Recent Computability Models Inspired from Biology: DNA and Membrane Computing

Gheorghe PĂUN, Mario J. PÉREZ-JIMÉNEZ

ABSTRACT: We briefly present two areas of natural computing, vividly investigated in the recent years: DNA computing and membrane computing. Both of them have the roots in cellular biology and are rather developed at the theoretical level (new concepts, models, paradigms of computer science, with mathematical and epistemological significance have been considered in this framework), but both areas are still looking for implementations of a practical interest.

Keywords: Computer Science, Mathematics, Turing computability, Biochemistry, DNA computing, Membrane Computing.

1 Introduction

In a great extent, the history of (theoretical) computer science is the history of attempting to model (formalize) the computations performed in nature, starting with the way the humans compute (this was, for instance, the explicit goal of Leibniz, at the turn of seventeenth and eighteenth centuries, and of Turing, in 1935–36), going through the supposed organization and functioning of the brain, of the networks of neurons (this is the origin of finite automata, see McCulloch & Pitts 1943, Kleene 1956, and of neural networks area, see, e.g., Andersson 1996), and getting closer and closer to the molecular biology, to the cell and its constituents. The general assumption/observation is that life has used for millions of years many *processes*, taking places in specific material environments and using specific material structures, which may be considered as *computing processes and devices*.

This assertion is debatable: What is a computation? Does nature computes? At which levels? Which processes are and which are not computations? And so on and so forth.

We do not enter here this debate (we adhere to the opinion that nature just evolves, the goal of life is life itself, and a process can be considered a computation only by a human being, 'computing' is not a natural activity, but an artifact) and we adopt a mathematical perspective: computing means Turing computing (an input-output relation, established by "mechanically" following an algorithm: a precise sequence of instructions which always halts). Moreover, when we look to a piece of reality we search suggestions for two basic ingredients of a computing model: data structures (supports for computing) and operations about these data structures. Of course, these elements of any computing model do not exist as such, we abstract them from the biochemical objects/structures and processes. (We anticipate, illustrating this discussion with the structure of the DNA molecules and the many operations which are possible with these molecules.) After abstracting a data structure and some operations about it, we can proceed in a rather standard way in order to obtain a computing device where the data structure can be processed by means of the considered operations: consider an initial configuration of our device, consisting of one or several sets of operation-based 'instructions' and a given collection of data; by using the instructions in a specified manner we pass to a new configuration; by iterating the operations we ob-

tain a computation; define in some way the notion of a successful computation; starting from the initial configuration of the 'device', where the input of the computation is placed, and proceeding along a successful computation, we can get the output of the computation, its result.

This scenario is followed both in DNA and membrane computing, the two areas of natural computing we are going to discuss in the next sections. Before going into some details, some generalities are still worth mentioning. Natural computing is an important trend of computer science looking for computing models – perhaps also computers – inspired from the way nature "computes", and it contains areas which were already proved as successful both theoretically and practically. The examples of neural networks and of genetic algorithms (more generally, of evolutionary computing) are illustrative in this respect. Especially interesting from our point of view are genetic algorithms (see, e.g., Beyer 2001): optimization problems are formulated in terms of improving a fittness function over a population of "chromosomes", represented by strings of digits, which evolve by means of operations known from the genetics, such as recombination and point mutation. At the first sight, this is just a random walk through the space of possible solutions, a brute force approach, which highly contrasts with the surprising success of such algorithms in a large number of applications. There is no mathematical "explanation" of this success, but this observation brings optimism to natural computing research: the fact that life uses certain tools (again, at an abstract level: data structures and operations), which were improved during evolution, during a long interval of time, can be an indication that these tools have some features which make them useful also for computing. If recombination, point mutation, selection, etc, were useful and so efficient in genetic algorithms, why not assuming that other similarly useful tools and computing ideas can be found at the genetic level or at upper levels, such as the living cell. Insisting at the DNA level we get DNA computing, insisting at the cellular level we get the membrane computing.

Some differences between the above mentioned areas are visible and interesting for what follows. Neural networks and genetic algorithms are inspired from biology and implemented on usual electronic computers. They aim at finding new types of algorithms, for a better use of the classic computer, maybe for also improving the architecture of the classic computer. In turn, DNA computing has a different goal, a new ambition: to use DNA molecules as a support for computing, to replace/supplement the electronic chip with a "wet chip". The need for this arises from the observation that the electronic chips cannot continue too much to become smaller, cheaper, faster, because, if they will do it as in the last decades (in sixties was formulated the so-called Moore law, saying that every year the computers will become twice smaller, cheaper, and faster; later the interval was increased to 18 months, then to two years, now the law seems to be abandoned), they will soon reach the quantum barrier. Moreover, the sequential computers have intrinsic efficiency limits (they cannot solve problems of exponential complexity in a feasible time); a possible way to overpass this intrinsic limit is the parallelism, but this raises serious difficulties in the electronic case, for instance, in what concerns the control of processors, the energy they dissipate, etc. A promise comes from biology, from DNA and cellular levels. The parallelism made

possible by DNA molecules is really massive: billions of molecules – hence "processors" – can find room in a tiny biochemical test tube; moreover, biochemistry is highly nondeterministic, which means that the control of computations could be of a new type, "cheap" from several points of view.

DNA computing deals with computing *in vitro*, with implementing algorithms in a laboratory (there also are researches about DNA computing *in vivo*, mainly dealing with the computing-like processes which take place in certain cells at the genetic level, for instance, in ciliates). What about the cell, the smallest living unit we know? The cell is a very complex biochemical "factory", where many processes take place, some of them of an informational type. Can the living cell be considered/interpreted/used as a computing device? The question is not simple: What is a cell (from a mathematical point of view)? Which features of the cell structure and which processes taking place in the cell compartments are useful/essential for a computing model? After abstracting a cell-like computing model, as we will immediately see, where should we try to implement it, on a usual electronic computer (like in the case of genetic algorithms) or on a bio-support (as it is the goal of DNA computing)? We do not answer here these questions; the aim of this paper is only to let the reader having a first contact with DNA and membrane computing areas, at an introductory and informal level. Details (including further questions) can be found in the titles mentioned in the bibliography, and the web pages devoted to the two discussed domains.

2 A *Glimpse to DNA Computing*

The fact that the DNA molecules can be used as a support for computing has been speculated since several decades (Ch. Bennett, M. Conrad, etc), but the first (successful) experiment has been reported in 1994, by L.M. Adleman. The problem dealt with was the so-called Hamiltonian path problem for directed graphs: whether or not a path exists which visits each node of a graph exactly once. The problem is known to be computationally difficult (it is NP-complete), but the solution was very efficient (obtained in linear time). Seen after eight years, the experiment looks simple from a biochemical point of view: one encodes the nodes by single stranded DNA molecules of length 20 (consisting of 20 nucleotides), then one encodes the edges of the graph by single stranded DNA molecules which are complementary, in the Watson-Crick sense, to the node-codes and consist of the second half of the code of the emerging node and the first half of the target node of the edge, one places billions of such molecules in a test tubes and one lets them to anneal and form double stranded molecules; in this way, the edge-codes act as splints of node-codes, hence the possible paths in the graph are encoded by chains of node-codes and edge-codes; by well-known filtering procedures one selects the paths which visit all nodes exactly once (the Hamiltonian ones); if any molecule exists which encodes such a path, then the problem has the answer "yes".

Adleman's experiment was a great event in computer science, in spite of the fact that the graph considered was a small one, with only seven nodes and thirteen edges. However, the experiment was the first one of this type, it proved that DNA computing is possible; in the terms of Hartmanis (1994), this was a *demo*. Many experiments

have followed, in USA, Japan, Europe, conferences were initiated in this area, a large number of papers were published. However, up to now no computation of a practical interest was reported. The passing from a toy-problem to a problem of a significant size is not at all easy, because of the quantity of required DNA (and of other biochemical tools), and, mainly, because of the difficulty of coping with errors. The biochemical reactions cannot (yet) be perfectly controlled, the algorithms based on them are error-prone, and this requests both progresses in bioengineering and in improving the theoretical models (and in finding the adequate classes of problems to be attacked in this area, e.g., with error resistant solutions).

To abstract a little bit, Adleman has used as data structure the DNA molecule (single or double stranded) and the annealing operations as the basic operation. In a massively parallel manner, this operation ensures the generation of all candidate solutions of the problem (providing that "enough" DNA is present). The filtering phase, when it was checked whether or not a solution exits, can be considered as the phase of reading the result; it was done manually, but the number of steps was of the same order of magnitude as the number of nodes of the graph – hence the linear time of solving the problem.

The many experiments which were reported in the meantime use similar data structures (sometimes, circular molecules, or molecules with other shapes, such as hairpins), but several other operations. One of the most interesting case is that of the splicing operation, considered for the first time by T. Head, in 1987, in a theoretical framework not directly dealing with computing. This operation was explicitly used in a computing model by Păun, Rozenberg, and Salomaa (1996), where the notion of an H system was introduced. Because these systems are among the most investigated DNA computing models and because they are typical for this area, we will present them with some details in the next section. For further models of DNA computing we refer to the web page www.wi.liacs.nl/home/pier/aaa, to the monograph Păun, Rozenberg, and Salomaa, 1998, to the proceedings volumes of the series of conferences *DNA Based Computers*, initiated in 1995 in Princeton, as well as to the new Kluwer journal *Natural Computing* and the new Springer series of books with the same title.

3 Computing by Splicing: H Systems

The abstract splicing operation was introduced in 1987 by T. Head, as a mathematical model of the recombination of DNA molecules under the influence of restriction enzymes (and ligases) –therefore (theoretical investigations of) computing by splicing has been initiated seven years before Adleman's experiment.

The splicing of two DNA molecules corresponds to two operations: cutting the molecules by restriction enzymes and pasting together the fragments obtained in this way, providing that they have matching sticky ends. For example, consider the following two (double stranded) DNA molecules:

5' - CCCCCTCGACCCCC - 3'
3' - GGGGGAGCTGGGGG - 5'
and
5' - AAAAAGCGCAAAAA - 3'

3' - TTTTTCGCGTTTTT - 5'

and the restriction enzymes *Taq*I and *Sci*NI, for which the recognition sites are:

T C G A and G C G C
A G C T C G C G

espectively (we have also indicated the cuts that these enzymes make within their recognition sites). These enzymes will cut the above two molecules producing the following four molecules:

5' - CCCCCT CGACCCCC - 3'
3' - GGGGGAGC, TGGGGG - 5'

5' - AAAAAG CGCAAAAA - 3'
3' - TTTTTCGC, GTTTTT - 5'

Because the fragments obtained in this way have complementary sticky ends, the annealing of sticky ends followed by ligation will either reproduce the two original molecules, or the following two new molecules will be formed:

5' - CCCCCTCGCAAAAA - 3'
3' - GGGGGAGCGTTTTT - 5'
5' - AAAAAGCGACCCCC - 3'
3' - TTTTTCGCTGGGGG - 5'

As a model of the above biochemical operation, T. Head considered a string operation (passing from double stranded sequences to strings is allowed due to the precise Watson- Crick complementarity of nucleotides) which was further abstracted in (Pãun 1996a). In short, one considers splicing rules of the form $r=u_1\#u_2\$u_3\#u_4$, where u_1, u_2, u_3, u_4 are strings over a given alphabet. Given such a rule r, and two strings $w_1u_1u_2w_2$, $z_1u_3u_4z_2$, by the splicing of these strings we get the strings $w_1u_1u_4z_2$, $z_1u_3u_2w_2$. The relation with the biochemical operation of recombination is clear: each string u1u2, u3u4 corresponds to the site of a restriction enzyme, two sites stay together in the same rule if they produce matching sticky ends, while the crossing is supposed to be included in the context strings (hence it can be empty).

From string operations we pass to language operations in the natural manner: consider a set R of splicing rules (of any type) and a set L of strings; by splicing any two possible strings from L we get a new set of strings, $R(L)$; the process can be iterated, starting either from $R(L)$ or from $L\cup R(L)$ (the latter case corresponds to the observation that when a DNA molecule is present, we may assume that arbitrarily many copies of it are present, obtained by amplification; in the first case we may assume that the reaction is complete, all old strings having been processed).

In this way, a computing (language generating) device is obtained, of the form $\gamma=(V, A, R)$, where V is an alphabet, A is a set of strings over V, and R is a set of splicing rules over V. Such a machinery generates a language in the following way: start from the strings in A, splice them in all possible ways with respect to the rules in R, add all resulting strings to A and iterate the process. The language generated by , denoted by $L(\gamma)$, consists of all strings which can be obtained in this way. A standard extension is to also consider a terminal alphabet, $T\subseteq V$, and to accept in $L(\gamma)$ only the strings consisting of symbols from T.

Splicing systems $\gamma=(V, A, R)$ with finite sets A and R (hence without a terminal alphabet) cannot generate all regular languages, but, conversely, the language $L(\gamma)$ generated by such a system is regular. For Head rules this was proved by K. Culik II and *T.* Harju already in 1991, while for Pãun rules it was proved by D. Pixton (1996).

Splicing systems of the form $\gamma=(V, T, A, R)$, with finite A and R, characterize the family of regular languages, hence the power of finite automata. From a computational point of view, the competence of finite automata is too limited. *A* characterization of recursively enumerable languages (hence of the power of Turing machines) is obtained when using a set of splicing rules which is a regular languages (the rules are written as strings, hence it makes sense to speak about the type of their language).

From a computational point of view, the above mentioned results are quite "frustrating": finite H systems compute only at the level of finite automata, while the computational universality is obtained by using an infinite set of splicing rules. Fortunately, the proof of the universality (Pãun 1996b) indicates a number of ways for overcoming this drawback. This proof goes as follows. Starting from a type-0 Chomsky grammar G, one constructs an equivalent extended H system γ whose sentential forms are circularly permuted versions of the sentential forms of G, and the simulation of the rules of G takes place within suffixes of the sentential forms of γ (the circular permutation ensures that each derivation step in G can be simulated in this way). Very crucial for this "rotateand- simulate" procedure are the first and the last symbols of each sentential form, which in fact are markers, holding some information about the current stage of the simulation. That is, we can ignore the strings we splice as long as we know their first and last symbols, and the splicing sites. In other words, it is sufficient to have a finite number of splicing rules, and to associate with each rule certain "promoters", which are symbols whose presence allows the splicing of a given string.

This observation leads to *extended H systems with permitting contexts*, whose rules have associated finite sets of symbols such that a rule is applicable only to strings which contain the associated symbols. Actually, many other types of controlled H systems were considered. About a dozen such systems can be found in the literature, in general imitating the types of controls known from the "classic" regulated rewriting area in formal language theory. In particular, the following controls were investigated: *forbidding contexts* (symbols are associated with rules and a string cannot be spliced if it contains such a symbol), *target languages* (the splicing of two strings is allowed only if the resulting strings belong to a given regular language which is associated with the rule or associated with the whole set of rules; in the former case we say that we have *local* targets, and in the latter case we have a *global* target), *programmed control* (a next mapping is given on the set of rules, which indicates the sequencing of rules), *evolving sets of rules* (at each step, a different set of rules is produced, by point mutation rules which act on the splicing rules themselves), *double splicing* (the strings resulting from the splicing of two strings are immediately spliced again by any available rule), considering *multisets* of strings (the strings are counted, by splicing they are consumed, the multiplicity of strings resulting from a splicing operation is increased by one). References can be found in the sources mentioned at the end of the previous section.

In all these cases one gets characterizations of recursively enumerable languages. Similar results are obtained for various types of *distributed H systems*, where the strings and the set of splicing rules are separated in various ways, so that a system of several "simple" H systems is obtained, working in a parallel manner and cooperating in obtaining a common result.

All the proofs are constructive, hence, starting from universal type-0 grammars, one can obtain *universal H systems*, hence *programmable* H systems able of computing at the level of Turing machines.

These results have some general consequences. The fact that iterated splicing with respect to a finite set of rules computes only regular languages indicates that what we can compute *in the free mode* (hence in the mode encountered in nature) is not too much. Therefore, we need a further ingredient in order to achieve the desired universality.

A series of possibilities are suggested by the controlled and the distributed H systems mentioned above. However, the controls and the distributed architectures considered up to now seem not easy to be implemented within the present day biotechnology. So, an important dilemma arises: should we confine at the level of finite automata and hope to implement a weak computing device based on splicing soon, or, providing that we need a higher computing power, should we have to look for improved models and for improved bio-technologies in order to, hopefully, implement a universal DNA computer? A question of a great interest, to be addressed in an interdisciplinary team, in the near future.

4 Membrane Computing

DNA computing deals with processes taking place at the genetic level, with the hope to repeat them *in vitro*. However, many processes which are observed *in vivo* cannot be repeated at all, or they develop in a different way *in vitro*. A possible solution is to use the cell itself as the environment of a computation, and this is the starting point of membrane computing.

The area is rather young – it was initiated by (Pãun 2000) (the paper was circulated on web at the end of 1998) –but it is rather developed from a mathematical point of view.

We will recall here only the most basic ideas and results, the main classes of membrane systems and their properties; for further details, we refer the reader to the web page http://psystems.disco.unimib.it and to the monograph (Pãun 2002).

Membrane computing starts from the assumption that the processes taking place in the compartmental structure of a living cell can be interpreted as computations. Abstracting from the biochemical details, one gets *membrane systems* (called also P systems), which, roughly speaking, consists of a cell-like *membrane structure*, in the compartments of which one places *multisets* of *objects* which evolve according to given *rules* in a synchronous, parallel, and non-deterministic manner. The objects can be described by symbols or by strings of symbols from a given alphabet. The objects can also pass through membranes, the membranes can be dissolved, divided, created. An evolution of a membrane system is a *computation*; we consider as successful only the

halting computations, with which a *result* is associated. Many classes of P systems were already considered in the literature. Most of them are computationally complete, i.e., equal in power to Turing machines. If an exponential workspace can be created (in polynomial time), by dividing membranes, or by replicating string-objects, or by creating membranes from objects which can be replicated, then polynomial time solutions to NP-complete problems can be obtained.

The membranes appearing in a P system try to mimic the role and the functioning of membranes from living cells. The basic function of biological membranes is to *define compartments* and to *relate compartments to their environment*, including neighbouring compartments. The currently accepted model of the membrane structure is the so-called *fluid-mosaic model*, proposed in 1972 by S. Singer and G. Nicolson. According to this model, a membrane is a phospholipid bilayer in which protein molecules (as well as other molecules) are totally or partially embedded.

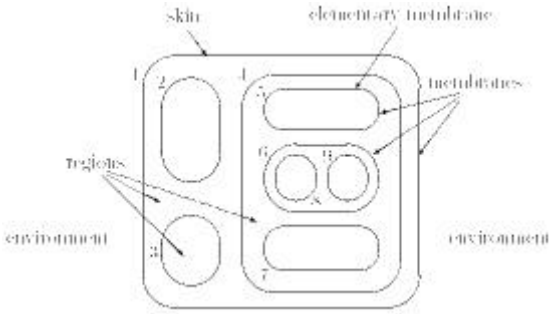

Figure 1: A membrane structure

The (plasma) membrane is only partially permeable (in general, to small non-charged molecules), but various molecules can pass through membranes by means of *protein channels*. The transmembrane transfer can take place in a *passive* manner, e.g., by diffusion towards the region of lower concentration, and in an *active* (mediated) manner. Actually, there are two main types of protein channels: those which just select the moving objects by their size, and those which interact with specific molecules when helping them to cross the membrane; the latter type is called *carrier protein*.

The protein channels are also important for the inter-cellular communication: neighbouring cells can link their protein channels, and in this way, a complex communication network can be established among cells. It is important to note that the protein channels can be open or closed, depending on the contents of the adjacent compartments. For instance, if one of the cells is invaded by "undesired" molecules, then the cell isolates itself from the neighbouring cells by closing the passage channels – they may be re-opened again, once the emergency situation has been resolved.

Membrane Systems (with Symbol-Objects) – An Informal Introduction.

The *membrane structure* of a P system is a hierarchical arrangement of membranes (understood as three dimensional vesicles), embedded in a *skin* membrane, the one which separates the system from its *environment*. A membrane without any membrane inside is called *elementary*. Each membrane defines a *region*. For an elementary membrane this is

the space enclosed by it, while the region of a non-elementary membrane is the space inbetween the membrane and the membranes directly included in it. Figure 1 illustrates these notions. We label membranes (by positive integers in Figure 1) in order to be able to address them in programming computations by membrane systems. Since each region is delimited ("from the outside") by a unique membrane, we will use the labels of membranes to also identify (label) the regions they delimit.

Each region contains a multiset of *objects*, and a set of (*evolution*) *rules*. The objects are represented by symbols from a given alphabet. Typically, an evolution rule from region r is of the form $ca \rightarrow cb_{in_j} d_{out} e_{here}$, and it "says" that a copy of the object *a*, in the presence of a copy of the *catalyst c* (this is an object which is never modified, it only assists the evolution of other objects), is replaced by a copy of the object *b* and two copies of the object *d*. Moreover, the copy of *b* has to enter "immediately" the inner membrane of region *r* labeled by *j* (hence to enter region *j*), one copy of object *d* is sent out through the membrane of region *r*, and one copy of *e* remains in region *r*. Note that the considered evolution rule can be applied in the region *r* only if this region includes the membrane *j*.

Membrane systems are *synchronous*, in the sense that a global clock is assumed, i.e., the same clock holds for all regions of the system. In each time unit a transformation of a *configuration* of the system takes place by applying the rules in each region, in a *nondeterministic* and *maximally parallel manner*. This means that the objects to evolve and the rules governing this evolution are chosen in a nondeterministic way; this choice is "exhaustive" in the sense that, after the choice was made, no rule can be applied anymore in the same evolution step (there are not enough objects available anymore for any rule to be applied now –this is the maximality of application).

In this way, one gets *transitions* between the configurations of the system. A sequence of transitions is called a *computation*. A configuration is *halting*, if no rule is applicable in any region. A computation is *halting* if it reaches a halting configuration. The *result* of a (halting) computation is the *number* of objects sent (through the skin membrane) to the environment during the computation.

Many modifications/extensions of this very basic model sketched above are discussed in the literature. We will briefly mention here only a few of them.

The first extension is to consider a priority relation among rules. This means that in each region a strict partial order relation on the set of rules from this region is given –then, a rule can be chosen (to process a multiset of objects) in a given step only if no rule of a higher priority is applicable.

Another useful "control device" is the possibility to modify the membrane permeability. Thus, a membrane can be made thinner (action δ) or thicker (action τ). A membrane of normal thickness is *dissolved* by action δ (the objects of a dissolved membrane remain in the region surrounding it, while the rules are removed; the skin membrane cannot be dissolved), or made impermeable (no object can pass through such a membrane) by action τ. An impermeable membrane is returned to normal thickness (hence it is again permeable) by action δ.

Many possibilities are offered by the communication commands. For instance, there are a number of ways of weakening the programming power provided by in_j: to

only indicate *in* (an object associated with this command has to enter any adjacently lower membrane; the choice of a membrane to enter is nondeterministic), to associate *electrical charges* both with objects and with membranes (a polarized object will enter the region of any adjacently lower membrane of the opposite polarization; the polarization of objects and of membranes may change during the computation).

In the above, the symbol-objects were processes by multiset rewriting-like rules (some objects are transformed into other objects, which have associated targets). Coming closer to the trans-membrane transfer of molecules, we can consider purely communicative systems, based on the three classes of such transfer known in the biology of membranes: *uniport*, *symport*, and *antiport*. Symport refers to the transport where two molecules pass together through a membrane in the same direction, antiport refers to the transport where two molecules pass through a membrane simultaneously, but in opposite directions, while the case when a molecule does not need a "partner" for a passage is referred to as uniport. In terms of P systems, we can consider object processing rules of the following forms: a symport rule (associated with a membrane i) is of the form (ab, *in*) or (ab, *out*), stating that the objects a and b enter/exit together membrane i, while an antiport rule is of the form (a, *out*; b, *in*), stating that, simultaneously, a exits and b enters membrane i; uniport corresponds to a "particular" case of symport rules, of the form (a, *in*), (a, *out*).

Another way of organizing computations by communication only is to use *carriers* (corresponding to "vectors" and to plasmids used in biochemistry), that is, to consider objects of two types: carriers ("vehicles") and passengers. As in the case of symport/antiport, no object ever changes. The passengers can pass through membranes only when carried by carriers. The used rules specify the way to attach passengers to carriers, the way these "aggregates" pass through membranes, and the way to detach passengers from carriers.

It is worth mentioning that in the case of symport/antiport and in the case of carriers no object is created or destroyed, only the location of the objects can be changed. Hence, the "conservation law" is observed – which does not necessarily happen in other classes of P systems. Also, in both these cases the environment is an active participant in the computation, holding as many copies of each object as necessary, and involved in a two-way communication with the skin region of the system.

Structuring the Objects – P Systems with String-Objects.

In a cell, many objects can be considered as being *atomic* (with no internal structure), but many other objects, such as, e.g., DNA molecules, have a structure, which, sometimes, can be described by a string. This leads to consider P systems where objects are strings – hence the evolution rules are based on string processing operations: rewriting, splicing, insertion, deletion, cut-and-paste, etc.

For instance, the rewriting rules are of the form ($X \rightarrow v$; *tar*), where $X \rightarrow v$ is a usual context-free rule and *tar* is a target indication, one of *here*, *out*, *in*, specifying in the standard way the region where the result of rewriting should go. We can also append to v the symbols δ and τ, which control the membrane thickness in the way discussed above, or we can consider a priority relation among rules.

A computationaly powerful idea is to combine the rewriting of strings with their duplication, considering rules of the form $r: a \to (u_1, tar_1) \mid\mid (u_2, tar_2)$. By applying r to a string $w = w'aw''$ we obtain the strings $w'u_1w''$ and $w'u_2w'$, which are sent to regions as indicated by the targets tar_1, tar_2, respectively.

An attractive variant is to process the string-objects by the splicing operation, and in this way a more uniformly biologically inspired device is obtained.

In the case of P systems with string-objects, the result of a computation can consists of all strings which are sent out of the system at any time during the computation (hence it is no longer necessary to work with halting computations), or –in the case when we take into consideration the number of strings (that is, we work with multisets of strings)– the result is the number of strings sent out during the computation. In the latter case, it is necessary to use string-processing operations which change the number of strings; rewriting and splicing does not have this property, but replication, and splitting (cutting a string into two strings, with local changes at the cutting place) can increase the number of strings. By sending strings out of the system or storing them in certain "garbage" membranes, we can also decrease the number of string-objects.

Computational Completeness and Universality.

As we have mentioned before, many classes of P systems, combining various ingredients described above, are able of simulating Turing machines, hence they are *computationally complete*. Note that in the case when we deal with P systems which compute numbers, we consider Turing machines as number recognizers; in the case of string-objects we can obtain the family of languages which are recognized by Turing machines (the recursively enumerable languages). Always, the proofs of results of this type are constructive, and this have the important consequence from the computability point of view that we can get *universal* (hence *programmable*) P systems: starting from a universal Turing machine (or an equivalent universal type-0 Chomsky grammar), we get an equivalent universal P system. This implies that in the case of Turing-complete classes of P systems, the hierarchy on the number of membranes always collapses (at most at the level of the universal P systems). Actually, the number of membranes sufficient in order to characterize the power of Turing machines by means of P systems is always rather small: in most cases, two or three membranes suffice. We can conclude that the compartmental computation in a membrane structure (using various ways of communicating among compartments) is rather powerful.

Computational Efficiency.

The computational power (the "competence") is only one of the important questions to be dealt with when defining a new computing model. The other fundamental question concerns the computing *efficiency*.

A deterministic Turing machine can be simulated in polynomial time by a family of deterministic P systems (see Pérez-Jiménez et al. 2002b), and a deterministic P system of the type $(Cat, Pri, tar, \delta, \tau)$ (hence working with symbol -objects, and using all features: catalysts, priorities, the control of membrane thickness, and addressing by *here*, *out*, *in*$_j$) can be simulated by a deterministic Turing machine with a polynomial slowdown (see Zandron et al. 2001). This means that by using such systems we cannot

solve exponential problems in polynomial time, in spite of the fact that exponentially many objects can be produced in linear time, for instance, by rules of the form $a \to aa$. Therefore, in order to improve the computational performance of our systems it is necessary to provide more e_cient ways for producing an exponential space. Three such ways have been considered so far in the literature, and *all of them were proven to lead to polynomial solutions of* NP*-complete problems.*

These three ideas are membrane division, membrane creation, and string replication.

Very briefly, in the case of membrane division one uses rules of the form $[_i a]_i \to [_i b][_i c]_i$ (the membrane with label i is divided, the contents of the former membrane is replicated in the two resulting membranes, with the exception of object a, which is replaced by b and c in the resulting membranes, respectively), in the case of membrane creation one uses rules of the form $a \to [_i b]_i$ (a new membrane, with label i, is created from object a), while in the case of string duplication one uses rules of the form $a \circledR u_1, \| u_2$ (from a string xay one passes to the strings xu_1y, xu_2y, maybe with targets associated with the resulting strings).

By using such operations, one can obtain an exponential workspace (in the form of membranes or string-objects) in a linear time, and in this way one can devise "P algorithms" which can solve NP-complete problems in polynomial (often, linear) time. This assertion was illustrated by SAT, the Hamiltonian path problem, the problem of inverting one-way functions, etc.

In the systems discussed up to now we have considered membrane structures which correspond to trees. An attractive generalization is to consider arbitrary graphs (in such a case, the "regions" associated with the nodes do not necessarily have a spatial counterpart in the form of a membrane structure – unless we consider direct communication among regions, corresponding to the inter-celular communication through common protein channels). A related possibility is to consider "tissue-like" P systems, with several elementary membranes swiming in a common environment, directly linked (hence communicating) or not.

The research in membrane computing area is rather active, but up to now no bioimplementation of a computation was reported in terms of P systems. In turn, several implementations of various classes of P systems on the electronic computer were reported; such implementations have a didactic interest, but not yet a practical interest (the electronic computer cannot support such important features of membrane systems as nondeterminism and parallelism).

However, theoretical applications of (notions and ideas central to) P systems were considered in several papers: to artificial life (Suzuki et al. 2001), for simulating the photosynthesis (Nishida 2001), to linguistics (Bel Enguix 2002).

This leads to considerations concerning the significance of P systems (for biology, for mathematics, and for computing). The approach is clearly motivated from a mathematical point of view, not only because it is natural to (try to) model the cell computational behavior, but also because the new computing model has a number of intrinsically interesting features. Examples of such features are: the use of multisets, the inherent parallelism, the possibility of devising computations which can solve ex-

ponential (intractable) problems in polynomial time (by making use of an exponential space created in a natural manner). At this moment, all these features are only potentially useful from a practical computational point of view. How should the implementation problem be approached? Should one try to develop, in laboratory, wet membrane computers (as this happens now in DNA computing), or should one try to implement P systems on electronic computers? We have mentioned that the latter approach has a long and quite successful tradition in natural computing, and perhaps, this will also be the case for membrane computing, possibly implemented on a dedicated architecture, specifically designed for P systems (as sometimes advocated).

5 Concluding Remarks

Our excursion through DNA and membrane computing was selective and superficial, without any attempt to enter into technical details –and also without any attempt to discuss the significance of this intellectual enterprise from a more general point of view. Maybe it is too early for addressing such questions, as, for the time being, we still have no answer to the basic question whether or not DNA and membrane computing will be useful for practical applications. However, what is already clear is that these domains are very attractive from a mathematical point of view, that they raise new problems, including questions about the very contents of the idea of computing, and suggest new computing paradigms, of a more natural type, which make necessary rethinking, perhaps, of basic computer science notions. Just an example: both theoretical and practical computer science are based on handling strings, in most cases in a local manner (by "rewriting"); however, we can compute at the level of Turing machines by splicing, which is an operation common in nature and completely different from rewriting; moreover, in the membrane computing area we compute at the universal level by using a data support with a rather weak structure, the multisets, again very close to the contents of the compartments of living cells. What about reconstructing the computability on the basis of multisets or of splicing? What about designing computers using the multisets or the splicing? Is this possible? Is this useful? Questions which are not in the scope of this paper.

BIBLIOGRAPHY

Adleman, L. M. (1994) "Molecular computation of solutions to combinatorial problems", *Science* 226, 1021–1024.

Andersson, J. A. (1996) *An Introduction to Neural Networks*, Cambridge, MA: The MIT Press.

Bel Enguix, G. (2002) "Preliminaries about some possible applications of P systems in linguistics" in *Preproceedings of Workshop on Membrane Computing*, Curtea de Arge¸s, Romania, 81–96.

Beyer, H.-G. (2001) *The Theory of Evolution Strategies*, Berlin: Springer.

Conrad, M. (1972), "Information processing in molecular systems", *Currents in Modern Biology* 5, 1–14.

Culik Culik II, K., Harju, T. (1991) "Splicing semigroups of dominoes and DNA", *Discrete Appl. Math.* 31, 261–277.

Hartmanis, J. (1994) "About the nature of computer science", *Bulletin of the EATCS* 53, 170–190.

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

Kleene, S. C. (1956), "Representation of events in nerve nets and finite automata", *Automata Studies*, Princeton, NJ: Princeton University Press, 3–42.

McCulloch, W. S., Pitts, W. H. (1943) "A logical calculus of the ideas immanent in nervous activity", *Bulletin of Mathematical Biophysics* 5, 115–133.

Păun, Gh. (1996a) "On the splicing operation", *Discrete Appl. Math.* 70, 57–79.

Păun, Gh. (1996b) "Regular extended H systems are computationally universal", *J. Automata, Languages, Combinatorics* 1, 27–36.

Păun, Gh. (2000) "Computing with membranes", *Journal of Computer and System Sciences* 61, 108–143.

Păun, Gh. (2002) *Membrane Computing. An Introduction*, Berlin: Springer.

Păun, Gh., Rozenberg, G., Salomaa, A. (1996) "Computing by splicing", *Theoretical Computer Science* 168/2, 321–336.

Pérez-Jiménez, M. J., Romero-Jiménez, A., Sancho-Caparrini, F. (2002a) *Teoría de la Complejidad en modelos de Computación Celular con membranas*, Sevilla: Ed. Kronos.

Pérez-Jiménez, M. J., Sancho-Caparrini, F. (2002) *Computación Celular con membranas*, Sevilla: Ed. Kronos.

Pérez-Jiménez, M.J., Romero-Jiménez, A., Sancho-Caparrini, F. (2002b), "Decision P systems and the P6=NP conjecture" in *Pre-proceedings of Workshop on Membrane Computing*, Curtea de Arge¸s, Romania, 345–354.

Pixton, D. (1996), "Regularity of splicing languages", *Discrete Appl. Math.* 69, 101– 124.

Suzuki, Y., Fujiwara, Y., Takabayashi, J., Tanaka, H. (2001) "Artificial life applications of a class of P systems: Abstract rewriting systems on multisets" in C. Calude, Gh. Păun, G. Rozenberg, A. Salomaa (eds.) *Multiset Processing. Mathematical, Computer Science, and Molecular Computing Points of View*, Lecture Notes in Computer Science, 2235, Berlin: Springer, 299–346.

Turing, A. M. (1936) "On computable numbers, with an application to the Entscheidungsproblem" in *Proceedings of the London Mathematical Society*, Ser. 2, 42, 230–265.

Zandron, C., Ferretti, C., Mauri, G. (2000) "Solving NP-complete problems using P systems with active membranes" in I. Antoniou, C. S. Calude, M. J. Dinneen (eds.) *Unconventional Models of Computation*, London: Springer, 289–301.

Gheorghe Păun is Ramon y Cajal Researcher at Rovira i Virgili University. He works in theory of formal languages, molecular computing based on DNA and cellular computation with membranes (model created by himself in October 1998). He is member of European network MolCoNet and permanent member of the Institute of Mathematics of Academy of Romania since 1994. He has published around 400 articles in prestigious scientific journals (almost 300 of them are reviewed in the Mathematical Reviews). He has published 22 books about Mathematics and Computation, and 14 books about popular mathematics.

Address: Institute of Mathematics of the Romanian Academy, PO Box 1-764, 70700 Bucureºti, Romania, and Rovira i Virgili University, Pl. Imperial Tàrraco 1, 43005 Tarragona, Spain. E-mail: gpaun@imar.ro, gp@astor.urv.es

Mario de Jesús PÉREZ JIMÉNEZ is lecturer in Computer Science and Artificial Intelligence area at University of Seville, where he manages the Group of Research in Natural Computation. He works in not conventional models of computation. He has been responsible of a project supported by European Community (Finite Model Theory and Bounded Arithmetic) and nowadays of another supported by McyT (Desarrollo, verificación y automatización de modelos moleculares y celulares con membranas). He has published 8 books of Mathematics and Computation, and more than 60 scientific articles in national and international journals. He is member of European Molecular Computing Consortium.

Address: Department of Computer Science and Artificial Intelligence, Sevilla University, Avenida Reina Mercedes s/n, 41012 Sevilla, Spain. E-mail: Mario.Perez@cs.us.es