Boolean Circuits and a DNA Algorithm in Membrane Computing

Mihai Ionescu[1] **and Tseren-Onolt Ishdorj**
Research Group on Mathematical Linguistics
Rovira i Virgili University
Pl. Imperial Tàrraco 1, 43005 Tarragona, Spain
mi@urv.net; tserenonolt.ishdorj@estudiants.urv.es

Abstract. In the present paper we propose a way to simulate Boolean gates and circuits in the framework of P systems with active membranes using inhibiting/de-inhibiting rules. This new approach on the simulation of Boolean gates has the advantage of a self-embedded synchronization, an extra system to solve this problem not being needed. Moreover, the number of membranes and objects we use for the Boolean gates simulation is only two. **NP**-complete problems, particularly `CIRCUIT-SAT`, were also considered here. In addition, we propose a 'DNA-like' way of solving `SAT` problem using the tools given by merging and separation operations in P systems.

1 Introduction

P systems are a class of distributed parallel computing devices of a biochemical type, which can be seen as a general computing architecture where various types of objects can be processed by various operations.

In membrane computing, P systems with active membranes have a special place, because they provide biologically inspired tools to solve computationally hard problems. In [6] was introduced and explored the computational power of a class of P systems using catalytic and non-cooperative inhibiting/de-inhibiting rules, and in [7] this controlling mechanism was investigated in the framework of P systems with active membranes.

Boolean circuits are well known classical computing devices, which incorporate features of parallelism. Various possibilities to simulate Boolean circuits with P systems with promoters/inhibitors, mobile catalysts, and weak priorities for rules were considered in [8].

In this paper, we propose a model to simulate Boolean circuits through inhibiting/de-inhibiting P systems with active membranes (AID P Systems). The idea behind the simulation of such a circuit is to construct a global AID P system for the whole circuit having distributed sub-AID P systems for each gate. The sub-AID P systems work in a parallel manner producing an unique output as the result of the computation of the whole system. One can make a parallelism between the concept of *inhibition* (which means *blocking* the execution of a rule in a membrane), and the term *switch-off* frequently used

[1]The work of first author was supported by the fellowship "Formación de Profesorado Universitario" from the Spanish Ministry of Education, Culture and Sport.

in the theory of circuits, and, conversely, between *de-inhibition* and *switching on* some parts of the circuits.

Using this model to simulate Boolean circuits, one does not need an extra system to coordinate the entrance of the two inputs in an AND or an OR gate as presented in [8]. Here, the input(s) wait each-other and produce the right result when the circuit is simulated. We say that the system has a self-embedded synchronization.

Boolean `CIRCUIT-SAT` (which is **NP**-complete) is also considered here.

The second proposal of this paper is the construction of a 'DNA-like' system for solving `SAT` problem, following the idea of Lipton presented in [13], this time, using the properties of merging and separation operations from membrane computing. The motivation behind this proposal is to create a model as close as possible to the experimental results on solving the above mentioned problem using DNA strands (see, e.g., [4], [23]). We only say here that the main idea in such a simulation is to consider the test tubes used in the experiments as being membranes, and, of course, the DNA-strands contained in such a test tube as being the objects inside membranes. Details can be found in Section 6.

In the following section we will recall some of the notions regarding P systems with active membranes, Boolean gates, circuits, and DNA way of solving `SAT` we will use in the next sections.

2 Preliminaries

We assume the reader to be familiar with the fundamentals of formal language theory and complexity theory (for instance, from [17, 24, 25]), as well as with the basics of membrane computing, from [20].

2.1 P Systems with Active Membranes

Informally speaking, in P systems with polarization and active membranes one uses the following types of rules: (a) multiset rewriting rules, (b) rules for introducing objects into membranes, (c) rules for sending objects out of membranes, (d) rules for dissolving membranes, (e) rules for dividing elementary membranes, and (h) rules for separating membranes, see [2, 19, 22, 15, 16]. The rules of type (a_0) are applied in a parallel way (all objects which can evolve by such rules have to evolve), while the rules of types (b), (c), (d), (e), (g), (h) are used sequentially, in the sense that one membrane can be used by at most one rule of these types at a time. In total, the rules are used in the non-deterministic maximally parallel manner: all objects and all membranes which can evolve, should evolve. Only halting computations give a result, in the form of the number (or the vector) of objects expelled into the environment during the computation.

In this paper we will make use only of some of the mentioned rules, so we define P systems with active membranes using only such rules.

A P system *with active membranes* (and electrical charges) is a construct

$$\Pi = (O, H, \mu, w_1, \ldots, w_m, R),$$

where:

- $m \geq 1$ (the initial degree of the system);
- O is the alphabet of *objects*;

- H is a finite set of *labels* for membranes;
- μ is a *membrane structure*, consisting of m membranes, labeled (not necessarily in a one-to-one manner) with elements of H;
- $w_1, \dots, w_m$ are strings over O, describing the *multisets of objects* placed in the m regions of μ;
- R is a finite set of *developmental rules*, of the following forms:

  (a) $[\ a \to v]_h^e$,
  for $h \in H, e \in \{+, -, 0\}, a \in O, v \in O^*$
  (object evolution rules, associated with membranes and depending on the label and the charge of the membranes, but not directly involving the membranes, in the sense that the membranes are neither taking part in the application of these rules nor are they modified by them);

  (c) $[\ a\ ]_h^{e_1} \to [\ \ ]_h^{e_2} b$,
  for $h \in H, e_1, e_2 \in \{+, -, 0\}, a, b \in O$
  (communication rules; an object is sent out of the membrane, possibly modified during this process; also the polarization of the membrane can be modified, but not its label);

  (g) $[\ \ ]_h^{e_1} [\ \ ]_h^{e_2} \to [\ \ ]_h^{e_3}$,
  for $h \in H$, $e_1, e_2, e_3 \in \{+, -, 0\}$
  (merging rules for elementary membranes; in reaction of two membranes, they are merged into a single membrane; the objects of the former membranes are put together in the new membrane);

  (h) $[\ \ ]_h^{e_1} \to [\ K]_h^{e_2} [\ \neg K]_h^{e_3}$,
  for $h \in H$, $e_1, e_2, e_3 \in \{+, -, 0\}$, $K \subseteq O$
  (separation rules for elementary membranes; the contents of membrane h is split into two membranes, the first one containing all objects from K and the second one containing all objects which are not in K).

The set H of labels has been specified because it is also possible to allow the change of membrane labels. For instance, a separation rule can be of the more general form

(h') $[\ \ ]_{h_1}^{e_1} \to [\ K]_{h_2}^{e_2} [\ \neg K]_{h_3}^{e_3}$,
for $h_1, h_2, h_3 \in H$, $e_1, e_2, e_3 \in \{+, -, 0\}, K \subseteq O$.

The change of labels can also be considered for other types of rules (but not for rules of type (a)).

P systems with active membranes without electrical charges were also considered and investigated (see [2, 3, 1, 12]). Let us consider now some rules without polarizations. They are of the following forms ("no electrical charges" means "neutral polarization"; as above, O is the alphabet of objects and H is the set of labels of membranes):

(a_0) $[\ a \to v]_h$, where $a \in O, v \in O^*$, and $h \in H$,

(b_0) $a[\ \ ]_h \to [\ b]_h$, where $a, b \in O$ and $h \in H$,

(c_0) $[\ a]_h \to [\ \ ]_h b$, where $a, b \in O$ and $h \in H$,

(g_0) $[\ ]_h[\ ]_h \rightarrow [\ ]_h$, where $h \in H$,

(h_0) $[\ ]_h \rightarrow [\ K]_h[\ \neg K]_h$, where $K \subseteq O$ and $h \in H$.

We recommend the reader unfamiliar with these rules to consult the above mentioned references for a better understanding of their functionality.

The subscript 0 indicates the fact that we do not use polarization for membranes. When the rules of a given type (α_0) are able to change the label(s) of the involved membranes, we denote that type of rules by (α'_0). For example, the primed versions of merging and separation rules are of the following forms:

(g'_0) $[\ ]_{h_1}[\ ]_{h_2} \rightarrow [\ ]_{h_3}$, for $h_1, h_2, h_3 \in H$.
(h'_0) $[\ ]_{h_1} \rightarrow [\ K]_{h_2}[\ \neg K]_{h_3}$, for $h_1, h_2, h_3 \in H, K \subseteq O$.

To understand the difference of uniform construction and semi-uniform construction of P systems, we recall some notions about solving decidability problems in the membrane computing framework. Given a decision question X, we say that it can be solved in polynomial (linear) time by recognizing P systems in a uniform way, if, informally speaking, we can construct in polynomial time a family of recognizing P systems Π_n, $n \in \mathbb{N}$, associated with the sizes n of instances $X(n)$ of the problem, such that the system Π_n; starting from a code of some $X(n)$, will always stop in a polynomial (linear, respectively) number of steps, sending out the object `yes` if the instance $X(n)$ has a positive answer and the object `no` if the instance $X(n)$ has a negative answer. In [19], the complexity classes related to P systems are defined in the semi-uniform way: P systems are constructed starting not from the size n, but from an instance $X(n)$. For a clearer description of the difference between uniform and semi-uniform constructions, please refer to [22].

2.2 Inhibiting/De-inhibiting (AID) P Systems with Active Membranes

The basic idea of the AID P systems model is that, when a rule (acting on the membranes or on the objects) is inhibited, then it cannot be applied until another rule de-inhibits it. The application of a rule can inhibit other rules (and in particular might inhibit itself).

A P system *with active membranes and inhibiting/de-inhibiting mechanism*, in short, an AID P system, without electrical charges and without using catalysts, is a construct

$$\Pi = (O, H, I, \mu, w_1, \ldots, w_m, R),$$

where:

- $m \geq 1$ is the initial degree of the system;
- O is the alphabet of *objects*;
- H is a finite set of *labels* for membranes;
- I is a finite set of *labels* for rules;
- μ is a *membrane structure*, consisting of m membranes, labeled with elements of H;
- $w_1, \ldots, w_m$ are strings over O, describing the *multisets of objects* placed in the m regions of μ;
- R is a finite set of *developmental rules* of various forms. Here are some examples:

(b_0) $r : a[\ ]_h \rightarrow [\ b]_h\{S\}$, for $r \in I, h \in H, a, b \in O, S \subseteq I$
(communication rules; an object is introduced in the membrane during this process);

(c_0) $r : [\ a\ ]_h \rightarrow [\ ]_h b\{S\}$, for $r \in I, h \in H, a, b \in O, S \subseteq I$
(communication rules; an object is sent out of the membrane during this process).

The rules in R are written as $r_j : \neg r\{S\}$ or as $r_j : r\{S\}$, where $r_j \in I$ and r is a rule of type $(a_0) - (l_0)$ (replicative-distribution rules are: (k_0) $r : a[\ ]_{h_1}[\ ]_{h_2} \rightarrow [\ u]_{h_1}[\ v]_{h_2}$, for $h_1, h_2 \in H, a \in O, u, v \in O^*$ – for sibling membranes; (l_0) $r : [\ a[\ ]_{h_1}]_{h_2} \rightarrow [\ [\ u]_{h_1}]_{h_2}v$, for $h_1, h_2 \in H, a \in O, u, v \in O^*$ – for nested membranes) from [1, 2, 6]; S is a string that represents a subset of I. The AID P systems works like general P systems with active membranes. The only difference consists in the fact, that, at each step, only the non-inhibited rules can be used. When a rule $r_j : r\{S\}$ is applied, the rules whose labels are specified in S are inhibited (if they were de-inhibited) or de-inhibited (if they were inhibited). Now, starting from an initial configuration, the system evolves according to the rules and objects present in the membranes, in a non-deterministic maximally parallel manner, and according to an universal clock. The system will make a successful computation if and only if it halts, meaning there is no applicable rule to the objects present in the halting configuration. The result of a successful computation is the number of objects present in the output membrane (or environment) in a halting configuration of Π. If the computation never halts, then we will have no output.

2.3 Boolean Functions and Circuits

An n-ary *Boolean function* is a function $f\{true, false\}^n \mapsto \{true, false\}$. $\neg$ (negation) is a unary Boolean function (the other unary functions are: constant functions and identity function). We say that Boolean expression φ with variables $x_1, \ldots, x_n$ *expresses* the n-ary Boolean function f if, for any n-tuple of truth values $t = (t_1, \cdots, t_n)$, $f(t)$ is *true* if $T \models \varphi$, and $f(t)$ is false if $T \not\models \varphi$, where $T(x) = t_i$ for $i = 1, \ldots, n$.

There are three primary boolean functions that are widely used: The NOT function - this is a just a negation; the output is the opposite of the input. The NOT function takes only one input, so it is called a unary function or operator. The output is true when the input is false, and vice-versa. The AND function - the output of an AND function is true only if its first input and its second input and its third input (etc.) are all true. The OR function - the output of an OR function is true if the first input is true or the second input is true or the third input is true (again, etc.). Both AND and OR can have any number of inputs, with a minimum of two.

Any n-ary Boolean function f can be expressed as a Boolean expression φ_f involving variables $x_1, \ldots, x_n$.

There is a potentially more economical way that expressions for representing Boolean functions–namely *Boolean circuits*. A Boolean circuit is a graph $C = (V, E)$, where the nodes in $V = \{1, \ldots, n\}$ are called the *gates* of C. Graph C has a rather special structure. First, there are no cycles in the graph, so we can assume that all edges are of the form (i, j), where $i < j$. All nodes in the graph have the "in-degree" (number of incoming edges) equal to 0, 1, or 2. Also, each gate $i \in V$ has a *sort* $s(i)$ associated with it, where $s(i) \in \{true, false, \vee, \wedge, \neg\} \cup \{x_1, x_2, \ldots\}$. If $s(i) \in \{true, false\} \cup \{x_1, x_2, \ldots\}$, then the in degree of i is 0, that is, i must have no incoming edges. Gates with no incoming edges are called the *inputs* of C. If $s(i) = \neg$, then i has "in-degree" one. If $s(i) \in \{\vee, \wedge\}$, then the in degree of i must be two. Finally, node n (the largest numbered gate in the circuit, which necessarily has no outgoing edges) is called the *output gate* of the circuit.

This concludes our definition of the *syntax* of circuits. The *semantics* of circuits specifies a truth value for each appropriate truth assignment. We let $X(C)$ be the set of all Boolean variables that appear in the circuit C (that is, $X(C) = \{x \in X \mid s(i) = x$ for some gate i of $C\}$). We say that a truth assignment T is appropriate for C if it is defined for all variables in $X(C)$. Given such a T, *the truth value of gate* $i \in V$, $T(i)$, is defined, by induction on i, as follows: If $s(i) = true$ then $T(i) = true$, and similarly if $s(i) = false$. If $s(i) \in X$, then $T(i) = T(s(i))$. If now $s(i) = \neg$, there is a unique gate $j < i$ such that $(j, i) \in E$. By induction, we know $T(j)$, and then $T(i)$ is *true* if $T(j) = false$, and vice-versa. If $s(i) = \vee$, then there are two edges (j, i) and (j', i) entering i. $T(i)$ is then *true* if only if at least one of $T(j)$, $T(j')$ is *true*. If $s(i) = \wedge$, then $T(i)$ is *true* if only if both $T(j)$ and $T(j')$ are *true*, where (j, i) and (j', i) are the incoming edges. Finally, *the value of the circuit*, $T(C)$, is $T(n)$, where n is the output gate.

2.4 Brief Description of Solving SAT in DNA Computing

Lipton's DNA-based solution of the satisfiability problem [13] uses some of the basic operations in DNA Computing:

–*merge* (given test tubes N_1 and N_2, we consider their union, understood as a multiset),

–*separate* (given a test tube N and a word w over the alphabet A, C, T, G, produce two test tubes $+(N, w)$ and $-(N, w)$, where $+(N, w)$ consists of all strands in N which contain w as a (consecutive substring), while $-(N, w)$ is its negation), and

–*detect* (given a tube N, return *true* if N contains at least one DNA strand, otherwise return $false$).

We begin with a graphical description of truth assignments. Assume that we are dealing with a propositional formula containing k variables. Consider the following directed graph:

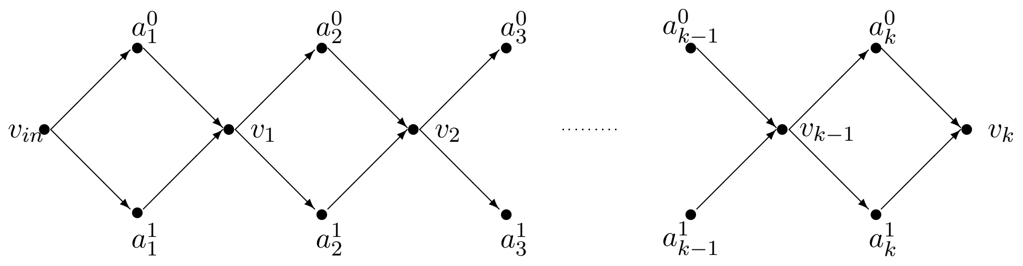

Fig. 1 A graph associated with a truth assignment

There are 2^k paths from v_{in} to v_{out}, none of the paths being Hamiltonian. One can observe that in each of the v_i nodes ($i \neq out$) there are two independent choices (0 or 1). The construction of the graph prevents the (unwanted) possibility of choosing for the same variable both 0 and 1 values. Moreover, the paths and the truth assignments for the variables $x_1, x_2, \ldots, x_k$ have a natural one-to-one correspondence.

Each vertex of the graph is encoded by a random oligonucleotide of length 20 and an arc between two vertices will be the Watson-Crick complement of the last half and the first half of the start and end nodes, respectively. More precisely, consider the encodings

s_i and s_j of two vertices such that there is an edge $e_{i,j}$ from the former to the latter. If $s_i = s'_i s''_i$, where s'_i and s''_i have equal length, and similarly, $s_j = s'_j s''_j$ then the edge $e_{i,j}$ is encoded by Watson-Crick complement of $s''_i s'_j$.

Now, having encoded all the possible truth-assignments with the help of the operations mentioned above and strictly depending on the clauses given in the `SAT` problem, an algorithm (based on separation, merging, and, only in the end, detection) will select the right solution(s) of the given problem. An example is presented in Section 6 where we compare the two (DNA and P-based) ways of solving a particular, simple `SAT` problem.

3 Simulating Logical Gates

In this section we present AID P systems which simulate logical gates. We will consider that the input for a gate is given in the inner membrane, while the output will be computed and sent out to the outer region.

Simulation of AND Gate

Lemma 3.1 *Boolean AND gate can be simulated by AID P systems with rules of types* (b_0) *and* (c'_0)*, using two membranes and two objects (only the input), in at most four steps.*

Proof. We construct the AID P system

$$\begin{aligned}
\Pi_{AND} &= (O, H, I, \mu, w_0, w_s, R), \text{with} \\
O &= \{0, 1\}, \\
\mu &= [\,[\ ]_0]_s, \\
w_0 &= w_s = \lambda, \\
H &= \{0, 1, s\}, \\
I &= \{r_i \mid 0 \le i \le 9\},
\end{aligned}$$

and the set R consisting of the following rules:

$r_1 : [\ 0]_0 \to [\ \ ]_1 0$

$r_2 : [\ 0]_1 \to [\ \ ]_0 \lambda \{r_2, r_8\}$

$r_3 : [\ 1]_0 \to [\ \ ]_1 1 \{r_2, r_4, r_5, r_6\}$

$r_4 : [\ 1]_1 \to [\ \ ]_0 \lambda \{r_2, r_8\}$

$r_5 : \neg[\ 0]_1 \to [\ \ ]_1 0 \{r_5, r_7\}$

$r_6 : \neg[\ 1]_1 \to [\ \ ]_1 \lambda \{r_4, r_6, r_9\}$

$r_7 : \neg 1[\ \ ]_1 \to [\ \lambda]_0 \{r_4, r_6, r_7, r_8\}$

$r_8 : \neg[\ 0]_s \to [\ \ ]_s 0 \{r_2, r_8\}$

$r_9 : \neg[\ 1]_s \to [\ \ ]_s 1 \{r_2, r_5, r_9\}$

We start by placing the input values x_1 and x_2 in the membrane with label 0. Depending on the value of the initial variables x_1 and x_2, the rules we apply for each of the four cases are:

for $x_1x_2 = 00 - r_1, r_2, r_8$,
for $x_1x_2 \in \{01, 10\} - r_1, r_4, r_8$, or r_3, r_5, r_7, r_8, and
for $x_1x_2 = 11 - r_3, r_6, r_9$.

More precisely, if two 1s are in membrane 0, in the first step, rule r_3 is applied, a 1 is expelled and membrane's label is changed to 1. In the same time according to the inhibition/de-inhibition concept, rules r_2 and r_4 are inhibited, while rules r_5 and r_6 are de-inhibited and ready to be used. In the second step we notice that only rule r_6 can be applied, thus, object 1, placed inside membrane labeled 1 is transformed, in its way out, into λ. One may notice that rule r_6, after is applied, restores the original status of rule r_4 and itself, and also de-inhibits rule r_9. In the third step, rule r_9 performs and the right answer 1 is sent out the skin membrane, while rules r_2, r_5, and r_9 come back to their original status.

In other words, after these three steps, our system sends out the skin membrane the right answer (given the input 11) and comes back to its initial configuration, thus being ready for a new input.

In the case when the input is 01 or 10, we can start by using r_1 or r_3. Let us examine the second case. Rule r_3 sends 1 out of membrane 0 and changes its label to 1. In the same time, rules r_2 and r_4 are inhibited, while rules r_5 and r_6 are de-inhibited. The only rule we can use in the second step is r_5 which expels 0 out of membrane 1, inhibits itself and de-inhibits rule r_7. In this moment we have the following configuration of our system $[\ [\ \ ]_1 01]_s$. We now apply rule r_7 which transforms object 1 to λ on its way in the inner membrane and changes its label from 1 to 0. Rule r_7 de-inhibits the inhibited rule r_4, inhibits r_6 and itself, and de-inhibits rule r_8. The fourth step is the one in which the right answer 0 is sent out skin membrane, while the system gets back to its initial configuration.

Thus, our system gives the right answer, in four steps, when we have input 01. In the other two cases (when we have the input 01 and we start by using first the rule r_1, or the input is 00) our system performs the rules mentioned above; the details are left to the reader. □

Simulation of OR Gate

Lemma 3.2 *Boolean OR gate can be simulated by AID P systems with rules of types (b'_0) and (c'_0), using two membranes and two objects (only the input), in at most four steps.*

Proof. We construct the AID P system

$$\begin{aligned}
\Pi_{OR} &= (O, H, I, \mu, w_0, w_s, R), \text{with} \\
O &= \{0, 1\}, \\
\mu &= [\ [\ \ ]_0]_s, \\
w_0 &= w_s = \lambda, \\
H &= \{0, 1, s\}, \\
I &= \{r_i \mid 0 \leq i \leq 9\},
\end{aligned}$$

and the following set of R of rules:

$r_1 : [\ 1]_0 \to [\ \ ]_1 1$

$r_2 : [\ 1]_1 \to [\ \ ]_0 \lambda\{r_2, r_8\}$

$r_3 : [\ 0]_0 \to [\ \ ]_1 0\{r_2, r_4, r_5, r_6\}$

$r_4 : [\ 0]_1 \to [\ \ ]_0 \lambda\{r_2, r_8\}$

$r_5 : \neg[\ 1]_1 \to [\ \ ]_1 1\{r_5, r_7\}$

$r_6 : \neg[\ 0]_1 \to [\ \ ]_1 \lambda\{r_4, r_6, r_9\}$

$r_7 : \neg 0[\ \ ]_1 \to [\ \lambda]_0\{r_4, r_6, r_7, r_8\}$

$r_8 : \neg[\ 1]_s \to [\ \ ]_s 1\{r_2, r_8\}$

$r_9 : \neg[\ 0]_s \to [\ \ ]_s 0\{r_2, r_5, r_9\}$

As in the case of AND gate, we place initial values x_1 and x_2 in the membrane labeled 0 from the membrane structure. The succession of rules we apply for each case is (as expected due to the duality of the system) the following:
for $x_1x_2 = 00 - r_3, r_6, r_9$,
for $x_1x_2 \in \{01, 10\} - r_3, r_5, r_7, r_8$, or r_1, r_4, r_8, and
for $x_1x_2 = 11 - r_1, r_2, r_8$.

We only give here the details of the case when x_1 and x_2 are both 1. Our system has the following initial configuration: $[\ [\ 11]_0]_s$. As mentioned above, the only rule we can apply is r_1, and our system evolves to the following configuration: $[\ [\ 1]_1 1]_s$. The next rule we can apply is r_2 through which the object in membrane 1 is transformed into λ and the membrane label changes to 0, the system evolving to $[\ [\ \ ]_0 1]_s$. After applying rule r_2, rule r_8 is de-inhibited while rule r_2 is inhibited. We now can apply r_8, which sends out the skin membrane the answer 1 and restores the initial configuration of the system inhibiting rule r_8 and de-inhibiting rule r_2.

We showed how our systems expels, in three steps, the right answer, given the input 11.

The details of the behavior of the system in the other three cases are left to the reader. □

Simulation of NOT Gate

Lemma 3.3 *Boolean unary NOT gate can be simulated by AID P systems with rules of type* (b_0)*, in one step.*

Proof. We construct the AID P system

$$\begin{aligned} \Pi_{NOT} &= (O, H, S, \mu, w_s, R), \text{with} \\ O &= \{0, 1\}, \\ \mu &= [\ \ ]_s, \\ w_s &= x_1x_2, \\ H &= \{s\}, \\ S &= \{r_0, r_1\}, \\ R &= \{r_0 : [\ 0]_s \to [\ \ ]_s 1, r_1 : [\ 1]_s \to [\ \ ]_s 0\}. \end{aligned}$$

The correct computation of the NOT gate is obvious. □

4 Simulating Circuits

We give now an example of how to construct a global AID P system which simulates a Boolean circuit, designed for evaluating a Boolean function, using sub-AID P systems in it, namely including Π_{AND}, Π_{OR} and Π_{NOT} constructed in the previous section.

4.1 An Example

We take into consideration the example used in [8], namely we consider the function $f : \{0,1\}^4 \to \{0,1\}$ given by the formula $f(x_1, x_2, x_3, x_4) = (x_1 \wedge x_2) \vee \neg(x_3 \wedge x_4)$. Here is our circuit and its assigned membrane structure:

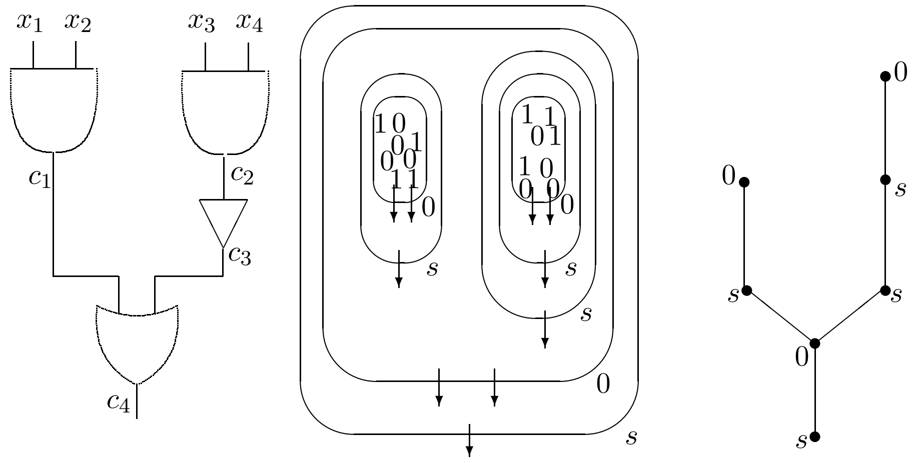

Fig. 2 A Boolean circuit and its associated membrane structure of simulation in AID P systems

As shown above, the circuit has a tree as its underlying graph, with the leaves as input gates, and the root as output gate.

We simulate this circuit with the P system $\Pi_C = (\Pi^{(1)}_{AND}, \Pi^{(2)}_{AND}, \Pi^{(3)}_{NOT}, \Pi^{(4)}_{OR})$ constructed from the distributed sub-AID P systems which work in parallel in the global P system, and we obtain an unique result in the following way:

1. for every gate of the circuit with inputs from input gates, we have an appropriate P system simulating it, with the innermost membrane containing the input values;
2. for every gate which has at least one input coming as an output of a previous gate, we construct an appropriate P system to simulate it by embedding in a membrane the "environments" of the P systems which compute the gates at the previous level.

For the particular formula $(x_1 \wedge x_2) \vee \neg(x_3 \wedge x_4)$ and the circuit depicted in Figure 2 we will have:

- $\Pi^{(1)}_{AND}$ computes the first AND$_1$ gate $(x_1 \wedge x_2)$ with inputs x_1 and x_2.
- $\Pi^{(2)}_{AND}$ computes the second AND$_2$ gate $(x_3 \wedge x_4)$ with inputs x_3 and x_4; these two P systems, $\Pi^{(1)}_{AND}$ and $\Pi^{(2)}_{AND}$, act in parallel.

– $\Pi^{(3)}_{NOT}$ computes NOT gate $\neg(x_3 \wedge x_4)$ with input $(x_3 \wedge x_4)$. While $\Pi^{(3)}_{NOT}$ is working, the output value of the first AND$_1$ gate performs the rules that can be applied (in $\Pi^{(4)}_{OR}$) and at a point waits for the second input (namely, the output of $\Pi^{(3)}_{NOT}$) to come.

– after the second input enters in the inner membrane of OR gate, P system $\Pi^{(4)}_{OR}$ will be able to complete its task. The result of the computation for OR gate (which is the result of the global P system), is sent into the environment of the whole system.

The idea we want to stress here is that, as noticed from the above explanations, our system has a self-embedded synchronization. By that, we mean that if either of the gates AND or OR receives only one (part of the) input from an upper level of the tree, the gate will wait for the other part of the input to come in order to expel the output. So, an extra synchronization system, as considered in [8], is not needed in AID P Systems.

Based on the previous explanations the following result holds:

Theorem 4.1 *Every Boolean circuit α, whose underlying graph structure is a rooted tree, can be simulated by a P system, Π_α, in linear time. Π_α is constructed from AID P systems of type Π_{AND}, Π_{OR} and Π_{NOT}, by reproducing in the architecture of the membrane structure, the structure of the tree associated to the circuit.*

Proof. The statements follows from the previous considerations, with the observation that the simulation lasts at most four times the duration of evaluating the Boolean circuit. □

Property 4.1 *Any Boolean circuit α, with n gates, can be simulated using at most $2n$ membranes.*

Proof. Let us consider the worst case in which our circuit contains only OR and AND gates. Then it is obvious that for the individually simulation of these gates we use $2n$ membranes(every gate is simulated using 2 membranes). In fact, this coincides with the upper bound stated in the property due to the embedded synchronization and the fact we do not need additional membranes in order to simulate it. □

5 `CIRCUIT-SAT` Efficiency

There is an interesting computational problem related to circuits, called `CIRCUIT-SAT`. Given a circuit C, is there a truth assignment T appropriate to C such that $T(C) = true$? It is easy to argue that `CIRCUIT-SAT` is computationally equivalent to `SAT`, and thus presumably NP-Complete.

We can now appeal to a well-known construction (see, e.g. [17]) to reduce a `CIRCUIT-SAT` instance to a CNF formula. Given a circuit C, we will construct a CNF formula φ_C such that there is an assignment to the inputs of C producing a 1 output iff the formula φ_C is satisfiable. The formula φ_C will have $n+|C|$ variables, where $|C|$ denotes the number of gates in C; if C acts on inputs $x_1, \cdots, x_n$ and contains gates $g_1, \cdots, g_{|C|}$, then φ_C will have variable set $\{x_1, \cdots, x_n, g_1, \cdots, g_{|C|}\}$. For each gate $g \in C$, we define a set of clauses as follows:

1. if $c = \mathrm{AND}(a, b)$, then add $(\neg c \vee a), (\neg c \vee b), (c \vee \neg a \vee \neg b)$,
2. if $c = \mathrm{OR}(a, b)$, then add $(c \vee \neg a), (c \vee \neg b), (\neg c \vee a \vee b)$,
3. if $c = \mathrm{NOT}(a)$, then add $(c \vee a), (\neg c \vee \neg a)$.

The formula φ_C is simply the conjunction of all the clauses over all the gates of C.

We assume below that C consists of gates from a standard complete basis such as AND, OR, NOT. Our results can easily be generalized to allow other gates (e.g. with a larger fan-in); the final bounds are interesting as long as the number of clauses per gate (and the maximum fan-in in the circuit) is upper bounded by a constant. Recall that a circuit C is a directed acyclic graph (DAG). We define the underlying undirected graph as G_C:

Definition 5.1 *Given a circuit C with inputs $X = \{x_1, \cdots, x_n\}$ and gates $S = \{g_1, \cdots, g_s\}$, let $G_C = (V, E)$ be the undirected and unweighted graph with $V = X \cup S$ and $E = \{\{x, y\} \mid x$ is an input to gate y or vice versa$\}$.*

Theorem 5.1 *For a circuit C with gates from $\{AND, OR, NOT\}$, the* `CIRCUIT-SAT` *instance for C can be solved by an AID P system.*

Proof. Here is the sketch of the proof.

We know that propositional formula φ_C in CNF is simply the conjunction of all the clauses over all the gates of C. In our previous example, for the Boolean circuit considered in Section 4, φ_C is:

$$\begin{aligned}\varphi_C = \ & (\neg c_1 \vee x_1) \wedge (\neg c_1 \vee x_2) \wedge (c_1 \vee \neg x_1 \vee \neg x_2) \wedge \\ & (\neg c_2 \vee x_3) \wedge (\neg c_2 \vee x_4) \wedge (c_2 \vee \neg x_3 \vee \neg x_4) \wedge \\ & (c_2 \vee c_3) \wedge (\neg c_2 \vee \neg c_3) \wedge \\ & (\neg c_1 \vee c_4) \wedge (\neg c_3 \vee c_4) \wedge (\neg c_4 \vee c_1 \vee c_3).\end{aligned}$$

There are already known algorithms which solve `SAT` (written as Boolean propositional formula in CNF) with P systems with active membranes (see [1, 2, 7, 14, 15, 19]). Then our φ_C can be solved easily following the proof ideas from these papers.

We leave the technical details of the proof for the reader. □

6 A DNA-like Proposal to Solve SAT

For a better understanding of the proposed system we start first with an example and then we give the general details.

Let us begin with the example promised in Subsection 2.4, which was first considered in [13] and later mentioned in [21]. Starting from this example we will make a parallelism between the classical DNA way of solving satisfiability and the 'DNA-like' way of solving it with P systems using the tools of merging and separation of the membranes, and polarization.

Consider the propositional formula

$$\alpha = (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2).$$

Thus, we have two variables with the following corresponding graph:

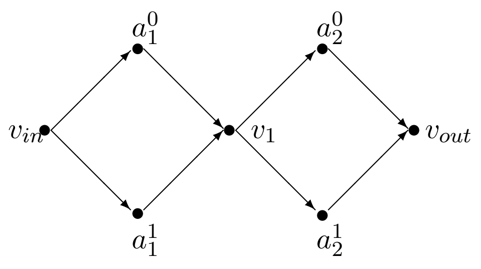

Fig. 3 The graph associated to formula α

As mentioned in Section 2.4, each of the 4 paths through this graph corresponds to one of the 4 truth assignments for the variables x_1 and x_2. The core of the procedure of solving SAT with DNA strands is the operation of *separation*. Let us denote by N_0 the initial test tube which contains all 4 paths (strands) from the initial to the final vertex of the graph. If we apply the operation separate, forming the test tube $+(N_0, a_1^1)$, we get those truth assignments where x_1 assumes the value 1 (true).

A truth assignment is denoted by two-bit sequence in the natural way. Thus, 01 stands for the assignment $x_1 = 0$, $x_2 = 1$. Similar notation is also used if there are more than two variables. This simple notation of bit sequences is extended to the DNA strands resulting from our basic graphs. Thus, the strand $v_{in}a_1^0v_1a_2^1v_{out}$ is denoted simply by 01.

Following the idea in [21], we denote by $S(N, i, j)$ the test tube of such strands in N, where the ith bit equals j, $j = 0, 1$. Thus, as we observed above, $S(N, i, j)$ results from N by the operation *separate*:

$$S(N, i, j) = +(N, a_i^j).$$

The tube of such strands in N, where the ith bit equals the complement of j is also considered:

$$S^-(N, i, j) = -(N, a_i^j).$$

Here is the algorithm of solving SAT for the propositional formula α:

```
(1)  $input(N_0)$
(2)  $N_1 = S(N_0, 1, 1)$
(3)  $N_1' = S^-(N_0, 1, 1)$
(4)  $N_2 = S(N_1', 2, 1)$
(5)  $merge(N_1, N_2) = N_3$
(6)  $N_4 = S(N_3, 1, 0)$
(7)  $N_4' = S^-(N_3, 1, 0)$
(8)  $N_5 = S(N_4', 2, 0)$
(9)  $merge(N_4, N_5) = N_6$
(10) $detect(N_6)$
```

The program is based on exhaustive search. The initial tube at step (1) contains all possible truth-assignments. The test tube at step (5) contains the assignments satisfying the *first clause* of the propositional formula α. (Either x_1 or x_2 must assume the value 1. At step (2) we have those assignments for which x_1 is 1. Of the remaining ones we still take, at step (4), those for which x_2 is 1.) The assignments in this tube, N_3, are filtered further to yield at step (9) those assignments that also satisfy the second clause of the propositional formula α.

Let us consider now the same propositional formula α and solve it using the new technique we propose here, namely P systems using polarizations and separation/merging rules following closely the principle of separation/merging as above.

We start our computation having in one membrane, labeled $(1,1)$ all truth-assignments plus the clauses of the given problem (α) encoded as follows:

– truth-assignments

$$\begin{aligned} &00\ - a^0_{1,1}, a^0_{1,2}, \quad 01 - a^0_{2,1}, a^1_{2,2},\\ &10\ - a^1_{3,1}, a^0_{3,2}, \quad 11 - a^1_{4,1}, a^1_{4,2}. \end{aligned}$$

More precisely, $a^0_{1,2}$ says that first position - (1) in the subscript indicates the truth-assignment, the second position - (2) indicates the place in the assignment of the value indicated by the superscript - (0).

– clauses

$$\begin{aligned} &(x_1 \vee x_2) - x^1_{1,1}, x^1_{1,2},\\ &(\neg x_1 \vee \neg x_2) - x^0_{2,1}, x^0_{2,2}; \end{aligned}$$

Here by $x^1_{1,2}$ we understand that in the first instance of the formula (1), variable x_2 (2) is not negated (1).

In the given example, one can imagine 12 objects (given by the sum of the 4 truth-assignments (8 objects) and 4 variables of the propositional formula) of two types, a and x, floating in membrane labeled $(1,1)$; but, at any time, picking any of these objects we can precisely state which is its value and position/value in the set of string of truth-assignments/clauses. In other words, a more clear image is that of some "strings" of two types floating inside that cell (membrane), an image very close to the one of the initial test tube N_0 having all truth-assignments encoded as we previously saw.

This last image is also very close to the biological image of DNA in an eukaryotic cell enclosed in the *nuclear envelope* through *inner nuclear membrane* and *outer nuclear membrane.*

Coming back to our example, we want now to separate the membrane labeled $(1,1)$ having the polarization 1 into two membranes one containing the truth assignments which have 1 on the first position (10 and 11, encoded as $a^1_{3,1}, a^0_{3,2}$, and $a^1_{4,1}, a^1_{4,2}$, respectively), the variable which is on the first position on the first clause (namely $x^1_{1,1}$ and the variables from the other clause ($x^0_{2,1}$, and $x^0_{2,2}$), while the second contains the rest of the truth-assignments (00 and 01) plus the second variable of the first clause ($x^1_{1,2}$). This step simulates the steps (2) and (3) from the DNA variant of the example. (Fig. 4 shows how the example is processed by the two techniques in parallel.)

This is done by applying rule r_1:

$r_1 : [\ ]^1_{1,1} \to [\ X_{1,1}]^0_{1,1}[\ X_{1,2}]^1_{1,2}$, where

$X_{1,1} = \{11, 10, \neg x_1, \neg x_2, x_1\} = \{a^1_{3,1}, a^0_{3,2}, a^1_{4,1}, a^1_{4,2}, x^1_{1,1}, x^0_{2,1}, x^0_{2,2}\}$,

$X_{1,2} = \neg X_{1,1} = \{x_2\}$.

(We recall the reader that in showing our procedure we start from an example and only after it we define the general framework.)

```
 (1) $input(N_0)$
     ($1_P$) input membrane labeled $1,1$
 (2) $N_1 = S(N_0, 1, 1)$
 (3) $N'_1 = S^-(N_0, 1, 1)$
     ($2_P$) separation of membrane labeled $1,1$
 (4) $N_2 = S(N'_1, 2, 1)$
     ($3_P$) separation of membrane labeled $1,2$
 (5) $merge(N_1, N_2) = N_3$
     ($4_P$) merge between membranes $1,1$ and $1,2$ to membrane $2,1$
 (6) $N_4 = S(N_3, 1, 0)$
 (7) $N'_4 = S^-(N_3, 1, 0)$
     ($5_P$) separation of membrane $2,1$
 (8) $N_5 = S(N'_4, 2, 0)$
     ($6_P$) separation of membrane $2,2$
 (9) $merge(N_4, N_5) = N_6$
     ($7_P$) merge between membranes $2,1$ and $2,2$ to membrane $3,1$
(10) $detect(N_6)$
     ($8_P$) detect if there is at least a solution
```

Fig. 4 Simulation of DNA (rules (1)-(10)) and DNA-like (rules (1_P)-(8_P)) models in parallel.

We continue the computation by separating membrane labeled $(1,2)$ with polarization 1 into two membranes labeled $(1,2)$ and $(1,3)$ with polarizations 0 and 1, respectively. First membrane will contain the truth-assignments that have 0 on the first position and 1 on the second position (so, only 01) plus the variable x_2 from the first clause. The second membrane is the negation of the above one, thus containing only the truth-assignment 00.

Here is schematic way of solving the problem (by r_1 we mean the application of the general rule r_1, etc.):

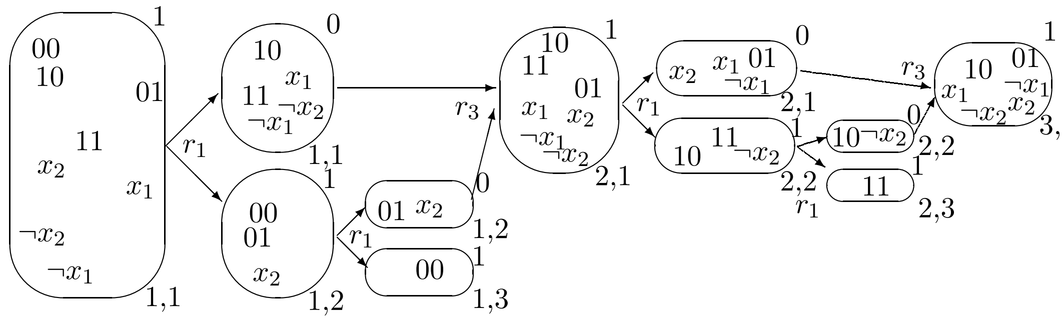

Fig. 5 Schematic representation of solving α

In the next step membranes labeled $(1,1)$ and $(1,2)$ will merge and form membrane $(2,1)$. The rule of merging is given below:

$$r_3 : [\ ]^0_{1,1}[\ ]^0_{1,2} \rightarrow [\ ]^1_{2,1}.$$

Membrane labeled $(2,1)$ contains strings 10, 11, 01 plus variables x_1, x_2, $\neg x_1$, and $\neg x_2$.

In this phase of the computation, our procedure has computed the *first clause* of the formula and continues with the second one.

From membrane labeled $(2,1)$ we separate using the rule r_1 (applicable to membrane $(2,1)$) those truth-assignments which begin with 0 from those that do not begin with 0. Thus, membrane labeled $2,1$ (with polarization 0) will contain truth-assignment 01 and variables x_1, x_2, and $\neg x_1$, while membrane labeled $2,2$ will contain the rest of the objects (namely, truth-assignments 10 and 11 plus variable $\neg x_2$).

In this step we separate the second membrane produced in the previous step into two membranes: one (with polarization 0) containing the truth-assignments that have 0 on the second position (so only 10) and variable $\neg x_2$, and the other one containing only the string 11. We now merge the two last membranes having polarization 0 ($(2,1)$, and $(2,2)$), thus completing the 7-th step of the computation.

One can notice that the answer to our particular problem floats in the membrane labeled $(3,1)$ (produced by the union of membranes labeled $(2,1)$ and $(2,2)$). So, now, we are in the stage of *detecting* the result of our problem.

For detection, we will use the following rules:

$$r_4 : [\ a^k_{i,j}]^1_{3,1} \rightarrow [\ ]^1_{3,1}1,$$
(where $a^k_{i,j} \in \{a^0_{2,1}, a^1_{2,2}, a^1_{3,1}, a^0_{3,2}\}$ is non-deterministically chosen)
$$r_5 : [\ ]^1_{1,4}[\ ]^1_{2,4} \rightarrow [\ ]^1_{2,4},$$
$$r_6 : [\ 1]^0_{0,0} \rightarrow [\ ]^1_{0,0}yes.$$

In the last step of the computation, which is done through rule r_6, the correct answer *yes* was sent to the environment, meaning that our problem has at least a solution.

The example we considered here has, as we saw, at least one solution to the given problem. In the general case, if the there is no solution to the given problem, the system will expel to the environment the answer *no*.

The problem with this solution is that we need first to produce 2^n truth-assignments for variables. This can be done in various way in membrane computing – see, e.g., [16],

[1], [15] – by using membrane division, separation, etc. However, the respective truth-assignments are obtained in 2^n separate membranes. By merging operations, we could put together these truth-assignments in a single membrane, but, in order not to "mix" them, we need to encode them separately. This, however, assumes using an exponential number of objects, and thus the system itself has an exponential size. For the moment, we do not know how to overcome this difficulty.

7 Final Remarks

In this paper we have introduced a new way of simulating Boolean gates and circuits, as an answer to a question formulated in [6]: simulate Boolean circuits with P systems using the inhibiting/de-inhibiting controlling mechanism of computation, as introduced and investigated in [6, 7]. This idea is very attractive because apart from using less biological resources (only two objects and two types of rules for the simulation of Boolean gates) than the previous simulations, we also proposed a system which has a self-embedded synchronization of the objects in the circuit without being necessary other systems to coordinate the computation.

We have also proposed here an approach to solving `SAT` problem simulating, in P systems with active membranes, the way this problem is effectively solved in laboratories using DNA strands. Technical details of this proposal still remains to be fixed, but we hope that this is a step ahead in our way to the laboratory. In addressing the mentioned problem we have used polarized/non-polarized P systems, while membranes are capable of merge/separate, changing or not-changing their labels. We found very natural to compare (and study the computational bridge between the) two notions from Natural Computing both having the tools of merging/separation already defined. Actually, such an attempt was already done in [11], but using different protocols of DNA computing and different operations with membranes in P systems.

In the end we propose the reader to investigate, using the new tools (uniformly way of solving `SAT`, but following Lipton's algorithm) presented above, other NP-hard problems, or to try to solve the proposed problem using P systems with different features.

References

[1] A. Alhazov, T.-O. Ishdorj. Membrane Operations in P Systems with Active Membranes, In: Gh. Păun, et all (eds.) *Second Brainstorming Week on Membrane Computing*, Sevilla, 2-7 February, 2004, Research Group in Natural Computing **TR 01/2004**, University of Sevilla, 37–44.

[2] A. Alhazov, L. Pan, Gh. Păun. Trading Polarizations for Labels in P Systems with Active Membranes, *Acta Informatica*, 41, 2-3 (2004), 111–144.

[3] A. Alhazov, L. Pan. Polarizationless P systems with active membranes, *Grammars*, 7 (2004), 141–159.

[4] R. S. Braich, N.Chelyapov, C. Johnson, P. W. K. Rothemund, L. Adleman. Solution to a 20-variable 3-SAT problem on a DNA computer. *Science*, 296, 5567 (2002), 499–502.

[5] C. Calude, Gh. Păun, G. Rozenberg, A. Salomaa eds. *Multiset Processing*, LNCS 2235, Springer-Verlag, Berlin, 2001.

[6] M. Cavaliere, M. Ionescu, T.-O. Ishdorj. Inhibiting/De-inhibiting Rules in P Systems, *LNCS*, 3365, 224–238, 2005.

[7] M. Cavaliere, M. Ionescu, T.-O. Ishdorj. Inhibiting/De-inhibiting P Systems with Active Membranes, *Cellular Computing (Complexity Aspects)*, ESP PESC Exploratory Workshop, Fénix Editora, Sevilla, 117–130, 2005.

[8] R. Ceterchi, D. Sburlan. Simulating Boolean Circuits with P Systems, Workshop on Membrane Computing WMC-Tarragona 2003, (A. Alhazov, C. Martín-Vide, G. Păun, eds), *LNCS*, 2933, 104–122, 2004.

[9] J. Dassow, Gh. Păun. *Regulated Rewriting in Formal Language Theory*, Springer-Verlag, Berlin, 1989.

[10] M.R. Garey, D.J. Johnson. *Computers and Intractability. A Guide to the Theory of NP-Completeness*. W. H. Freeman, San Francisco, 1979.

[11] T. Head. Aqueous Simulations of Membrane Computations, *Romanian Journal of Informatics Science and Technology*, 5, 4 (2002).

[12] M. Ionescu, T.-O. Ishdorj. Replicative - Distributed Rules in P Systems with Active Membranes, *Proceedings of First International Colloquium on Theoretical Aspects of Computing*, Guiyang, China, 20-24 September 2004, UNU/IIST Report No. 310, 263–278, and *LNCS* 4705, 69–84, 2005.

[13] R.J. Lipton. Using DNA to solve NP-complete problems, *Science*, 268 (1995), 542–545.

[14] L. Pan, A. Alhazov, T.-O. Ishdorj. Further Remarks on P Systems with Active Membranes, Separation, Merging and Release Rules, *Soft Computing*, 8 (2004), 1–5.

[15] L. Pan, T.-O. Ishdorj. P Systems with Active Membranes and Separation Rules, *Journal of Universal Computer Science*, 10, 5 (2004), 630–649.

[16] L. Pan, A. Alhazov. Solving HPP and SAT by P Systems with Active Membranes and Separation Rules, *IEEE Transactions on Computers*, submitted, 2005.

[17] Ch. P. Papadimitriou. *Computational Complexity*, Addison-Wesley, Reading, MA, 1994.

[18] Gh. Păun. Computing with Membranes, *Journal of Computer and System Sciences*, 61(1), 108–143, 2000, and *TUCS Research Report*, 208, 1998 (http://www.tucs.fi).

[19] Gh. Păun. P Systems with Active Membranes: Attacking NP-Complete Problems, *Journal of Automata, Languages and Combinatorics*, 6, 1 (2001), 75–90.

[20] Gh. Păun. *Membrane Computing: An Introduction*, Springer-Verlag, Berlin, 2002.

[21] Gh. Păun, G. Rozenberg, A. Salomaa. *DNA Computing. New Computing Paradigms*, Springer-Verlag, Berlin, 1998.

[22] M.J. Pérez-Jiménez, A. Romero-Jiménez, F. Sancho-Caparrini, Complexity Classes in Models of Cellular Computation with Membranes, *Natural Computing*, 2, 3 (2003), 265–285.

[23] K. Sakamoto, H. Gounzu, K. Komiya, D. Kiga, S. Yokoyama, T. Yokomori, M. Hagiya. Molecular computation by DNA hairpin formation, *Science*, 288 (2000), 1223–1226.

[24] A. Salomaa. *Formal Languages*, Academic Press, New York, 1973.

[25] A. Salomaa, G. Rozenberg eds. *Handbook of Formal Languages*, Springer-Verlag, Berlin, 1997.