A Gentle Introduction to Membrane Systems and Their Computational Properties

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

Dipartimento di Informatica, Sistemistica e Comunicazione
Università degli Studi di Milano-Bicocca
Viale Sarca 336, Building U14, 20126 Milano, Italy

leporati/luca.manzoni/mauri/porreca/zandron@disco.unimib.it

1 Introduction

The theory of computation investigates the nature and properties of algorithmic procedures. This field emerged in the 20s and 30s of the 20th century from the work on the philosophy and the foundations of mathematics. Of great importance and inspiration was David Hilbert's ambitious program to "dispose of the foundational questions in mathematics once and for all" [34], that led to fundamental results in logic such as Gdel's incompleteness theorems [6], and ultimately to the birth of recursion theory (nowadays mostly referred to as computability theory) and computer science itself.

The formal notion of computability that is almost universally adopted today is due to Alan Turing, who introduced in his ground-breaking paper *On computable numbers, with an application to the Entscheidungsproblem* [32] a simple, elegant and convincing mathematical formalisation of the notion of computation, as it is carried out by a human executor equipped with enough scratch paper. Turing's work showed that, as long as we accept his notion of computation, there exist well-formed mathematical questions whose answer cannot be computed. In particular, one of the main challenges of Hilbert's program, the *Entscheidungsproblem* (finding a decision procedure for the validity of statements in first-order logic) was proved to be unsolvable.

This formalisation, that rapidly became known as the *Turing machine*, is still the reference model for computing devices in theoretical computer science, as it also enjoys the property of being a good model of actual electronic computers; this is also due to the fact that it was itself an inspiration for the design of automatic computing machinery [5].

With the development of computers as a technology, being able to solve a particular problem proved not to be satisfying: *fast*, *efficient* solutions are needed. This led to the development of computational complexity theory, pioneered [9] by Hartmanis and Stearns in the paper *On the computational complexity of algorithms* [12], that also gives the name to the field. Identifying

the notion of "efficient" with "polynomial-time computable" is due [9] to Edmonds [7], while the central question of complexity theory, whether $\mathbf{P} = \mathbf{NP}$, arose from the work of Cook [4] and Karp [16]. This question has shaped the whole development of the field, and still remains open today.

However, the theory of computation is not entirely about Turing machines. Several authors sought to draw inspiration from the way nature "computes" in order to define alternative, unconventional computing models, or, from the opposite point of view, to interpret natural phenomena as computation [1]. For instance, artificial neural networks [21] are inspired by the functioning of neurons in the brain, and genetic algorithms seek to solve computationally hard problems by simulating the processes of mutation, mating and natural selection. A clear example of biological inspiration is given by DNA computing, which provided an actual *in vitro* implementation of an algorithm for the Hamilton path problem [2] (the theory was initiated a few years earlier, particularly by Head [13]).

Inspired by the work on DNA computing [26], Pun introduced in 1998 [24] the notion of *membrane systems*, initially called *super-cells*, and nowadays usually known as *P systems*. Here the computing device is an abstraction of biological cells. Unlike in neural networks, we do not deal with cells as atomic objects: as the name suggests, the focus is on the membranes that define the cells and their internal compartments, which work together by performing different individual functions. The chemical environment of the various compartments are described in terms of *multisets* of symbols (i.e., sets in which each symbol has a multiplicity). Another defining feature of P systems (as they were initially defined) is *maximal parallelism*: as many operations as possible are carried out simultaneously, and no part of the system remains inactive if it can carry out some part of the computation.

Although P systems have also been investigated from the perspectives of bioinformatics and systems biology, where they might be used as models of biological phenomena in computer simulations, most of the research in membrane computing has been carried out from a language-theoretic (including the seminal paper [24]), computability-theoretic and complexity-theoretic standpoint.

In the literature, many variants using various kinds of rules and different features have been considered and investigated in this respect. For example, the notion of multiset rewriting employed in the original model of P system [24] is context-sensitive (or *cooperative*, which is the term normally used in membrane computing). Other classes of P systems, however, possess other features that make them extremely powerful from a computational perspective: as a consequence, the use of *context-free* (or *non-cooperative*) rewriting rules has also been considered. Here, for the sake of explanation, we will present a simplified model using only basic ingredients and simple cooperative rewriting rules. After presenting some basic definitions, we will first show that such systems are Turing complete, by simulating register machines, a well known universal computing device, and we will show that direct simulation of Turing machines is time efficient. Then, by exploiting the possibility of creating new membranes by division of existing ones, we will show how to solve all problems from the complexity classes **NP** and **PSPACE** in polynomial time (and exponential space) by means of membrane systems. At the end of the chapter, we will shortly discuss how cooperation can be avoided, and we will provide pointers to the literature concerning main variants considered in the framework of membrane computing.

2 Membrane structures

The main feature of all variants of P systems is their *membrane structure*. Several variants have been proposed in the literature [27], but we consider here the original *cell-like* shape [24]: a hierarchy of membranes nested to an arbitrary depth. There is a clear bijective mapping between the set of membranes and the *regions* they define, delimited by the membrane itself from the outside, and any membrane immediately included in it from the inside. The outermost membrane (sometimes called the *skin*) separates the actual P system from the surrounding *environment*, which is a further implied region also playing a role in the computation. Membranes not containing further membranes inside them are called *elementary*; the others are called *nonelementary*.

The abstract shape of a membrane structure in any given instant is mathematically represented by a *rooted, unordered tree*. Here the membranes (equivalently, the regions they delimit) correspond to vertices, the outermost membrane being associated with the root of the tree, and an edge between two vertices indicates that the corresponding membranes are located one immediately inside the other; the "parent" and "child" roles are implied by the distance from the root. The leaves of the tree correspond to elementary membranes, while the internal nodes correspond to the nonelementary ones. The tree is unordered (i.e., there is no distinguishable "first" or "leftmost" child) because we do not keep track of any spatial relation between the membranes other than simple containment[1].

Membranes are not only set apart by their position in the membrane structure, but also by the role they supposedly perform. This is represented in P systems by attaching a *label* taken from a finite set Λ to each membrane (formally, to the vertex representing it in the corresponding unordered tree). As we shall see later, different labels correspond to different sets of rules that can be applied to the membranes or their contents. Initially, each membrane must be given a different label (although two distinct labels might be associated to the same set of rules), while the process of membrane division may create multiple membranes sharing the same label.

A membrane structure is traditionally described in a "linear" notation by means of a string of balanced brackets. Given the unordered rooted tree corresponding to the membrane structure, fix an arbitrary ordering of the children of each node. The resulting ordered tree, ignoring the node labels, is then uniquely interpreted as a string of balanced brackets belonging to the language defined by the following context-free grammar, beginning with the nonterminal S_1:

$$S_1 \to [S_2] \qquad\qquad S_2 \to [S_2] \mid S_2 S_2 \mid \epsilon.$$

These are all the non-empty strings of balanced brackets having a single outermost pair. The nesting of brackets is induced by the shape of the tree (and obviously corresponds to the containment relation of the original membrane structure). Finally, each bracket is subscripted by the label of the corresponding membrane. An example of membrane structure, represented in a pictorial form, as a tree, and in bracket notation, can be found in Figure 1.

[1] There exist other variants of P systems, such as the *spatial* ones, where the positions of membranes and objects do actually matter [3].

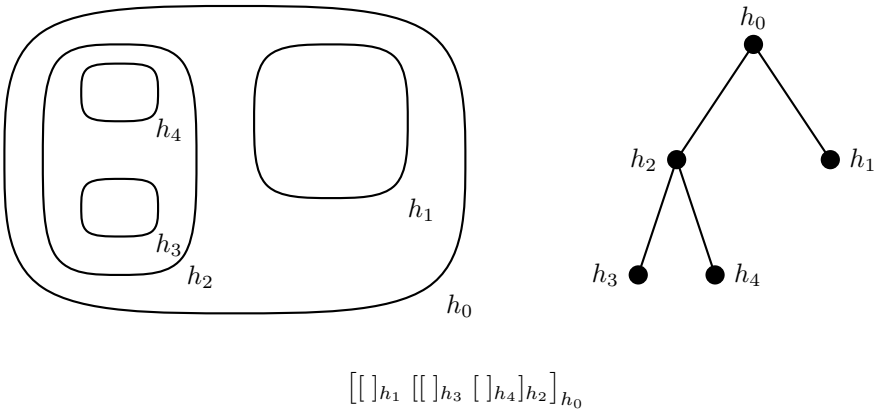


Figure 1: A membrane structure, and two formalisms representing it.

3 The content of regions

A single molecule can be represented abstractly as a symbol taken from an alphabet Γ. However, subsets of Γ are not an adequate description of a whole chemical environment, as they do not carry any information related to the (absolute or relative) quantities of molecules of the same species. Since the concentrations (that is, amounts) of chemicals in a region do actually matter from a metabolic standpoint, we use *multisets* to describe them.

A multiset w over a set Γ is simply a function $w\colon \Gamma \to \mathbb{N}$, mapping each object to its multiplicity. Generalisations of the usual set-theoretic operations can be defined on the sets of multisets over Γ; for instance, the union of multisets w_1 and w_2 is the multiset where, for all $a \in \Gamma$, the object a has multiplicity $w_1(a) + w_2(a)$. From an algebraic point of view, the set of multisets over Γ, with respect to the operation of union, is the *free commutative monoid on* Γ. Compare this to the set $\Gamma^\star$ of strings, which is the *free monoid on* Γ: here the operation (i.e., string concatenation) is *not* commutative. When describing the chemical environment of a region, we are interested in the number of molecules, but not in any particular ordering of them.

Another equivalent characterisation can be given in terms of equivalence classes of strings. For each string $u \in \Gamma^\star$ and each symbol $a \in \Gamma$, let $|u|_a$ be the number of occurrences of a in u. Now let u and v be strings, and let the binary relation $\simeq$ be defined by

$$u \simeq v \qquad \text{if and only if} \qquad |u|_a = |v|_a \text{ for all } a \in \Gamma.$$

It is trivial to prove that $\simeq$ is an equivalence relation, and in particular a congruence relation with respect to the operation of string concatenation. Then, the set of equivalence classes $\Gamma^\star/\simeq$ (together with the operation induced by concatenation) is isomorphic to the set of multisets over Γ (together with the operation of multiset union). This provides us with a simple way to denote multisets in writing. Let w be a multiset, and let u be any string such that $w(a) = |u|_a$ for all $a \in \Gamma$; then, we shall identify w with $[u]_{/\simeq}$ (the equivalence class of u modulo $\simeq$) and, furthermore, we shall drop the equivalence class notation, simply writing u for w, as the formal meaning of the symbol is implied;

correspondingly, multiset union will be written as string concatenation. This identification justifies the following (informal) sentence, which is usually found in the P systems literature: "a multiset w will be represented as a string u, together with all its permutations", since permuting the symbols of u does not change the number of their occurrences.

4 Computation rules

The elements we have described until now, i.e., the membrane structure and the multisets contained in its regions, define the instantaneous configurations of P systems. A configuration evolves by means of computation steps, where in each step some rules inspired by the biology of cells are applied.

4.1 Object evolution rules

A first kind of computation rule describes simple chemical reactions taking place inside a region. Consider, as an example, the following formula for the burning of methane as it may be described in a chemistry textbook:

$$CH_4 + 2O_2 \rightarrow CO_2 + 2H_2O.$$

This notation describes how a molecule of methane (CH_4) reacts with two molecules of oxygen (O_2) yielding a molecule of carbon dioxide (CO_2) and two molecules of water (H_2O).[2] The molecules on the left-hand side of the arrow are called *reactants* or *reagents*, while those on the right-hand side are called *products*.

This kind of reaction corresponds quite closely with a limited form of *multiset rewriting*, where a certain submultiset is replaced by another one. The notation is also quite similar, as for multisets we shall write $u \rightarrow v$ to indicate that multiset u is replaced by v. The general notion of multiset rewriting is more abstract, and it usually disregards the law of conservation of mass. A general chemical reaction viewed as a multiset rewriting rule possesses a powerful feature from a language-theoretic perspective: it is *context-sensitive*. For instance, methane may only burn if oxygen is also present, and we require at least *two* molecules of oxygen for each molecule of methane.

The first kind of computation rules used by P systems, called *object evolution rules* or rules of type (a), are thus denoted as follows:

$$[u \rightarrow v]_h$$

This rule can be applied inside each membrane labelled by $h \in \Lambda$, having a multiset z which contains the multiset u, i.e. $u \subseteq z$. When the rule is applied, the multiset of objects defined by u is removed from z and is replaced by the multiset v.

4.2 Communication rules

The main role of cell membranes is that of delimiting and separating regions where different functions are performed. However, these distinct regions must

[2]The molecules involved in this reaction are, in turn, made of several atoms; however, in this discussion we consider compound objects such as H_2O as indivisible.

also cooperate: for instance, the products of a chemical reaction occurred in membrane h_1 might be stored inside membrane h_2. The cell membranes are thus *selectively permeable*, allowing specific molecules to move between regions.

First consider the case when a multiset is absorbed by a membrane. This kind of rule is called *send-in communication rule*, or type (b) rule, and is formally written as:

$$u\ [\ ]_h \to [v]_h$$

This rule can be applied to a membrane having label $h \in \Lambda$; the region located outside the membrane must also contain at least an instance of multiset $u \in \Gamma^*$. The effects of this rule is that the multiset u is removed from the outside region, and the multiset $v \in \Gamma^*$ appears inside the membrane. As a restriction, it is assumed that no object can be brought into the outermost membrane of the P system from the external environment.

The converse kind of rule describes the process by which a membrane expels a multiset; these are called *send-out communication rules* or type (c) rules:

$$[u]_h \to [\ ]_h\ v$$

This rule can be applied to a membrane labelled by $h \in \Lambda$ containing an instance of the multiset u. By applying this rule, the system removes u from the membrane, and places the multiset $v \in \Gamma^*$ into the region immediately outside the membrane.

4.3 Division rules

We have already seen that the membrane structure of a cell needs not remain unchanged during its entire lifespan. In fact, new membranes can be created. The quintessential example is given by the process of mitosis, whereby a whole copy of the original cell is produced by division. The process of division also occurs *internally* in cells (that is, without involving the outermost cell membrane), as mitochondria also divide by binary fission [17]. In general, division allows the creation of new "processing units" when the need arises.

The corresponding *division rules*, or type (d) rules, have the following form:

$$[u]_h \to [v]_h\ [w]_h$$

A rule of this kind can be applied to a membrane labelled by $h \in \Lambda$ and containing at least one instance of the multiset $u \in \Gamma^*$. Its effect is as follows: a whole new copy of the membrane is created and placed into the same region as the original one; the contents are also copied, with the exception of the instance of u mentioned above, which is replaced by an instance of $v \in \Gamma^*$ in one of the two resulting membranes, and by $w \in \Gamma^*$ in the other one. This kind of rule cannot be applied to the outermost membrane.

Notice that the membrane h can be elementary or nonelementary: the difference is usually underlined by calling the corresponding rules *elementary membrane division* and *nonelementary membrane division*, respectively. In case h is a nonelementary membrane, all the membranes contained in it (as well as their content) are copied in both of the new instances of h.

5 A formal definition

We are now able to give a precise definition of our model of computation of interest.

Definition 1. A *P system* of initial degree $d \geq 1$ is a structure $\Pi = (\Gamma, \Lambda, \mu, w_{h_1}, \ldots, w_{h_d}, R)$ where

- Γ is the alphabet;
- Λ is the finite set of labels;
- μ is the membrane structure, a rooted undirected tree of d nodes labelled in a one-to-one way by elements of Λ;
- $w_{h_1}, \ldots, w_{h_d}$, with $h_1, \ldots, h_d \in \Lambda$, are multisets over Γ;
- R is a finite set of rules.

For each $h \in \Lambda$, the multiset w_h describes the objects initially located in the region delimited by the membrane labelled by h.

5.1 Configurations and computations

An instantaneous *configuration* of Π is given by the rooted unordered tree describing its current membrane structure, augmented as follows: each node is labelled by a couple (h, w) where

- $h \in \Lambda$ is the label of the corresponding membrane;
- w is the multiset over Γ contained in the membrane.

A computation step changes the current configuration by applying the rules in a *maximally parallel way*, according to the following principles.

- Each membrane can be subject to at most one communication or division rule per step. Any number of evolution rules can be simultaneously applied inside the same membrane, except for the restrictions on objects described below.
- Each object can be subject to at most one rule per step.
- A maximal combination of rules must be applied during each step. Each object that appears on the left-hand side of an applicable evolution, communication, or division rule must be subject to exactly one of them. We say that a rule is "applicable" if the label of the membrane containing the object corresponds to those appearing on the left-hand side of the rule. The only objects that do not evolve are those associated to no applicable rule (or to no rule at all).
- Each membrane that appears on the left-hand side of an applicable (i.e., having the correct labels on the left-hand side) communication or division rule must be subject to exactly one of them. The only membranes that do not evolve are those associated to no applicable rule (or to no rule at all).

- When several maximally parallel combinations of rules can be applied at the same time, a nondeterministic choice between them is made. As a consequence, in the general case multiple possible configurations can be reached by performing a computation step.
- After a legitimate maximally parallel combination of rules has been chosen (i.e., every membrane and object that can evolve has been assigned to a rule), the rules are applied in a logical bottom-up (depth-first) way, from the elementary membranes towards the outermost one. Inside each membrane in turn, first all object evolution rules are applied, then the remaining communication or division rule.

The *maximal parallel* way of applying the rules may be easily visualized as follows. Note that this is not the only way used in the literature to apply the rules: other common strategies are *sequential* application and *minimal parallelism*, whose details of operation can be found in the *Handbook of Membrane Computing* [27]. Assume that a given region of a P system contains a multiset u of objects and a set R of rules. Each of these rules is either *enabled* by the objects in u (that is, the rule could in principle be applied to such objects) or not enabled, depending upon whether u contains the multiset of objects that appears in the left hand side (LHS) of the rule. Choose one of the enabled rules, and mark as used the objects of u that appear in the LHS of the rule. Now repeat the process with the remaining (unmarked) objects of u: select one of the rules enabled by them, and mark the objects used by the rule. Repeat the process until all objects are marked as used, or the remaining objects do not enable any rule.

Since at each step one rule is chosen in a nondeterministic way, the same initial multiset u of objects may lead to several different choices of the rules to be applied. For this reason, in the literature it is usually told that the rules are applied in a *nondeterministic* and *maximally parallel* way. Notice that no rule is actually applied until all the rules have been selected; they are then applied in parallel, each one consuming the objects mentioned in its LHS, and producing all the objects listed in the right hand side. The selection process together with the parallel application of all selected rules, in each of the regions determined by the membrane structure, constitutes a single computation step of the P system. A global clock is assumed, that synchronizes the operation of all the regions of the system; each computation step takes a single clock tick.

Let us also note that maximal parallelism does not imply that the maximal number of rules which can be applied with the objects in u will be selected. Instead, maximal parallelism means that no additional rule should be applicable in the same step to idle objects of u. The difference between the two approaches can be seen through a simple example.

Example 2. Assume that a region contains the following set of rules:

$$r_1 : ab \to v_1 \qquad r_2 : c \to v_2 \qquad r_3 : bc \to v_3$$
$$r_4 : a^3c^2 \to v_4 \qquad r_5 : ad \to v_5$$

and the multiset of objects $u = a^3b^2c^2$ (which is a shorthand for *aaabbcc*, or any other permutation of this string).

Then, the following multisets of rules can be obtained through the selection process described above, and are indeed maximally parallel:

$$\{(r_1, 2), (r_2, 2)\}$$
$$\{(r_3, 2)\}$$
$$\{(r_1, 1), (r_2, 1), (r_3, 1)\}$$

Here the notation (r_i, n) indicates that the multiplicity of rule r_i is n, that is, rule r_i has been selected n times. It is easily seen that fact of being maximally parallel is not related in any way with the number of rules which is selected to be applied. □

A *halting computation* of Π is a sequence of configurations $(\mathcal{C}_0, \ldots, \mathcal{C}_k)$ starting from the initial one such that every $\mathcal{C}_{i+1}$ is reachable from $\mathcal{C}_i$ by performing a single computation step, and no rule at all can be applied in $\mathcal{C}_k$. If Π never reaches a halting configuration, the result is an infinite, non-halting computation $(\mathcal{C}_i : i \in \mathbb{N})$.

In the next sections we show that the model of P systems we have defined is able to:

- simulate register (counter) machines and Turing machines, hence P systems constitute a universal (in the sense of Turing-complete) model of computation;
- solve **NP**-complete problems (and thus all problems in the class **NP**) in polynomial time (and exponential space), by exploiting *elementary* membrane division rules to simulate the computations of nondeterministic Turing machines;
- solve all problems in the complexity class **PSPACE**, in polynomial time and exponential space, by using division rules for *nonelementary* membranes to simulate the computations of *alternating* Turing machines.

We just recall that **NP** is the class of decision problems (equivalently, languages) which can be solved (resp., accepted) by nondeterministic Turing machines working in polynomial time with respect to the input size. Similarly, **PSPACE** is the class of decision problems (resp., languages) that can be solved (resp., accepted) by deterministic or nondeterministic Turing machines operating in polynomial space with respect to the input size. An excellent reference for the computational properties of these (as well as many others) complexity classes is [23].

6 Universality of multiset rewriting

Cooperative (or context-sensitive) multiset rewriting rules $u \to v$, even using only two symbols on the left-hand side, are powerful enough to reach computational universality. This result is actually simple enough to work as an introductory example.

Let us consider register machines, also called counter machines or program machines, as described by Minsky [22]. This model, inspired by the working of electronic computers, is able to compute any recursive function on natural

numbers. A register machine consists of a finite number of registers $r_1, \ldots, r_n$, each one containing a non-negative integer (i.e., a natural number). The input of the register machine is located in register r_1, while the remaining registers are initially null (i.e., they contain zero). The behaviour of the counter machine is described by a finite sequence of instructions, uniquely labelled by natural numbers, and a program counter stores the label of the instruction currently being executed. The program counter is, by convention, initially null.

The variant of register machine we consider here only has two kinds of instructions. The first kind is an increment instruction, which is denoted by the syntax

$$i\colon\ \mathrm{inc}(r), j$$

Here i is the label of the instruction, r is the name of a register, and j is an instruction label. The execution of this instruction causes the increment by one of the value of register r, and changes the value of the program counter to j.

The second type of instruction is a conditional decrement, with the syntax

$$i\colon\ \mathrm{dec}(r), j, k$$

where i is the label of the instruction, r is a register, and j and k two further instruction labels. The execution of this instruction depends on the current value of register r. If this is greater than zero, then it is decremented by one and the computation proceeds with the instruction labelled by j; otherwise, the register remains null and the program counter is set to k.

The computation of the register machine begins with the initial value of the program counter and proceeds by updating it according to the current instruction. The computation halts when the program counter reaches a value not associated with any instruction; alternatively, the computation can go on forever if this never happens. Halting computations can be considered to produce, as output, the final value of register r_1.

An instantaneous configuration $(i, x_1, \ldots, x_n)$ of the register machine consists of the value of the program counter i and the values $x_1, \ldots, x_n$ of the registers $r_1, \ldots, r_n$. Such configuration can be encoded as a multiset over the alphabet

$$\{p_i : i \text{ is a label appearing in the program}\} \cup \{r_1, \ldots, r_n\}$$

where the multiplicity of r_j is exactly the value x_j of register r_j for $1 \leq j \leq n$, and exactly one of the p_i's occurs and with multiplicity one, representing the current value of the program counter. Hence, an arbitrary configuration $(i, x_1, \ldots, x_n)$ of the register machine is encoded as $p_i\, r_1^{x_1} \cdots r_n^{x_n}$.

The instructions of the register machine are simulated by multiset rewriting rules operating in the maximally parallel way. Each increment instruction $i\colon\ \mathrm{inc}(r), j$ is implemented as a single rewriting rule

$$p_i \to p_j r$$

which simultaneously replaces the program counter symbol with the updated one, and increments by one the multiplicity of symbol r as required.

A conditional decrement instruction $i\colon\ \mathrm{dec}(r), j, k$ requires checking whether symbol r occurs with multiplicity zero or greater than zero; this can be implemented by using appropriately synchronised cooperative rewriting rules [10]. First of all, symbol p_i is rewritten by

$$p_i \to p'_i\ d_r$$

where symbol d_r will try to delete an occurrence of r, while p'_i waits one step in order to allow this decrement to occur:

$$d_r r \to d'_r \qquad\qquad p'_i \to p''_i$$

Notice that the first rule can only be applied when at least one instance of symbol r occurs: in that case, symbol d'_r is produced; otherwise, symbol d_r remains inert and is carried on to the next configuration. Symbol p''_i is able to discriminate between these two cases by applying one of the following mutually exclusive rules:

$$p''_i\ d'_r \to p_j \qquad\qquad p''_i\ d_r \to p_k$$

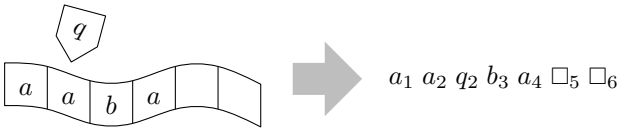
The rule that is actually applied produces the correct symbol encoding the updated value of the program counter.

The rules described above, applied to the encoding of a configuration of the register machine, produce (in up to three steps) the encoding of the next configuration, obtained by executing the prescribed inc or dec operation. This procedure halts if and when the simulated register machine halts; in that case, the output of the register machine can be recovered from the multiplicity of symbol r_1.

7 Efficient simulation of polynomial-time Turing machines

Although cooperative multiset rewriting rules suffice to attain universality, the register machines simulated in the previous section are, in general, exponentially slower than the reference model for efficient computation, namely Turing machines [8].

Turing machines themselves can be simulated by multiset rewriting systems when the space employed (i.e., the tape length m) is *a priori* known. The symbols in a multiset does not, by themselves, possess the linear ordering of the symbols on the tape of a Turing machine. This can be simulated by indexing the symbols of the tape alphabet Σ of the Turing machine with integer subscripts ranging from 1 to m, the space bound previously assumed. This allows us to encode the first m tape cells (including the blank ones) as a multiset of m symbols. The current state q of the Turing machine, ranging over the set of states Q, and the current position i of the tape head (also ranging from 1 to m) can be encoded by an extra symbol q_i, which contains both pieces of information. For instance, a sample Turing machine configuration is encoded as follows:



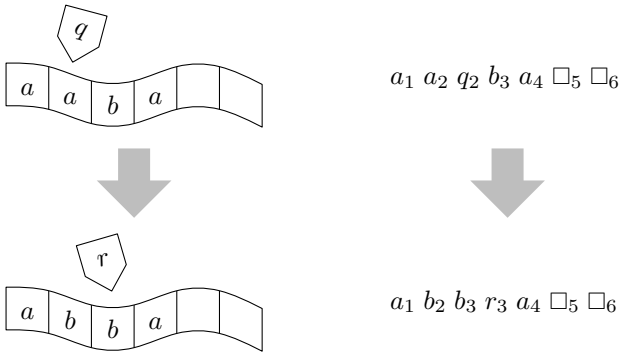


Figure 2: On the left, a Turing machine performing the transition $\delta(q, a) = (r, b, +1)$ and, on the right, the corresponding rewriting step between two multisets encoding the configurations of the Turing machine.

Let $\delta \colon Q \times \Sigma \to Q \times \Sigma \times \{-1, 0, +1\}$ be the transition function of the Turing machine. Each transition $\delta(q, a) = (r, b, d)$ is implemented by a rewriting rule for each position $1 \leq i \leq m$ of the tape:

$$q_i\ a_i \to r_{i+d}\ b_i$$

These rules simultaneously update the state and head position of the simulated Turing machine, and the content of the cell previously located under the tape head, as exemplified in Figure 2.

This simulation is carried out efficiently, actually in "real time" (each step of the Turing machine is simulated by a single rewriting step), and shows that multiset rewriting can be efficient, whenever the space bound is known. This means that, unlike the register machine simulation (where a single rewriting system simulates the given register machine on all possible inputs), a *family* of rewriting systems is needed, depending on the space bound (which is, ultimately, a function of the length of the input string).

The rewriting systems of the family differ only with respect to the subscripts of its symbols, since the rules simulating the transitions of the Turing machine are simply replicated for each cell index. This means that the family is *uniform*: there exists a single algorithm (and an *efficient* one on top of that) for constructing the m-th member of the family, where m is the length of the tape to be simulated. This echoes similar constructions for models of computation such as families of Boolean circuits [33], where the shape of each circuit does, of course, depend on the number of input bits.

8 Simulating nondeterminism with membrane division

Using membranes allows us to partition the space in multiple regions evolving in parallel, possibly according to distinct sets of rules. For instance, the rewriting rule $u_1 \to v_1$ might only be associated with membrane label h, while the rewriting rule $u_2 \to v_2$ is associated with label k. We denote such region-specific

rules with the syntax

$$[u_1 \to v_1]_h \qquad\qquad [u_2 \to v_2]_k$$

A mechanism to allow distinct regions to exchange objects is given by *communication rules*, either sending multisets towards the region outside a membrane h, or towards an internal membrane k:

$$[u_1]_h \to [\ ]_h\ v_1 \qquad\qquad u_2\ [\ ]_k \to [v_2]_k$$

These rules send out the objects in the multiset u_1 (resp., send in u_2) towards the target region, while simultaneously rewriting the multiset into v_1 (resp., v_2). Notice that a send-in communication rule is a potential source of nondeterminism, in the case that the region where the rule is applied contains two or more membranes with label k. By using communication rules, the multisets can be moved around the P system, allowing distinct processing to occur, such as conditional behaviour or "subroutines".

The role of membranes becomes dramatic when new membranes can be created during the computation. The most common approach is *membrane division*, which mimics the biological process of binary fission (occurring, for instance, in cell mitosis). A simple kind of membrane division rule has the form

$$[u]_h \to [v]_h\ [w]_h$$

where h is a membrane label and u, v, w are multisets. This rule is applicable to a region delimited by a membrane with label h that contains u as a submultiset. Membrane h is duplicated, its content is replicated inside both resulting membranes, with the exception that multiset u is rewritten into v in one membrane, and into w in the other one. This allows us to differentiate the two membranes, which can then operate in parallel on distinct data.

If further rules are simultaneously applicable inside membrane h, the usual semantics [25] requires the division process to logically occur after the remaining rules have been applied, although this only requires a single time step. This way, the only difference between the two resulting copies of membrane h is given by the two multisets v, w on the right-hand side of the division rule. If membrane h contains further membranes, the whole subtrees are replicated when the membrane divides.

The power of membrane division stems from the possibility to repeatedly apply division rules, creating in polynomial time an exponential number of membranes working in parallel. This allows us, for instance, to simulate nondeterminism using parallelism.

Let us consider the Turing machine simulation of Section 7, and extend it to nondeterministic Turing machines, allowing the transition function δ to assume (without loss of generality) one or two values for each input pair (q, a). Let us place the multiset encoding the instantaneous configuration of the Turing machine inside a membrane h. A deterministic transition $\delta(q, a) = \{(r, b, d)\}$ is simulated as before, by the rules

$$[q_i\ a_i \to r_{i+d}\ b_i]_h \qquad\qquad \text{for } 1 \le i \le m$$

On the other hand, a nondeterministic transition $\delta(q, a) = \{(r, b, d), (s, c, e)\}$ can be simulated by dividing the membrane, and carrying on the two possible

computations in parallel, inside the two resulting membranes:

$$[q_i\ a_i]_h \to [r_{i+d}\ b_i]_h\ [s_{i+e}\ c_i]_h \qquad \text{for } 1 \leq i \leq m$$

When the simulated Turing machine reaches a halting state, either an accepting state q_{yes} or a rejecting state q_{no}, the corresponding object $q_{yes,i}$ or $q_{no,i}$ (for some $1 \leq i \leq m$, corresponding to the position where the tape head is located upon halting) is sent out by a communication rule

$$[q_{yes,i}]_h \to [\ ]_h\ q_{yes} \qquad \text{for } 1 \leq i \leq m$$
$$[q_{no,i}]_h \to [\ ]_h\ q_{no} \qquad \text{for } 1 \leq i \leq m$$

The final result of the simulated Turing machine is given by the multiplicity of the object q_{yes} in the outside region when the P system halts: acceptance if there is at least one instance of q_{yes} (corresponding to at least one accepting computation of the Turing machine), and rejection otherwise (if all simulated computations were rejecting).

Figure 3 shows the correspondence between the nondeterministic choices of a Turing machine and the membranes obtained by division and operating in parallel in the P system simulating it.

9 Simulating alternation using non-elementary membrane division

The simulation of nondeterministic Turing machines in Section 8 only requires division rules for elementary membranes. The efficiency of P systems increases[3] by exploiting the division of non-elementary membranes, which replicates whole subtrees of the membrane structure. This makes it possible to simulate not only nondeterminism, but also *alternation*, which allows Turing machines to characterise the complexity class **PSPACE** in polynomial time [23].

Recall that an alternating Turing machine has a set of states Q partitioned into *existential* and *universal* states. A configuration leads to acceptance if either

- the state of the machine is accepting, or
- the state of the machine is existential, and at least one successor configuration leads to acceptance, or
- the state of the machine is universal, and all successor configurations lead to acceptance.

The Turing machine is said to accept its input if the corresponding initial configuration leads to acceptance, and to reject otherwise. A sample alternating computation tree is shown in Figure 4.

Notice that a standard nondeterministic Turing machine is simply an alternating Turing machine where all states are existential, and thus there is *no* alternation between existential and universal.

Let us consider an alternating Turing machine M working on input x; suppose, without loss of generality [23], that all computations have the same

[3] Under standard complexity theory assumptions, namely $\mathbf{NP} \neq \mathbf{PSPACE}$.

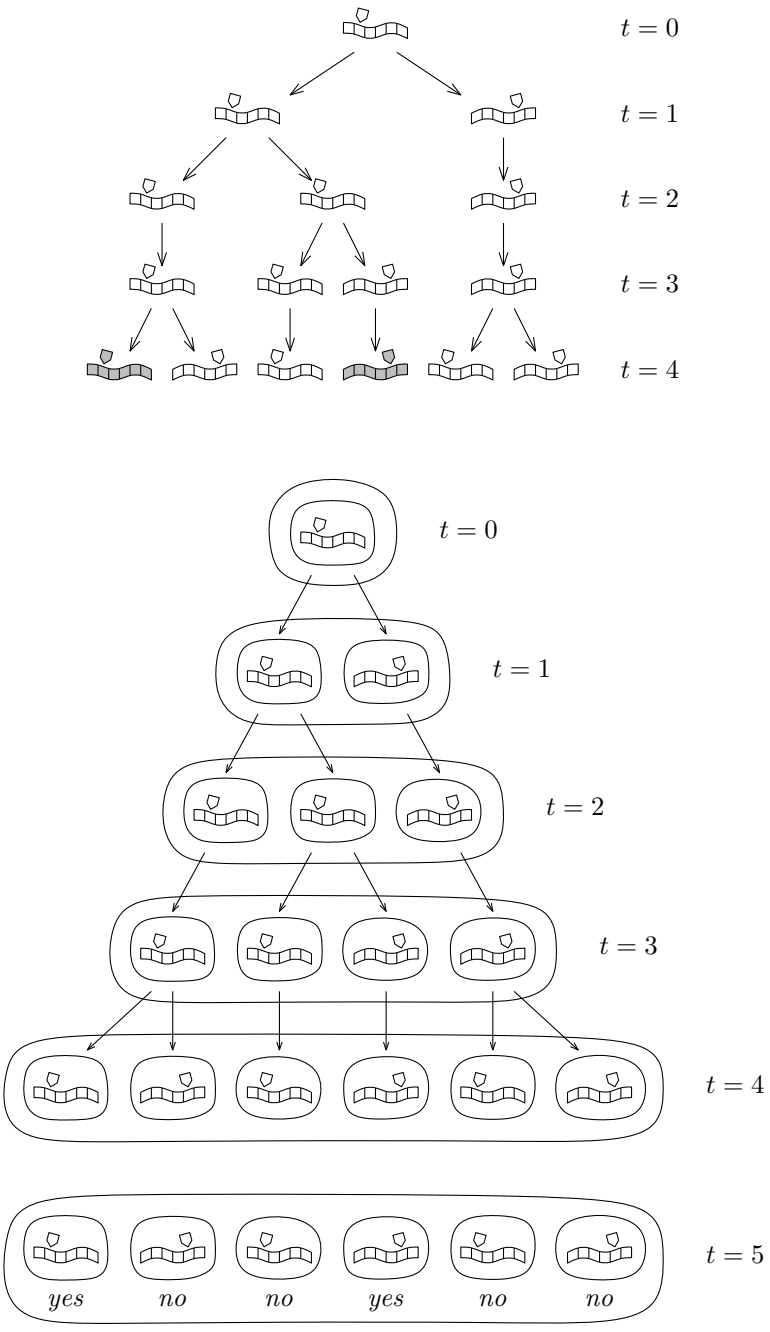


Figure 3: The computation tree of a nondeterministic Turing machine, with the accepting computations highlighted in grey (top), and the corresponding parallel computation of a P system simulating it (bottom). Each nondeterministic transition of the Turing machine is simulated by a membrane division, as highlighted by the corresponding arrows of the two diagrams. In the last computation step of the P systems ($t = 5$) the results of all simulated computations are sent out to the outermost membrane.

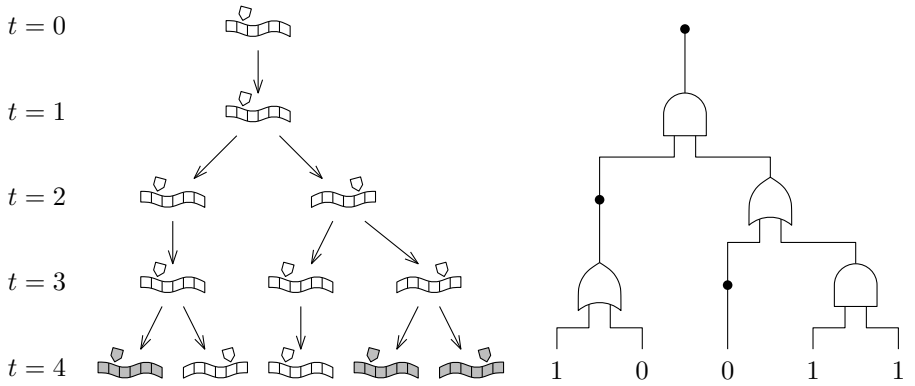


Figure 4: The computation tree of an alternating Turing machine (left), with the kind of state, existential or universal, represented by an isomorphic circuit with OR and AND gates, respectively (right).

length t. We simulate the machine by means of a P system by generating (via membrane division rules) a membrane structure isomorphic to the computation tree of M. The initial membrane structure consists of $t+1$ linearly nested membranes; since the rules will be the same for each membrane, these can be assumed to be all identically labelled by h. Notice that this last assumption is not strictly compatible with the formal definition of membrane structure, but can be simulated by replicating all rules for each membrane label; we can thus safely make this assumption without changing the computing power of the model. The encoding of the configuration of M (as described in Section 8) is initially located inside the outermost membrane.

The simulation of each computation step of M is similar to the one illustrated in Section 8, but before performing this simulation, the encoding of the configuration of M is moved one level deeper in the membrane structure, while leaving behind an object (q, a) representing the current state of the Turing machine and the object under its tape head. This process begins with the following rule:

$$[q_i\ a_i \to (q, a)\ q_{i,1}\ a_i]_h \qquad \text{for } q \in Q, a \in \Sigma, \text{ and } 1 \leq i \leq m$$

The object $q_{i,j}$, with $1 \leq j \leq m$ brings the object x_j encoding the symbol on the j-th tape cell of M into the internal membrane:

$$q_{i,j}\ x_j\ [\,]_h \to [q'_{i,j}\ x_j]_h \qquad \text{for } q \in Q, 1 \leq i \leq m, \text{ and } 1 \leq j \leq m$$

If there remain objects to bring in (i.e., if $j < m$), the object $q'_{i,j}$ is sent out with an incremented subscript:

$$[q'_{i,j}]_h \to [\,]_h\ q_{i,j+1} \qquad \text{for } q \in Q, 1 \leq i \leq m, \text{ and } 1 \leq j < m$$

Otherwise, the configuration of M has been completely moved one level deeper in the membrane structure, while leaving the object (q, a) outside. The current computation step of M can now be simulated. If this is a deterministic transition $\delta(q, a) = \{(r, b, d)\}$, then the P system applies one of the rewriting

rules

$$[q'_{i,m}\ a_i \to r_{i+d}\ b_i]_h \qquad \text{for } 1 \le i \le m$$

If the transition is nondeterministic, such as $\delta(q, a) = \{(r, b, d), (s, c, e)\}$, then the P system applies a membrane division rule, which is non-elementary in all computation steps but the last one:

$$[q'_{i,m}\ a_i]_h \to [r_{i+d}\ b_i]_h\ [s_{i+e}\ c_i]_h \qquad \text{for } 1 \le i \le m$$

Any subsequent computation steps are simulated recursively in the same way, at deeper and deeper levels of the membrane structure.

When a computation of the Turing machine reaches an accepting state q_{yes} or a rejecting one q_{no} in the last computation step, the corresponding object is sent out as *yes* or *no*, respectively:

$$[q_{yes,i}]_h \to [\,]_h\ yes \qquad [q_{no,i}]_h \to [\,]_h\ no \qquad \text{for } 1 \le i \le m$$

The membrane immediately outside receives either one or two objects yes, no from its child (resp., children) membranes, depending on whether the corresponding transition was deterministic or nondeterministic. The object (q, a) that was left behind when that transition was simulated stores two pieces of information: whether the transition was deterministic, and whether the state q of M at the time was existential or universal. This process is depicted in Figure 5.

If the transition was nondeterministic and the state existential, then the membrane receives two objects. The configuration previously located inside that membrane leads to acceptance if at least one of the two result objects is *yes*. This information is recursively propagated back towards the outermost membrane by using the rules

$$[(q, a)\ yes\ yes]_h \to [\,]_h\ yes \quad [(q, a)\ yes\ no]_h \to [\,]_h\ yes \quad [(q, a)\ no\ no]_h \to [\,]_h\ no$$

that is, the two objects are combined by logical disjunction, corresponding to an existential nondeterministic choice.

Dually, if the transition was nondeterministic and the state universal, the two result objects are combined by logical conjunction, corresponding to a universal nondeterministic choice:

$$[(q, a)\ yes\ yes]_h \to [\,]_h\ yes \quad [(q, a)\ yes\ no]_h \to [\,]_h\ no \quad [(q, a)\ no\ no]_h \to [\,]_h\ no$$

If the transition was instead deterministic, then the existential or universal nature of the state is immaterial, and the unique result object is simply propagated outside:

$$[(q, a)\ yes]_h \to [\,]_h\ yes \qquad [(q, a)\ no]_h \to [\,]_h\ no$$

A simple inductive reasoning shows that the *yes* or *no* object sent out from the outermost membrane corresponds to the actual result of the simulated alternating Turing machine.

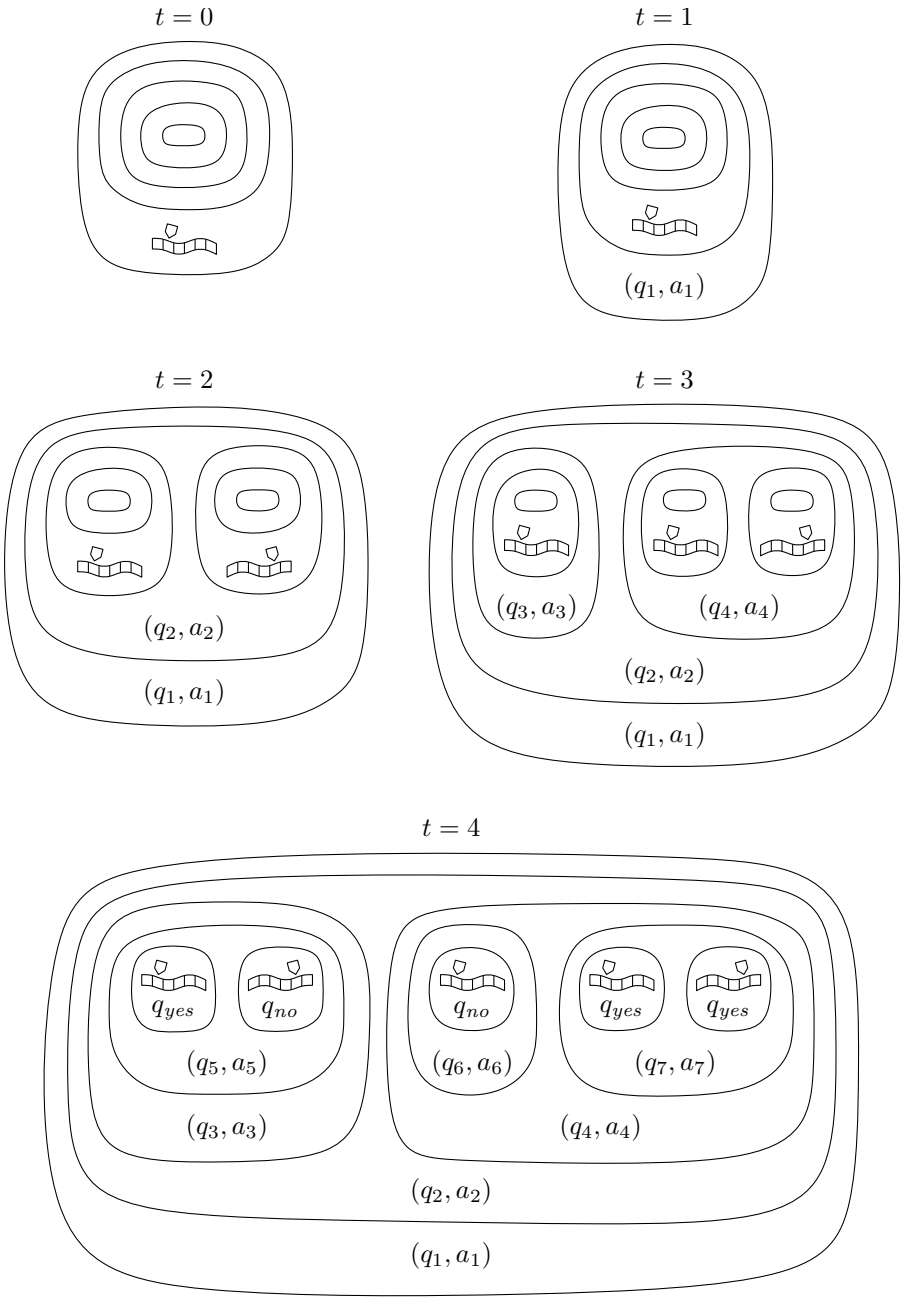


Figure 5: The computation tree of Figure 4 is simulated by the isomorphic membrane structure of a P system with membrane division (bottom, $t = 4$), which is constructed by using send-in and membrane division from an initially linear membrane structure ($t = 0$). The internal membranes contain encodings of configurations of the Turing machine, while the external ones each contain an object storing the information needed to combine (by OR or AND) the results of the subcomputations being simulated by its children membranes.

10 Conclusions and pointers to the literature

In this chapter we tried to give a gentle introduction to P systems, using a simplified cell-based model. Indeed, since their first appearance in the literature [24], it has been clear that P systems constitute more a *framework* to define classes of abstract models of computation, rather than a specific model. A number of variants of P systems, and of models of computation inspired by them, has been proposed in the last years.

A first class of these variants is obtained by removing or adding computational features to the cell-based model we have described above. So, for example, when investigating computability and computational complexity issues, a common observation made in the membrane computing community is that cooperative rules, having two or more objects in their LHS, are too powerful. Indeed, as we have shown above, it is very easy to obtain the computational power of register machines and of Turing machines (hence, universality) by means of a simple set of rules, and it is also easy to solve computationally difficult problems in polynomial time. Among the ways proposed in literature to avoid cooperative rules, one of the most successful (in terms of number of papers published) is endowing membranes with an electric charge, which can be either positive (denoted with $+$), negative ($-$), or null (0). The resulting model is called P systems with *active membranes*. The rules which can be used in this model are the same as defined in section 4, with two important differences:

- all the membranes mentioned in each rule have an electrical charge. This allows us to partition the set of rules associated with the region enclosed by a membrane into classes, each one composed of rules which can only be applied under a given charge;
- the LHS of every evolution rule contains just one object. No constraints are instead imposed on the multiset of objects that these rules can produce on their right hand side.

Adding electrical charges to membranes allows one to *move* cooperation from multisets of objects to single objects and membranes. With this limited version of cooperation, it clearly becomes more difficult to both obtain the computing power of Turing-machines and solving **NP**-complete [25] or **PSPACE**-complete [30] problems. However, by using standardized constructions [19] it actually becomes much easier to design such systems. An extended literature exists on membrane systems with active membranes, especially related with the capability of solving computationally hard problems: the reader may consult the references given at the end of this chapter to find good pointers to start with.

Another example of cell-based membrane systems is *P systems with symport/antiport rules* [28], in which the rules are cooperative and are inspired from the coordinated transport of substances across the membranes. Indicated with x and y two multisets of objects, the rules may be of the following forms:

- (x, in): the objects of x are moved into the current region (where the rule is defined) from the surrounding region;
- (x, out): the multiset x of objects is moved to the surrounding region;

- $(x, in; y, out)$: the objects of x are moved from the surrounding region into the current region, and simultaneously the objects of multiset y are moved in the opposite direction.

The first two kinds of rules are called *symport* rules, whereas the rule $(x, in; y, out)$ is called an *antiport* rule. Even inside the class of cell-based symport/antiport P systems, some variants have been defined and their computational properties have been investigated; we refer the interested reader to [11] and to chapter 5 of the *Handbook of Membrane Computing* [27].

Cell-based P systems are not the only model which has been proposed in the literature. By neglecting the internal structure of cells, and focusing instead the attention on the connections between different cells, *tissue P systems* can be defined [20]. In this model the cells are represented as the nodes of a graph, whose arcs indicate the connections between the cells, and thus provide a way to represent inter-cellular communication. Such a communication occurs via rules that closely resemble the above mentioned antiport rules, and consists of an exchange of substances between adjacent cells. The interested reader can find further information in chapter 9 of [27]. Also in this case, several variants of tissue P systems have been defined: in the original model [20] the cells had an internal state, a feature which has been dropped very soon since in the computability and computational complexity setting it is considered too powerful. On the other way, cell division rules have been added in order to allow the systems to solve computationally difficult problems [29], just like it happens with P systems with active membranes. However, the lack of internal structure prevents them from being able to solve **PSPACE**-complete problems; perhaps surprisingly, they turn out to be able to solve in polynomial time exactly the problems in $\mathbf{P}^{\#\mathbf{P}}$, the class of problems solved by Turing machines with oracles for counting problems [18].

Tissue P systems have also inspired *spiking neural* (SN) P systems [15], in which the graph of cells represents a neural network, on its turn inspired by the structure and functioning of living nervous systems. One peculiarity of SN P systems is that they use only one kind of objects, called the *spike*. Each cell – called *neuron* in this model – contains a set of rules, which may be of two different types: *spiking* rules, that send a predefined number of spikes to all neighbouring neurons, and *forgetting* rules, that delete a predefined number of spikes from the neuron itself. Spiking rules are enabled according to the number of spikes currently contained in the neuron: in fact, each rule has an associated regular expression which determines a regular set of non-negative integer numbers, that describe when the rule is enabled to spike. Also in this case the *Handbook of Membrane Computing* [27] is a valuable resource to obtain further information, but the reader should also be aware that several further variants of spiking neural P systems have been defined and investigated in the last few years.

For an updated bibliography, we refer the reader to the P systems Web page [31] as well as to the Web page of the International Membrane Computing Society [14], that publishes a Bulletin (four issues per year) containing the latest news about P systems, and promotes the International series of *Conferences on Membrane Computing* (CMC), and *Asian Conferences on Membrane Computing* (ACMC). By consulting the Bulletin and the Proceedings of these conferences, the reader will discover many other interesting P systems models,

such as *(enzymatic) numerical P systems*, *P automata*, *population P systems*, *metabolic P systems*, P systems with *antimatter*, SN P systems with *anti-spikes*, and still *probabilistic* and *quantum* and *asynchronous* variants, operating under different modes of applying the rules. There is an entire world to be explored!

We conclude by remarking that in this chapter we took a *theoretical* approach, interested to determine the computational properties of P systems when considered as abstract models of computation. From the point of view of applications there is once again a considerable literature on using P systems to model different kinds of biological and natural phenomena, and an even vaster number of papers and software tools dealing with efficient simulations of several variants of P systems, exploiting the parallel computing power of modern GPUs. The above mentioned references and resources provide also in this case the best pointers to start with.

References

[1] Scott Aaronson. Why philosophers should care about computational complexity. Technical report, `http://eccc.hpi-web.de/report/2011/108/`, 2011.

[2] Leonard M. Adleman. Molecular computation of solutions to combinatorial problems. *Science*, 266(5187):1021–1024, 1994.

[3] Roberto Barbuti, Andrea Maggiolo-Schettini, Paolo Milazzo, Giovanni Pardini, and Luca Tesei. Spatial P systems. *Natural Computing*, 10(1):3–16, 2011.

[4] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing, STOC '71*, pages 151–158, 1971.

[5] Jack Copeland, editor. *The Essential Turing: Seminal Writings in Computing, Logic, Philosophy, Artificial Intelligence, and Artificial Life, Plus the Secrets of Enigma*. Oxford University Press, 2004.

[6] Martin Davis, editor. *The Undecidable: Basic Papers on Undecidable Propositions, Unsolvable Problems and Computable Functions*. Raven Press, 1965.

[7] Jack Edmonds. Paths, trees and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965.

[8] Patrick C. Fischer, Albert R. Meyer, and Arnold L. Rosenberg. Counter machines and counter languages. *Theory of Computing Systems*, 2(3):265–283, 1968.

[9] Lance Fortnow and Steve Homer. A short history of computational complexity. *Bulletin of the European Association for Theoretical Computer Science*, 80:95–133, 2003.

[10] Pierluigi Frisco. P systems, Petri nets, and program machines. In Rudolf Freund, Gheorghe Păun, Grzegorz Rozenberg, and Arto Salomaa, editors,

Membrane Computing, 6th International Workshop, WMC 2005, volume 3850 of *Lecture Notes in Computer Science*, pages 209–223. Springer, 2006.

[11] Pierluigi Frisco. *Computing with Cells: Advances in Membrane Computing*. Oxford University Press, New York, NY, USA, 2009.

[12] Juris Hartmanis and Richard E. Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117:285–306, 1965.

[13] Tom Head. Formal language theory and DNA: An analysis of the generative capacity of specific recombinant behaviors. *Bulletin of Mathematical Biology*, 49(6):737–759, 1987.

[14] International Membrane Computing Society (IMCS). `http://membranecomputing.net/`.

[15] Mihai Ionescu, Gheorghe Păun, and Takashi Yokomori. Spiking neural P systems. *Fundamenta Informaticae*, 71(2,3):279–308, 2006.

[16] Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–104. Plenum Press, 1972.

[17] Tsuneyoshi Kuroiwa, Haruko Kuroiwa, Atsushi Sakai, Hidenori Takahashi, Kyoko Toda, and Ryuuichi Itoh. The division apparatus of plastids and mitochondria. *International Review of Cytology*, 181:1–41, 1998.

[18] Alberto Leporati, Luca Manzoni, Giancarlo Mauri, Antonio E. Porreca, and Claudio Zandron. Characterising the complexity of tissue P systems with fission rules. *Journal of Computer and System Sciences*, 90:115–128, 2017.

[19] Alberto Leporati, Luca Manzoni, Giancarlo Mauri, Antonio E. Porreca, and Claudio Zandron. A toolbox for simpler active membrane algorithms. *Theoretical Computer Science*, 673:42–57, 2017.

[20] Carlos Martín-Vide, Gheorghe Păun, Juan Pazos, and Alfonso Rodríguez-Patón. Tissue P systems. *Theoretical Computer Science*, 296(2):295–326, 2003.

[21] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 7:115–133, 1943.

[22] Marvin Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall, 1967.

[23] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1993.

[24] Gheorghe Păun. Computing with membranes. *Journal of Computer and System Sciences*, 61(1):108–143, 2000.

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

[26] Gheorghe Păun. Membrane computing: History and brief introduction. In Erol Gelenbe and Jean-Pierre Kahane, editors, *Fundamental Concepts in Computer Science*, pages 17–41. Imperial College Press, 2009.

[27] Gheorghe Păun, Grzegorz Rozenberg, and Arto Salomaa, editors. *The Oxford Handbook of Membrane Computing*. Oxford University Press, 2010.

[28] Andrei Păun and Gheorghe Păun. The power of communication: P systems with symport/antiport. *New Generation Computing*, 20(3):295–305, 2002.

[29] Gheorghe Păun, Mario J. Pérez-Jiménez, and Agustín Riscos-Núñez. Tissue P systems with cell division. *International Journal of Computers, Communications & Control*, 3(3):295–303, 2008.

[30] Petr Sosík and Alfonso Rodríguez-Patón. Membrane computing and complexity theory: A characterization of PSPACE. *Journal of Computer and System Sciences*, 73(1):137–152, 2007.

[31] P systems web site. `http://ppage.psystems.eu/`.

[32] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1936.

[33] Heribert Vollmer. *Introduction to Circuit Complexity: A Uniform Approach*. Texts in Theoretical Computer Science: An EATCS Series. Springer, 1999.

[34] Richard Zach. Hilbert's program then an now. In Dale Jacquette, editor, *Philosophy of Logic*, volume 5, pages 411–447. Elsevier, 2006.