, w_m$. A *computation step* changes the current configuration according to the following principles:

- Each object can be subject to at most one rule per computation step. Furthermore, each membrane can be involved in at most one rule of type send-in or send-out, but any number of object evolution rules can be applied inside it during a computation step.

- The rules are applied in a *maximally parallel way*: each object which appears on the left-hand side of applicable evolution or communication rules must be subject to exactly one of them; the same holds for each membrane which can be involved in a communication rule. The only objects and membranes which remain unchanged are those associated with no rule, or with unapplicable rules.

- When more than one rule can be applied to an object or membrane, the actual rule to be applied is chosen nondeterministically; hence, in general, multiple configurations can be reached from the current one.

- Every object which is sent out from the skin membrane cannot be brought in again.

A halting computation $\vec{\mathcal{C}}$ of a P system Π is a finite sequence of configurations $(\mathcal{C}_0, \ldots, \mathcal{C}_k)$, where $\mathcal{C}_0$ is the initial configuration of Π, every $\mathcal{C}_{i+1}$ can be reached from $\mathcal{C}_i$ according to the principles just described, and no further configuration can be reached from $\mathcal{C}_k$ (i.e., no rule can be applied). P systems might also

perform non-halting computations; in this case, we have infinite sequences $\vec{\mathcal{C}} = (\mathcal{C}_i : i \in \mathbb{N})$ of successive configurations.

We can use families of P systems with active membranes as language recognizers, thus allowing us to solve decision problems.

Definition 2. A *recognizer P system with active membranes* Π has an alphabet containing two distinguished objects *yes* and *no*, used to signal acceptance and rejection respectively; every computation of Π is halting and exactly one object among *yes*, *no* is sent out from the skin membrane during each computation.

In what follows we will only consider *confluent* recognizer P systems with active membranes, in which all computations starting from the initial configuration agree on the result.

Definition 3. Let $L \subseteq \Sigma^\star$ be a language and let $\mathbf{\Pi} = \{\Pi_x : x \in \Sigma^\star\}$ be a family of recognizer P systems. We say that $\mathbf{\Pi}$ *decides* L, in symbols $L(\mathbf{\Pi}) = L$, when for each $x \in \Sigma^\star$, the result of Π_x is acceptance iff $x \in L$.

Usually a uniformity condition, inspired by those applied to families of Boolean circuits, is imposed on families of P systems. In this paper we use plain (polynomial-time) uniformity, which is stricter than semi-uniformity.

Definition 4. A family of P systems $\mathbf{\Pi} = \{\Pi_x : x \in \Sigma^\star\}$ is said to be *uniform* when there exist two polynomial-time Turing machines M_1 and M_2 such that, for each $n \in \mathbb{N}$ and each $x \in \Sigma^n$

- M_1, on input 1^n (the unary representation of the length of x), outputs the description of a P system Π_n with a distinguished input membrane;
- M_2, on input x, outputs a multiset w_x (an encoding of x);
- Π_x is Π_n with w_x added to the multiset located inside its input membrane.

In other words, the P system Π_x associated with string x consists of two parts; one of them, Π_n, is common for all strings of length $|x| = n$ (in particular, the membrane structure and the set of rules fall into this category), and the other (the input multiset w_x for Π_n) is specific to x. The two parts are constructed independently and, only as the last step, w_x is inserted in Π_n.

Time complexity classes for P systems are defined as usual, by restricting the amount of time available for deciding a language. By $\mathbf{MC}_{\mathcal{D}}(f(n))$ we denote the class of languages which can be decided by uniform families of confluent P systems of type $\mathcal{D}$ (e.g., $\mathcal{AM}$ denotes P systems with active membranes) where each computation of $\Pi_x \in \mathbf{\Pi}$ halts within $f(|x|)$ steps. The class of languages decidable in polynomial time by P systems of type $\mathcal{D}$ is denoted by $\mathbf{PMC}_{\mathcal{D}}$.

Recently, a space complexity measure for P systems has been introduced [18]. We recall here the relevant definitions.

Definition 5. Let $\mathcal{C}$ be a configuration of a P system Π. The *size* $|\mathcal{C}|$ *of* $\mathcal{C}$ is defined as the sum of the number of membranes in the current membrane

structure and the total number of objects they contain[1]. If $\vec{\mathcal{C}} = (\mathcal{C}_0, \dots, \mathcal{C}_k)$ is a halting computation of Π, then the *space required by* $\vec{\mathcal{C}}$ is defined as

$$|\vec{\mathcal{C}}| = \max\{|\mathcal{C}_0|, \dots, |\mathcal{C}_k|\}$$

or, in the case of a non-halting computation $\vec{\mathcal{C}} = (\mathcal{C}_i : i \in \mathbb{N})$,

$$|\vec{\mathcal{C}}| = \sup\{|\mathcal{C}_i| : i \in \mathbb{N}\}.$$

Non-halting computations might require an infinite amount of space (in symbols $|\vec{\mathcal{C}}| = \infty$): for example, if the number of objects strictly increases at each computation step.

The *space required by* Π itself is then

$$|\Pi| = \sup\{|\vec{\mathcal{C}}| : \vec{\mathcal{C}} \text{ is a computation of } \Pi\}.$$

Notice that $|\Pi| = \infty$ might occur if either Π has a non-halting computation requiring infinite space (as described above), or Π has an infinite set of halting computations, such that for each bound $b \in \mathbb{N}$ there exists a computation requiring space larger than b.

Finally, let $\mathbf{\Pi} = \{\Pi_x : x \in \Sigma^\star\}$ be a family of recognizer P systems; also let $f : \mathbb{N} \to \mathbb{N}$. We say that $\mathbf{\Pi}$ *operates within space bound* f iff $|\Pi_x| \leq f(|x|)$ for each $x \in \Sigma^\star$.

By $\mathbf{MCSPACE}_{\mathcal{D}}(f(n))$ we denote the class of languages which can be decided by uniform families of confluent P systems of type $\mathcal{D}$ where each $\Pi_x \in \mathbf{\Pi}$ operates within space bound $f(|x|)$. The class of languages decidable in polynomial space by P systems of type $\mathcal{D}$ is denoted by $\mathbf{PMCSPACE}_{\mathcal{D}}$.

Next, we recall the definition of (deterministic) register machines.

Definition 6. A *deterministic n-register machine* is a construct $R = (n, I, m)$, where $n > 0$ is the number of registers, I is a finite sequence of instructions (program) bijectively labeled with the elements of the set $\{1, 2, \dots, m\}$, 1 is the label of the first instruction to be executed, and m is the label of the last instruction of I. Registers contain non-negative integer values. The instructions of I have the following forms:

- i: INC(r), j

  with $i \in \{1, 2, \dots, m\}$, $j \in \{1, 2, \dots, m+1\}$ and $r \in \{1, 2, \dots, n\}$. This instruction, labeled with i, increments the value contained in register r, and then jumps to instruction j.

- i: DEC(r), j, k

  with $i \in \{1, 2, \dots, m\}$, $j, k \in \{1, 2, \dots, m+1\}$ and $r \in \{1, 2, \dots, n\}$. If the value contained in register r is positive then decrement it and jump to instruction j. If the value of r is zero then jump to instruction k (without altering the contents of the register).

[1] An alternative definition, where the size of a configuration is given by the sum of the number of membranes and *the number of bits required to store the objects they contain*, has been considered in [18]. However, the choice between the two definitions is irrelevant as far as the results of this paper are concerned.

Computations start by executing the first instruction of I (labeled with 1), and terminate when the instruction currently executed tries to jump to label $m+1$.

Formally, a *configuration* is a $(n+1)$-tuple whose components are the contents of the n registers, and the label of the next instruction of I to be executed. In the *initial* configuration this label is set to 1, whereas it is equal to $m+1$ in any *final* configuration. A *computation* starts in the initial configuration and proceeds by performing computation steps, i.e., executing the current instruction, modifying accordingly the contents of the affected register and the pointer to the next instruction. A computation halts if it reaches a final configuration. The contents of the registers in the initial configuration are regarded as the input, and those in the final one as the output of the computation. A non-halting computation does not produce a result.

Register machines provide a simple universal computational model. Indeed, the results proved in [4] (based on the results established in [11]) as well as in [5] and [6] immediately lead to the following proposition.

Proposition 1. *For any partial recursive function* $f : \mathbb{N}^\alpha \rightharpoonup \mathbb{N}^\beta$ *there exists a deterministic* $(\max\{\alpha, \beta\} + 2)$*-register machine* M *computing* f *as follows: if* $f(x_1, \ldots, x_\alpha) = (y_1, \ldots, y_\beta)$*, then* M*, when started with* $(x_1, \ldots, x_\alpha) \in \mathbb{N}^\alpha$ *in registers* 1 *to* α*, all the other registers being empty, halts in the final label* $m+1$ *with registers* 1 *to* β *containing* y_1 *to* y_β*, and all other registers empty; if* $f(x_1, \ldots, x_\alpha)$ *is undefined, then the final label is never reached.* □

3 Simulating register machines by P systems

Let R be a register machine with n registers $r_1, \ldots, r_n$, and m instructions labeled by $i_1, \ldots, i_m$. In this section we design a P system with active membranes Π_R which computes the same function as R, under a chosen encoding of inputs and outputs in terms of multisets, by using only object evolution and communication rules. The construction is similar to the one described in [2]; however, we use deterministic register machines in the accepting mode (instead of nondeterministic ones in the generative mode). Furthermore, when an upper bound on the sum of the values in the registers of R is known in advance, we may also manage to simulate the register machine using *only* communication rules.

We begin by describing the membrane structure μ of Π_R. It consists of $n+2$ membranes, nested as follows:

$$\mu = \big[[\,]_1^0 \cdots [\,]_n^0 \; [\,]_z^0 \big]_0^0$$

that is, a skin membrane labeled by 0, containing n membranes labeled by $1, \ldots, n$, and a further membrane labeled by z. Membrane h, for $h \in \{1, \ldots, n\}$, corresponds to register r_h: these membranes will be referred to as *register-membranes* in the following discussion; membrane z is called the *waiting membrane*.

During the simulated computation of R, each register-membrane h of Π_R contains a number of occurrences of object a equal to the value stored in register r_h (i.e., the value of r_h in unary notation). We also need a *program counter object* (denoted by p, p' or p'', with a subscript indicating its value) to keep track of the current instruction label of R and to perform increment and decrement

operations: before each simulated instruction i of R, the object p_i is found inside the skin membrane.

The increment instruction "i: INC(r), j" is implemented as follows: first, object p_i enters membrane r using the communication rule

$$p_i\ [\ ]_r^0 \to [p_i']_r^0$$

Then, another copy of a is produced via object evolution:

$$[p_i' \to ap_i'']_r^0 \tag{1}$$

Finally, object pi_i'' is sent back to the skin membrane, with its subscript changed to the label of the next instruction:

$$[p_i'']_r^0 \to [\ ]_r^0\ p_j$$

Clearly, now membrane r contains one more a than before.

The simulation of a decrement instruction "i: DEC(r), j, k" is more involved, particularly since we want to keep (1) as the only object evolution rule (so that we may be able to easily eliminate evolution afterward). First p_i enters membrane r as before, but it changes the polarization to negative, thus signaling that a decrement instruction is being simulated:

$$p_i\ [\ ]_r^0 \to [p_i']_r^-$$

When r is negatively charged and contains an a, this object must be sent out in order to decrement the value of the simulated register:

$$[a]_r^- \to [\ ]_r^0\ a \tag{2}$$

Notice how the neutral polarization is restored by this communication rule. Now, to detect whether the decrement has actually been performed and update the next instruction label accordingly, we should make the program counter object wait for one step, then check whether r is neutrally or negatively charged. Normally, "waiting" is achieved by using evolution rules such as $[p_i' \to p_i'']_r^\alpha$; however, as said before, we are trying to limit the number of evolution rules at a minimum. Instead, we lead p_i' on a detour to the waiting membrane z and back to r again: this ensures that an object a inside r, if it exists at all, has the time to actually exit the membrane (recall that only one communication rule per step may be applied to a membrane, hence we need to handle conflicts between the rules involving a and p_i'). The rules we use are

$$\begin{aligned}
[p_i']_r^\alpha &\to [\ ]_r^\alpha\ p_i' && \forall \alpha \in \{0, -\} \\
p_i'\ [\ ]_z^0 &\to [p_i']_z^0 \\
[p_i']_z^0 &\to [\ ]_z^0\ p_i'' \\
p_i''\ [\ ]_r^\alpha &\to [p_i'']_r^\alpha && \forall \alpha \in \{0, -\}
\end{aligned} \tag{3}$$

Finally, object p_i'' looks at the polarization of r, thus establishing whether the content of the simulated register was zero or positive (in which case, an a was sent out to decrement it):

$$[p_i'']_r^0 \to [\ ]_r^0\ p_j$$
$$[p_i'']_r^- \to [\ ]_r^0\ p_k$$

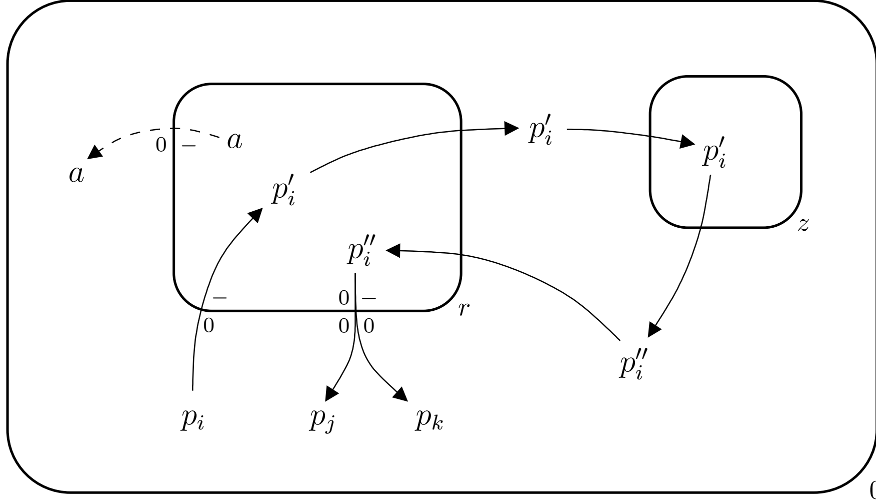

Figure 1: Simulating the instruction "i: DEC(r), j, k" by a P system with polarized membranes and using only communication rules.

Notice that the charge of r is set to neutral again, and that the subscript of p changes according to the outcome of the instruction.

The simulation of the decrement instruction is not completely deterministic: indeed, rule (2) may be applied either immediately before rule (3), or immediately after, with no consequences on the correctness of the simulation. The trajectories of the objects involved and the various changes of polarization are depicted in Figure 1.

Notice that this simulation immediately proves the universality of P systems with active membranes using evolution and communication rules; moreover, the simulation is efficient, having only a linear slowdown (see also [2]).

When an upper bound $B(x_1, \ldots, x_k)$ on the sum of the values of the registers of machine R during the whole computation is known beforehand, where $(x_1, \ldots, x_k)$ denote the inputs of R, object evolution rules can be avoided altogether. The idea is to place $B(x_1, \ldots, x_k) - \sum_{i=1}^{k} x_i$ occurrences of a inside the skin membranes, and to bring them in the register-membranes whenever an increment operation is performed (instead of creating new objects by evolution). The increment instruction "i: INC(r), j" can then be implemented by the following three communication rules:

$$p_i \; [\;]_r^0 \to [p_i']_r^+$$
$$a \; [\;]_r^+ \to [a]_r^0$$
$$[p_i']_r^0 \to [\;]_r^0 \; p_j$$

Here object p_i' exits from r only when the polarization is neutral again, due to a having entered the membrane (i.e., when the increment operation is finished).

4 Solving Q3SAT using register machines

A Boolean formula is said to be in *3-conjunctive normal form* (3CNF) if it is a conjunction of clauses and each clause is, in turn, the disjunction of exactly three literals (a literal is either a variable or a negated variable). For the present discussion we may assume, without loss of generality, that each variable occurs at most once in each clause. Since both conjunction and disjunction commute, formulae in 3CNF can be identified with unordered sets of unordered sets of three variables, each one associated with three bits indicating which variables are negated. We denote a Boolean formula by $\varphi(x_1, \ldots, x_m)$ to highlight the fact that the variables occurring in φ are among $x_1, \ldots, x_m$.

The Q3SAT decision problem is the following one: given a Boolean formula $\varphi(x_1, \ldots, x_m)$ in 3CNF, is $Q_1 x_1 \cdots Q_m x_m \varphi(x_1, \ldots, x_m)$ true? Here the Q_i denote alternating quantifiers, either $\forall$, if i is odd, or $\exists$, if i is even, and each quantified variable ranges over the set $\{0, 1\}$ of truth values. The Q3SAT problem is known to be **PSPACE**-complete [7, p. 46].

Instead of solving Q3SAT with a single device, as in the usual approach, we design a *family* of register machines, each one solving the problem for all inputs of a fixed size. These register machines will, in turn, be simulated in a later section by a uniform family of P systems with active membranes.

Once the number of variables m has been fixed, we have $\binom{m}{3}$ sets of three variables, and then only $8\binom{m}{3}$ possible clauses of three literals, as defined above, can be constructed, since 8 is the number of ways in which three variables can be chosen as positive or negated. Hence, we can describe each Boolean formula $\varphi(x_1, \ldots, x_m)$ in 3CNF with $n = 8\binom{m}{3} = O(m^3)$ bits: the i-th bit will be set to 1 iff the i-th clause (under a fixed ordering) appears in $\varphi(x_1, \ldots, x_m)$. The value n will be used as the size of the formula.

It is important to observe that the number of (potential) variables m can be recovered from n by finding the unique positive integer root of the polynomial

$$p(m) = 8\binom{m}{3} - n = \frac{4}{3}m^3 - 4m^2 + \frac{8}{3}m - n.$$

This process also allows us to identify malformed inputs, i.e., those having length different from $8\binom{m}{3}$ for every positive integer m.

Our family $\mathbf{R}$ of register machines consists of a machine R_n for all inputs $\varphi(x_1, \ldots, x_m)$ of size n, in symbols $\mathbf{R} = \{R_n : n \in \mathbb{N}\}$. Those machines R_n corresponding to valid input lengths are designed to decide, once their n input registers have been initialized with the binary representation of a formula $\varphi(x_1, \ldots, x_m)$ of length n, whether the quantified version of such formula holds. If n is, instead, an invalid input length, then R_n will reject any input. We will also impose a uniformity condition on $\mathbf{R}$: namely, that there exists a deterministic Turing machine M operating in polynomial time which outputs a description of R_n, given 1^n (the unary representation of n) as input.

4.1 A program for all instances of a given size

As noticed above, a Boolean formula $\varphi(x_1, \ldots, x_m)$ of m variables can be described by $n = 8\binom{m}{3}$ bits, one for each potential clause, once an ordering of the clauses has been fixed. We begin by ordering the literals as follows:

$$x_1 \prec \cdots \prec x_m \prec \neg x_1 \prec \cdots \prec \neg x_m.$$

```
EVALUATE φ(t_1, ..., t_m)

if c_1 then
    v_1 ← t_1 ∨ t_2 ∨ t_3
else
    v_1 ← 1
end;
  ⋮
if c_n then
    v_n ← ¬t_{m−2} ∨ ¬t_{m−1} ∨ ¬t_m
else
    v_n ← 1
end;
r ← v_1 ∧ ··· ∧ v_n
```

Figure 2: Evaluation of the Boolean formula $\varphi(x_1, \ldots, x_m)$ in 3CNF under a given truth assignment $t_1, \ldots, t_m$ to $x_1, \ldots, x_m$.

Then, given two clauses C_1, C_2, we identify them with triples of distinct literals $\vec{C}_1 = (\ell_{1,1}, \ell_{1,2}, \ell_{1,3})$ and $\vec{C}_2 = (\ell_{2,1}, \ell_{2,2}, \ell_{2,3})$ such that $\ell_{1,1} \prec \ell_{1,2} \prec \ell_{1,3}$ and $\ell_{2,1} \prec \ell_{2,2} \prec \ell_{2,3}$. Finally, we say that $C_1 < C_2$ iff $\vec{C}_1$ is less than $\vec{C}_2$ under the lexicographic order induced by $\prec$. All clauses over m variables can then be enumerated algorithmically in the given order in polynomial time.

For instance, this is a (left-to-right, top-to-bottom) ordered enumeration of the 8 clauses over the three variables x_1, x_2, x_3:

$$\begin{array}{llll} x_1 \vee x_2 \vee x_3 & x_1 \vee x_2 \vee \neg x_3 & x_1 \vee \neg x_2 \vee x_3 & x_1 \vee \neg x_2 \vee \neg x_3 \\ \neg x_1 \vee x_2 \vee x_3 & \neg x_1 \vee x_2 \vee \neg x_3 & \neg x_1 \vee \neg x_2 \vee x_3 & \neg x_1 \vee \neg x_2 \vee \neg x_3 \end{array}$$

The 3CNF Boolean formula

$$\varphi(x_1, x_2, x_3) = (x_2 \vee \neg x_1 \vee \neg x_3) \wedge (x_2 \vee \neg x_1 \vee x_3) \wedge (x_3 \vee \neg x_2 \vee \neg x_1)$$

is then described by the binary string $c = 0000\ 1110$.

We are now able to describe a program to evaluate a Boolean formula $\varphi(x_1, \ldots, x_m)$, given the binary representation $c = c_1 \cdots c_n$ of the formula and a truth assignment to the variables. Notice that the program only works for formulae having a binary description of length n. For each possible clause C_i, if C_i actually occurs in $\varphi(x_1, \ldots, x_m)$, i.e., if bit c_i is 1, then the value v_i of the clause is determined by computing the disjunction of the three corresponding literals; otherwise, we set v_i to 1. Finally, the result r of the evaluation is computed. The program is illustrated in Figure 2.

The evaluation of the *quantified* version of $\varphi(x_1, \ldots, x_m)$ is performed by trying all truth assignments to $x_1, \ldots, x_n$ and combining the various results according to the rules governing the quantifiers. A straightforward algorithm involves fixing the value of a variable x_i first to 0, then to 1; the algorithm is then called recursively on the resulting formulae, and the two results are combined either by conjunction (if x_i is universally quantified) or disjunction (if x_i is existentially quantified). The base case occurs when the values of all variables have been set: the formula can then be evaluated as described above.

```
EVALUATE ∀x₁∃x₂ ··· Q_m x_m φ(x₁, ..., x_m)

r_1 ← 1;
for t_1 ← 0 to 1 do
    r_2 ← 0;
    for t_2 ← 0 to 1 do
        ⋱
            r_m ← e;
            for t_m ← 0 to 1 do
                r ← φ(t_1, ..., t_m);
                r_m ← r_m ◇ r
            end;
        ⋰
        r_2 ← r_2 ∨ r_3
    end;
    r_1 ← r_1 ∧ r_2
end
```

Figure 3: Program P_n, with $n = 8\binom{m}{3}$, evaluating the quantified Boolean formula in 3CNF $\forall x_1 \exists x_2 \cdots Q_m x_m \varphi(x_1, \ldots, x_m)$. If m is odd, hence Q_m is $\forall$, then e is to be interpreted as 1 (the identity of conjunction) and $\Diamond$ as $\wedge$; if m is even, hence Q_m is $\exists$, then e is 0 (the identity of disjunction) and $\Diamond$ is $\vee$. The output of the program is the value of r_1.

However, when the number of variables m is known *a priori*, as in our case, the m levels of recursion can be replaced by m nested loops. The i-th outermost loop is associated with the i-th quantifier, and it evaluates the formula

$$\begin{aligned} &Q_{i+1}x_{i+1} \cdots Q_m x_m \varphi(t_1, \ldots, t_{i-1}, 0, x_{i+1}, \ldots, x_m) \Diamond \\ &Q_{i+1}x_{i+1} \cdots Q_m x_m \varphi(t_1, \ldots, t_{i-1}, 1, x_{i+1}, \ldots, x_m) \end{aligned}$$

where $t_1, \ldots, t_{i-1}$ is the truth assignment to $x_1, \ldots, x_{i-1}$ which has been fixed by the surrounding $i-1$ loops; the connective $\Diamond$ is to be interpreted as $\wedge$ if Q_i is $\forall$, and as $\vee$ otherwise. The base case is $i = m$, when all variables have been assigned a value: the evaluation may then proceed as described above. Clearly, the value r_1 computed by the outermost loop ($i = 1$) is the value of the whole quantified formula, and is considered as the output of the program. The complete program is described in Figure 3. Notice that all program variables range over $\{0, 1\}$, and can thus be interpreted as Boolean values.

Let us denote by P_n the program deciding the validity of all quantified 3CNF formulae of length $n = 8\binom{m}{3}$ and m variables, having input $c = c_1 \cdots c_n$ (if n is not a valid length, then P_n is the single-instruction program "$r_1 \leftarrow 0$"). Given the regular structure of the family of programs $\mathbf{P} = \{P_n : n \in \mathbb{N}\}$, each one consisting of:

- m alternated universal/existential nested loops, which only differ for the subscripts of the variables;
- n sequential conditional statements, essentially an enumeration of the n potential clauses of m variables;

- the conjunction of n variables,

hence having polynomial size with respect to n, there exists a deterministic Turing machine M_1 which, given the input 1^n (i.e., n in *unary* notation), outputs program P_n in polynomial time with respect to n. We call $\mathbf{P}$ a *uniform* family of programs.

As for the time and space complexity of $\mathbf{P}$, notice that program P_n executes $O(n2^m)$ steps, but only requires $O(n)$ memory space: indeed, each of the $O(n)$ program variables only assumes value 0 or 1.

4.2 Compiling programs into register machines

Given the simple language we chose for the family of programs $\mathbf{P}$, it is easy to describe how they can be translated, mechanically and efficiently, into register machines using the same amount of resources. In order to simplify this description even further, we begin by augmenting the instruction set of register machines.

The instruction "SETZERO(y), j" sets the value of register y to 0 and goes to instruction j. It is implemented in terms of increment and decrement as follows:

i_1: DEC(y), i_1, j

Analogously, "SETONE(y), j" sets the value of register y to 1 and jumps to j; it is translated as

i_1: SETZERO(y), i_2
i_2: INC(y), j

We also introduce an unconditional jump instruction "JUMP i", defined as

i_1: DEC(z), i, i

where z is a new auxiliary register which is only used by these jump instructions, and it is assumed to always contain 0. The instruction "JZERO(y), j, k" jumps to instruction j when register y contains 0, and to instruction k otherwise; in both cases, the value of register y is preserved (recall that the only values of y we use are 0 and 1):

i_1: DEC(y), i_2, j
i_2: INC(y), k

Finally, "JONE(y), j, k" jumps to instruction j when y contains 1, and to k otherwise; the value of y is preserved also in this case:

i_1: DEC(y), i_2, k
i_2: INC(y), j

We now turn our attention to the translation of program constructs into register machine instructions. First of all, each program variable is represented by a different register (here we use the same name for both of them). Simple assignments such as "$y \leftarrow 0$" and "$y \leftarrow 1$" are compiled into "SETZERO(y), j" and "SETONE(y), j" respectively, where j denotes the label of the next instruction.

Assignments of the form "$y \leftarrow \ell_1 \diamond \cdots \diamond \ell_k$", with $\diamond$ either $\wedge$ or $\vee$, are first rewritten as "$y \leftarrow \ell_1 \diamond \ell_2; y \leftarrow y \diamond \ell_3; \cdots ; y \leftarrow y \diamond \ell_k$". Then, the assignment "$y \leftarrow u \wedge v$", with u and v program variables, is compiled into

i_1: JZERO(u), i_4, i_2
i_2: JZERO(v), i_4, i_3
i_3: SETONE(y), i_5
i_4: SETZERO(y), i_5

where i_5 is the instruction after i_4. Other kinds of assignment such as "$y \leftarrow u \vee v$" and "$y \leftarrow u \vee \neg v$" are translated similarly, according to the logical meaning of the expression on the right-hand side.

Conditional statements, which are of the form "if y then *branch*$_1$ else *branch*$_2$ end", are translated as follows:

i_1: JZERO(y), i_4, i_2
i_2: $\langle branch_1 \rangle$
i_3: JUMP i_5
i_4: $\langle branch_2 \rangle$

where i_5 is the instruction after i_4, and $\langle branch_k \rangle$ denotes the register machine code corresponding to $branch_k$.

Finally, all loop statements are of the form "for $y \leftarrow 0$ to 1 do *body* end" and can be compiled into the following code:

i_1: SETZERO(y), i_2
i_2: $\langle body \rangle$
i_3: JONE(y), i_5, i_4
i_4: INC(y), i_2

where i_5 is the label of the next instruction.

A deterministic Turing machine M_2 can then compile program P_n into a register machine R_n in polynomial time, by first building the parse tree of P_n, then traversing it while outputting the register machine instructions corresponding to each node. This allows us to prove the following result:

Lemma 1. *The* Q3SAT *problem can be solved by a uniform family of register machines* $\mathbf{R} = \{R_n : n \in \mathbb{N}\}$ *such that the sum of the values of the registers of* R_n *is* $O(n)$ *on every possible input.*

Proof. The family $\mathbf{R}$ can be constructed in polynomial time, given the unary representation of n as input, by a deterministic Turing machine M which simulates M_1 on 1^n, thus obtaining the text of program P_n, then compiles it into register machine R_n by simulating M_2 on P_n. Each register machine R_n uses $O(n)$ Boolean-valued variables because it is a direct translation of program P_n, which enjoys this property. □

5 Solving Q3SAT via P systems

We now turn our attention to a solution to Q3SAT by simulation of the family of register machines via a uniform family of P systems with active membranes. Two questions still need to be addressed: (i) How are the register-membranes initialized? Recall that in the uniform construction of P systems, the input must be placed inside a *single* membrane. (ii) How is the output of the P systems generated from the output of the register machine (which is a binary output)?

Our answer to question (i) is the following: if x_i is the input to be placed in register r_i, then we put x_i occurrences of a_i inside the skin membranes; these are then moved to register-membrane i by communication rules:

$$a_i\ [\ ]_i^0 \to [a]_i^0$$

However, the program counter object needs to wait for all the register-membranes to be initialized. Let $\ell = \max\{x_1, \ldots, x_k\}$ be the maximum of the input values of the simulated register machine; before introducing the program counter object, a timer q (initialized to ℓ) is decremented, by traveling through the waiting membrane z back and forth, until it reaches 0; then, the first step of the register machine can be finally simulated. These are the rules we need:

$$\begin{aligned} q_t\ [\ ]_z^0 &\to [q_{t-1}]_z^0 && \forall t \in \{1, \ldots, \ell\} \\ [q_t]_z^0 &\to [\ ]_z^0\ q_{t-1} && \forall t \in \{1, \ldots, \ell - 1\} \\ q_0\ [\ ]_z^0 &\to [q_0]_z^0 \\ [q_0]_z^0 &\to [\ ]_z^0\ p_1 \end{aligned}$$

To answer question (ii), we simply need to check if the output register of the machine (called r_1 in all machines of the family $\mathbf{R}$) contains 0 or 1, and then produce *no* or *yes* accordingly. If i_m is the label of the last instruction of the simulated machine, add the further decrement instruction "i_{m+1}: $\textsc{dec}(r_1)$, i_{m+2}, i_{m+3}" after i_m; then, the following rules produce the correct output for the P system:

$$\begin{aligned} [p_{m+2}]_0^0 &\to [\ ]_0^0\ \textit{yes} \\ [p_{m+3}]_0^0 &\to [\ ]_0^0\ \textit{no} \end{aligned}$$

Having fixed these two details, we can define a family $\mathbf{\Pi} = \{\Pi_n : n \in \mathbb{N}\}$ of P systems with active membranes, using communication rules only, which simulates the family $\mathbf{R}$ of register machines. This is possible since Lemma 1 gives us an upper bound on the total number of occurrences of a we need.

Lemma 2. *The family* $\mathbf{\Pi}$ *of P systems is uniform.*

Proof. Each Π_n can be constructed from R_n, hence from 1^n, in polynomial time:

- the membrane structure of Π_n is only larger than the number of registers of R_n by a constant amount (due to membranes 0 and z);
- each instruction of R_n is simulated by $O(1)$ evolution rules, which can be computed efficiently from it;
- we only added $O(n)$ rules to initialize the register-membranes in this section, and these rules can be easily determined from n;
- the number of copies of a we put inside the skin membrane of Π_n, in order to simulate increment instructions, is $O(n)$.

The input of Π_n consists of an occurrence of object a_i for each nonzero bit c_i in the binary description of $\varphi(x_1, \ldots, x_m)$; hence, it can also be computed in polynomial time by another Turing machine. This input is then placed inside the skin membrane of Π_n before the beginning of the computation. □

We have thus proved our main result.

Theorem 1. *Let* $\mathcal{AM}(-\text{ev}, +\text{com}, -\text{dis}, -\text{div})$ *be the class of P systems with active membranes using only communication rules. Then*

$$\text{Q3SAT} \in \mathbf{PMCSPACE}_{\mathcal{AM}(-\text{ev},+\text{com},-\text{dis},-\text{div})}$$

hence $\mathbf{PSPACE} \subseteq \mathbf{PMCSPACE}_{\mathcal{AM}(-\text{ev},+\text{com},-\text{dis},-\text{div})}$.

Proof. Keeping in mind the results of Lemmata 1 and 2, just observe that the uniform family of P systems $\mathbf{\Pi}$ operates in polynomial space, since it uses $O(n)$ membranes and $O(n)$ objects. This proves

$$\text{Q3SAT} \in \mathbf{PMCSPACE}_{\mathcal{AM}(-\text{ev},+\text{com},-\text{dis},-\text{div})}.$$

The inclusion $\mathbf{PSPACE} \subseteq \mathbf{PMCSPACE}_{\mathcal{AM}(-\text{ev},+\text{com},-\text{dis},-\text{div})}$ follows from the closure under polynomial-time reductions of the latter class [18] and the **PSPACE**-completeness of Q3SAT. □

6 Conclusions and directions for future research

We have shown that recognizer P systems with active membranes (using three polarizations) are able to solve the **PSPACE**-complete problem Q3SAT when working in polynomial space and exponential time. The proposed solution is *uniform*, in the sense that a fixed P system is able to solve all the instances of Q3SAT of a given size. Moreover, we use only communication rules: evolution rules, as well as membrane division and dissolution rules, are not needed.

Our result shows that, as it happens with Turing machines, recognizer P systems with active membranes can solve in exponential time and polynomial space problems that cannot be solved in polynomial time, unless $\mathbf{P} = \mathbf{PSPACE}$.

As a byproduct, we have shown that recognizer P systems using polarized membranes, evolution and communication rules (that is, without using division and dissolution rules) are able to efficiently simulate deterministic register machines. In particular, the space complexity is proportional to the number of registers and their values, and each instruction of the register machines requires a constant number of computation steps to be simulated. Furthermore, if the values contained in the registers of the simulated machine can be bounded *a priori*, then evolution rules can be avoided.

Our main result can be restated in terms of computational complexity classes by saying that $\mathbf{PSPACE} \subseteq \mathbf{PMCSPACE}_{\mathcal{AM}(-\text{ev},+\text{com},-\text{dis},-\text{div})}$, where the symbol $\mathcal{AM}(-\text{ev}, +\text{com}, -\text{dis}, -\text{div})$ denotes the above class of recognizer P systems. A problem that remains open is whether the opposite inclusion also holds; we conjecture that the answer is affirmative.

7 Acknowledgments

We would like to thank Damien Woods for the suggestion to avoid object evolution rules in our solution to Q3SAT. This work was partially supported by the Italian project FIAR 2007 "Modelli di calcolo naturale e applicazioni alla Systems Biology".

References

[1] Alhazov A, Freund R (2005) On the efficiency of P systems with active membranes and two polarizations. In: Membrane Computing, Fifth International Workshop, WMC 2004, Lecture Notes in Computer Science, vol 3365, pp 81–94

[2] Ciobanu G, Marcus S, Păun G (2009) New strategies of using the rules of a P system in a maximal way: Power and complexity. Romanian Journal of Information Science and Technology 12(2):157–173

[3] Freund R, Leporati A, Oswald M, Zandron C (2005) Sequential P systems with unit rules and energy assigned to membranes. In: Machines, Computation and Universality, MCU, Lecture Notes in Computer Science, vol 3354, pp 200–210

[4] Freund R, Oswald M (2002) GP systems with forbidding context. Fundamenta Informaticae 49(1–3):81–102

[5] Freund R, Păun G (2001) On the number of non-terminals in graph-controlled, programmed, and matrix grammars. In: Machines, Computation and Universality, MCU, Lecture Notes in Computer Science, vol 2055, pp 214–225

[6] Freund R, Păun G (2004) From regulated rewriting to computing with membranes: Collapsing hierarchies. Theoretical Computer Science 312:143–188

[7] Garey MR, Johnson DS (1979) Computers and intractability: A guide to the theory of NP-completeness. W.H. Freeman

[8] Gutiérrez-Naranjo MA, Pérez-Jiménez MJ, Riscos-Núñez A, Romero-Campero FJ (2006) On the power of dissolution in P systems with active membranes. In: Membrane Computing, Sixth International Workshop, WMC 2005, Lecture Notes in Computer Science, vol 3850, pp 224–240

[9] Gutiérrez-Naranjo MA, Pérez-Jiménez MJ, Riscos-Núñez A, Romero-Campero FJ, Romero-Jiménez A (2006) Characterizing tractability by cell-like membrane systems. In: Formal models, languages and applications, Series in Machine Perception and Artificial Intelligence, vol 66, pp 137–154

[10] Krishna SN, Rama R (1999) A variant of P systems with active membranes: Solving NP-complete problems. Romanian Journal of Information Science and Technology 2(4):357–367

[11] Minsky ML (1967) Computation: Finite and infinite machines. Prentice Hall

[12] Obtułowicz A (2001) Deterministic P systems for solving SAT problem. Romanian Journal of Information Science and Technology 4(1–2):551–558

[13] Păun G (2000) Computing with membranes. Journal of Computer and System Sciences 1(61):108–143

[14] Păun G (2001) P systems with active membranes: Attacking NP-complete problems. Journal of Automata, Languages and Combinatorics 6(1):75–90

[15] Păun G (2002) Membrane computing: An introduction, Springer-Verlag

[16] Pérez-Jiménez MJ, Romero-Jiménez A, Sancho-Caparrini F (2006) A polynomial complexity class in P systems using membrane division. Journal of Automata, Languages and Combinatorics 11(4):423–434. A preliminary version appears in: Proceedings of the Fifth International Workshop on Descriptional Complexity of Formal Systems, 2003, pp 284–294

[17] Pérez-Jiménez MJ, Romero-Jiménez A, Sancho-Caparrini F (2004): The P versus NP problem through cellular computing with membranes. In: Aspects of Molecular Computing, Lecture Notes in Computer Science, vol 2950, pp 338–352

[18] Porreca AE, Leporati A, Mauri G, Zandron C (2009) Introducing a space complexity measure for P systems. International Journal of Computers, Communications & Control 4(3):301–310

[19] Sosik P (2003) The computational power of cell division: Beating down parallel computers? Natural Computing 2(3):287–298

[20] Zandron C, Ferretti C, Mauri G (2000) Solving NP-complete problems using P systems with active membranes. In: Unconventional Models of Computation, Springer-Verlag, pp 289–301

[21] The P Systems Webpage, `http://ppage.psystems.eu`