GHEORGHE PĂUN

(A tribute in honour of his 60th birthday)

We know him for a whole life. At first it was just a name, often found in journals – particularly in the abstract journals, because in the '70s we did not have too much access to literature from abroad, or even from the neighboring country. "I have a dream", – confessed one of colleagues – "to visit Bucharest, Gheorghe Păun." For those times a research visit, especially from the Moldavian Soviet Socialist Republic to Romania could be just a dream.

The dream was realized much later, only in 1996, when to our colleague that had a speech in the plenary conference, came a tall, handsome man, addressing his congratulations and expressing interest in what he had heard. ”My name is Gheorghe Păun”, – he introduced himself. Thus began our collaboration, which already lasts for 15 years and has brought fruitful results. A testimony to this collaboration is exposed on the pages of this issue, which we devote to the 60 years anniversary of the eminent scientist and cultural figure, Corresponding Member of the Romanian Academy, Member of the Romanian Writers Association, Gheorghe Păun.

Gheorghe Păun had a powerful influence on the development of theoretical computer science, especially the area of natural computing. He is the inventor of the new rapidly developing area of biocomputing – computing using membrane systems, or P systems. Due to the results and ideas of Gheorghe Păun in biocomputing the team of scientists working successfully in this direction was formed in the Institute of Mathematics and Computer Science. Gheorghe Păun is not only an outstanding scientist, but he is an active promoter of science, writes and publishes collections of poetry, publishes science fiction novels and books. We are very grateful that he, being so much busy, has time to be the member of editorial board of ”Computer Science Journal of Moldova”, and it is a great honour for us.

We wish Gheorghe Păun long fruitful creative life, opening new roads in science. And we are happy to follow him.

Editorial board of Computer Science Journal of Moldova

Computer Science Journal of Moldova, vol.**18**, no.2(53), 2010

Investigations on Natural Computing in the Institute of Mathematics and Computer Science

Artiom Alhazov Elena Boian Liudmila Burtseva
Constantin Ciubotaru Svetlana Cojocaru
Alexandru Colesnicov Valentina Demidova
Sergiu Ivanov Veaceslav Macari Galina Magariu
Ludmila Malahova Vladimir Rogojin Yurii Rogozhin
Tatiana Tofan Sergey Verlan Tatiana Verlan

Abstract

We describe the investigations on natural computing in the Institute of Mathematics and Computer Science of the Academy of Sciences of Moldova during last fifteen years. Most of these investigations are inspired by results and ideas belonging to Corresponding Member of the Romanian Academy **Gheorghe Păun.**

1 Introduction

In this paper we present a short overview of investigations on natural computing carried out in the Institute of Mathematics and Computer Science of Academy of Sciences of Moldova during last fifteen years.

Exactly fifteen yeas ago one of the authors of this paper, Prof. Yurii Rogozhin has started his study in the scope of natural computing. His studies and studies of his colleagues in the Institute were inspired by scientific activity of Corresponding Member of the Romanian Academy Gheorghe Păun.

This overview includes investigations at the Institute on DNA computing, membrane computing, insertion-deletion systems and other models of biocomputing. The most of results obtained by scientists of the Institute of Mathematics and Computer Science were presented at different international workshops and conferences on natural computing and published in prestigious international journals. Three PhD theses and one Habilitation thesis were defended since 2004 and their results are reflected in this volume of the journal.

2 DNA Computing

2.1 Test Tubes Systems

Molecular computers have been attracting many people from chemistry, biology and computer science. A major break through was a concrete molecular computer by Adleman [1] that could solve instances of the travelling-salesman-problem.

In a remarkable paper T. Head [59] draw the connections between molecular computers and formal language theory. The molecules from biology are replaced by words over a finite alphabet and the chemical reactions are replaced by the *splicing* operation. An H system specifies a set of rules used to perform splicing and a set of initial words or axioms. A splicing rule may be applicable to two molecules. It breaks both molecules at fixed locations, defined by the splicing rule, and recombines the initial string of one broken molecule with the final string of the other one. The computation is done by applying iteratively the rules to the set of words until no more new words can be generated. This corresponds to a bio-chemical experiment where we have enzymes (splicing rules) and initial molecules (axioms) which are put together in a tube and we wait until the reaction stops.

T. Head's smooth connection to formal language theory brought this field to the attention of many people from formal language theory. E.g., Gh. Păun, G. Rozenberg and A. Salomaa [89] asked what classes of formal languages are derivable by molecular computers depending on certain classes in formal language theory that the initial molecules and enzymes belong to.

One of such results says that any regular language is derivable from finitely many initial molecules with finitely many splicing rules. E. Csuhaj-Varjù, L. Kari, Gh. Păun [47] modified T. Head's concept slightly to systems of n test tubes. Here, any test tube is an H-system with an additional filter. In a single macro-step any test tube generates new molecules according to its set of starting molecules and its set of splicing rules. Afterwards, the outcome of all test tubes is poured into the filters of all test tubes. Those molecules that may pass the filter of test tubes $i, 1 \leq i \leq n$, form the new starting molecules for the i-th test tube for the next macro-step.

This new process, filtering results of one test tube into another, increases the computational capability of molecular computers. Let us call a system of n test tubes *finite* if initially any test tube contains (arbitrarily many) copies of molecules from a finite set of molecules and possesses only finitely many splicing rules.

It is known [89, 1] that a finite 1-test-tube-system generates only regular sets of molecules. However, finite 2-test-tube-system may generate more complicated non-regular sets [47]. C. Ferretti, G. Mauri, C. Zandron [54] have shown that any recursively enumerable (r.e.) set of molecules is derivable in a finite 9-test-tube-system (or in a finite 6-test-tube-system if one allows for a rather simple encoding of the molecules to be generated).

These results have implications for molecular computers as r.e. languages have many undecidable properties. E.g., the membership problem is in general not decidable for r.e. languages. This means that there exists no algorithm $\mathcal{A}$ which can tell, when presenting a word w and an r.e. languages L to $\mathcal{A}$, whether w belongs to L or not. Further, there is a fixed language, U, such that there exists no algorithm $\mathcal{A}$ which can tell, when presenting a word w to $\mathcal{A}$, whether w is an element of U or not.

Thus, a trivial consequence of the result of [54] is that there exists no algorithm which can compute which molecules may be generated in a finite 6-test-tube-system. I.e., the results of a finite 6-test-tube-system cannot be algorithmically predicted in general. We improved this result by showing how to generate any r.e. language in a finite 3-test-tube-system [96]. Thus, there is no way to predict the outcome of the reactions of only three test tubes starting with molecules and enzymes from finite set of molecules and enzymes.

This question is still open for finite 2-test-tube-systems.

2.2 TVDH Systems

Head splicing systems (H systems) were one of the first theoretical models of biomolecular computing and they were introduced by T. Head [59].

Ordinary H systems are not very powerful and a lot of other models introducing additional control elements were proposed. One of these well-known models are time-varying distributed H systems (TVDH systems) introduced by Gh. Păun in [90] as another theoretical model of biomolecular computing, based on splicing operations.

He started from the biological observation that at each moment there is a set of active enzymes which behave depending on conditions of the environment. If the environment (temperature, acidity or other parameter) changes, then the set of active enzymes also change. In the proposed model, the set of splicing rules changes periodically. More exactly, the model contains a set of words, the axioms, and a finite number of sets of splicing rules, the components. At each step, the current words are spliced once using the rules of the current component, and the result of this splicing forms the set of words for the next iteration.

We remark that this elimination procedure is very powerful and it permits to obtain a big computational power. If the elimination procedure is changed by permitting several splicings to be applied, then another model similar to TVDH systems is obtained: Enhanced Time-Varying distributed H systems, see [73, 75, 103, 104].

It is worthy to note that TVDH systems with one component generate all RE languages. This result highlights the importance of the elimination and shows that for ordinary H systems only small modifications are needed in order to pass from regularity to rationality, see also [105].

A study of TVDH systems from a computational point of view may be found in [102] where TVDH systems for main arithmetic operations (addition, multiplication, exponentiation and division) and for the Ackermann function were explicitly given. A computer simulator of TVDH systems was also developed [101].

Moreover, the simple structure of TVDH systems permit to use these systems as target for universality proofs for systems based on splicing. For example, in [79] a simulation of the TVDH system from [78] is done. In [106] there is an example of simulation of TVDH systems with one component by splicing membrane systems (P systems) [100].

TVDH systems have a very simple structure and a powerful control. These features stimulated several articles investigating the computational power of these systems. In [92, 91] Gh. Păun showed that TVDH systems are computationally complete by constructing a system having 7 components that simulate any type-0 grammar.

Subsequent articles decreased consequently the number of components needed to obtain the computational completeness. This was done in two ways. In the first case TVDH systems simulating Turing machines and tag systems were constructed. We remark that in this case the proof is strictly sequential: only one molecule which encodes the tape (or the working word) and the state of the machine shall be present in the system. In 1998 M. Margenstern and Yu. Rogozhin showed first that TVDH systems with 2 components are able to do universal computations, see [70, 71]. Their proof was based on a simulation of tag systems [45, 82] and the obtained system is quite complicated. Later, a proof based on a simulation of Turing machines was proposed [71].

Shortly after that the same authors showed that with 2 components it is possible to generate all recursively enumerable languages [72, 74]. Such generation was done in the following way. It is known that for any RE language $\mathcal{L} = \{w_1, w_2, \dots\}$ there is a Turing machine that, given an input 01^n, $n > 0$, will compute $01^{n+1}w_n$. The system that they constructed behaves in the following way. First, a simulation of corresponding Turing machine on the input 01^n is done. After that special rules cut off word w_n and the system restarts the simulation of the Turing machine on the new input 01^{n+1}. In this way any RE language is generated word by word.

The same authors obtained in 2001 a very important result: TVDH systems with one component are universal [77]. The core of the proof consists in a simulation of tag systems. In the same year they proposed a system having one component that generates all RE languages [76]. The proof is based on the same ideas that were presented in previous paragraph.

Another way to show the computational completeness of TVDH systems consists in simulation of type-0 grammars. This introduces a parallelism in computations because several evolutions of molecules are made in the same time. Also this case is more complicated than the sequential one and needs much more accuracy. Almost all results in this case are based on "rotate-and-simulate" method.

The article of Gh. Păun [92] is an example of this technique. In 1999 A. Păun showed that it is possible to simulate the work of an arbitrary grammar with 4 components [88]. This result was improved by M. Margenstern and Yu. Rogozhin who showed that a type-0 grammar can be simulated with 3 components, see [73].

We improved the above results by showing that it is possible to generate any recursively enumerable language in a parallel way with 2 components, see [78]. Finally, the final point was reached by showing that it is possible to generate all recursively enumerable languages in a parallel way with one component, see [79]. The last result was obtained by using the method of directing molecules, see [104], which is a modification of the "rotate-and-simulate" method for systems based on splicing.

3 Membrane (P) Systems

The research area of membrane computing originated as an attempt to formulate a model of computation motivated by the structure and functioning of a living cell - more specifically, by the role of membranes in compartmentalization of living cells into protected reactors.

Therefore, initial models were based on a cell-like (hence hierarchical) arrangement of membranes delimiting compartments, where multisets of chemicals (called objects) evolve according to given evolution rules. These rules were either modeling chemical reactions and had the form of (multiset) rewriting rules, or they were inspired by other biological processes, such as passing objects through membranes (either in symport or antiport fashion), and had the form of communication rules. These initial models were then modified by incorporating various additional features motivated by considerations rooted in biology, mathematics, or computer science.

The next important step in the development of research in membrane computing was to also consider other (nonhierarchical) arrangements of membranes. While hierarchical (cell-like) arrangements of membranes correspond to trees, tissue-like membrane systems consider arbitrary graphs as underlying structures, with membranes placed in the nodes while edges correspond to communication channels (see [94, 87]).

3.1 Transitional P Systems

We introduced a new approach to study the family of languages generated by the transitional membrane systems without cooperation and without additional ingredients ([28, 29, 30, 31]). The fundamental nature of these basic systems makes it possible to also define the corresponding family of languages in terms of derivation trees of context-free grammars. We also compare this family to the well-known language families and discuss its properties.

We considered some theoretical tasks for P systems. In particular, we considered a new variant of the halting condition in P systems ([20, 55]), i.e., a computation in a P system is already called halting if not for all membranes a rule is applicable anymore at the same time, whereas usually a computation is called halting if no rule is applicable anymore in the whole system. This new variant of partial halting is especially investigated for several variants of P systems using membrane rules with permitting contexts and working in different transition modes, especially for minimal parallelism.

Both partial halting and minimal parallelism are based on an arbitrary set of subsets from the set of rules assigned to the membranes.

We considered the problem of synchronizing the activity of all membranes of a P system ([9, 23]). After pointing at the connection with a similar problem dealt with in the field of cellular automata where the problem is called the firing squad synchronization problem, FSSP for short, we provided two algorithms to solve this problem. One algorithm is non-deterministic and works in $2h + 3$ steps, the other one is deterministic and works in $3h + 3$ steps, where h is the height of the tree describing the membrane structure. We introduced a new derivation mode for P systems that permits to make a look-ahead on the next configuration and check for some forbidding conditions on it [108]. The interesting point is that the software implementation of this mode needs very small modifications to the standard algorithm of rule assignment for maximally parallelism. As benefits of this mode some non-deterministic proofs become deterministic.

As an example we present a generalized communicating P system that accepts numbers 2^n in n steps in a deterministic way. Another example shows that in the deterministic case this mode is more powerful than the maximally parallel derivation mode. Finally, this mode gives a natural way to define P systems that may accept or reject a computation.

3.2 Communication P Systems

Communication P systems [52, 56, 98, 110] are inspired by the idea of communicating substances through membrane channels of a cell. Molecules may go in the same direction together – *symport* – or some of them may leave while at the same time other molecules enter the cell – *antiport*. Communicating objects between membrane regions is a powerful tool yielding computational completeness with one membrane using antiport rules or symport rules of size three, i.e. involving three objects, in the maximally parallel mode. As register machines can be simulated in a deterministic manner, P systems with antiport rules or symport rules can accept any recursively enumerable set of (vectors of) natural numbers in a deterministic way.

In tissue P systems, the objects are communicated through channels between cells. In each transition step we apply only one rule for each channel, whereas at the level of the whole system we work in the maximally parallel way. Computational completeness can be obtained with a rather small number of objects and membranes or cells, in the case of tissue P systems even with copies of only one object.

The computational power of P systems with antiport rules or symport rules involving copies of only one object remains one of the most challenging open questions.

The concept of P systems with antiport and/or symport rules can be generalized to systems using membrane rules evolving multisets of objects on both sides of the membrane even depending on permitting contexts (also called promoters) and/or forbidden contexts (also called inhibitors).

P systems with communication rules can also be used as language generators – we take the sequences of terminal objects sent out to the environment as the strings generated by the system. A generalized communicating P system, or a GCPS for short, corresponds to a graph where each node, called a cell, contains a multiset of objects which – by communication – may move between the cells.

The communication rules are rather restricted, any rule identifies four cells, two input cells and two output cells, such that a pair of objects from the two input cells moves synchronously to the two output cells. The form of a communication rule is $(a;i)(b;j) \rightarrow (a;k)(b;l)$ where a and b are objects and i,j,k,l are numbers that identify the input and the output cells. Such a rule means that an object a from cell i and an object b from cell j move synchronously to cell k and cell l, respectively. It can easily be seen that these very simple communication rules can also be interpreted as interaction rules.

Depending on the relation of i,j,k,l, nine restricted variants of communication rules (modulo symmetry) can be distinguished. (For example, $i \neq j \neq k \neq l$ is one of these restrictions, called a parallel-shift rule). When the GCPS has only one type of these restricted rules, we speak of generalized communicating P systems with minimal interaction, a GCPSMI for short.

We considered generalized communicating P systems which use only one type of the above interaction operations [51]. We proved that in 7 of these cases computational completeness is obtained, i.e., the corresponding GMPCSs are able to determine any recursively enumerable set of non-negative integers; the only exception determines only finite singletons of natural numbers.

The constructions in the proofs also demonstrate that this large expressive power can be obtained by P systems with relatively small numbers of cells and simple graph architectures. We also proved [52] that GCPSs still remain computationally complete if they are given with a singleton alphabet of objects and with one of the restricted types of rules: parallel-shift, join, presence-move, and chain.

3.3 Polymorphic P systems

We introduced a variant of the multiset rewriting model of P systems where the rules of every region are defined by the contents of interior regions, rather than being explicitly specified in the description of the system [33]. This idea is inspired by the von Neumann's concept of "program is data" and also related to the research direction proposed by Gh. Păun about the cell nucleus.

Membrane computing is a fast growing research field opened by Gh. Păun in 1998. It presents a formal framework inspired from the structure and functioning of the living cells.

In this paper we define yet another, relatively powerful, extension to the model, which allows the system to dynamically change the set of rules, not limited to some finite prescribed set of candidates. There are three motives for this extension:

- first, our experience shows that "practical" problems need "more" computing potential than just computational completeness;
- second, we attempt to import a very important computational

ingredient into P systems, this time from the conventional computer science;

- third, this extension correlates with the biological idea that different actions are carried out by different objects, which can be acted upon as well.

Let us first explain these motives. Most papers of the field belong to the following categories:

1. introducing different models and variants,
2. studying the computational power of different models depending on what ingredients are allowed and on the descriptional complexity parameters,
3. studying the computational efficiency of solving intractable problems (supercomputing potential) depending on the ingredients,
4. using membrane computing to represent and model various processes and phenomena, including but not limited to biology,
5. other applications.

There is a surprisingly big gap between the sets of ingredients needed to fulfill requirements in directions 2, 3, and the sets of ingredients demanded by other applications. For instance, very weak forms of cooperation between objects are often enough for the computational completeness, but many "practical" problems cannot be solved in a satisfactory way under the same limitations. This leads to the following question. What is implicitly required in most "practical" problems? We will mention just a few of these requirements below.

A) Determinism or at least confluence. Clearly, the end user wants to obtain the answer to the specified problem in a single run of a system instead of examining infinitely many computations. This is a strong constraint, e.g., catalytic P systems and P systems with minimal symport/antiport are universal, while in the deterministic case non-universality is published for the first ones and claimed for the latter ones. Informally speaking, less computational power is needed to just compute the result than it is to also enforce choice-free behavior of the system.

B) Input/output. Most of the universality results are formulated as generating languages or accepting sets of vectors, or in an even more restricted setup. There is no need to deal with input in the first case, and in the latter case the final configuration itself is irrelevant (except yes or no in case of the efficiency research). On the other side, both input and output are critical for most applications.

C) Representation. Clearly, any kind of discrete information can be encoded in a single integer in some consistent way. However, a much more transparent data representation is typically required; even the intermediate configurations in a computation are expected to reflect a state of the object in the problem area.

D) Efficiency. Suppose numbers are represented by multiplicities of certain objects. The number of steps needed to multiply two numbers by plain (cooperative) multiset processing is proportional to the result. If the multiset processing can be controlled by promoters/inhibitors/priorities, then the number of steps needed for multiplication is proportional to one of the arguments. However, many applications would ask for a multiplication to be performed in a constant number of steps. Similar problems appear for string processing.

E) Data structures. Membrane computing deals with multisets distributed over a graph, while conventional computers provide random memory access and pointer operations, allowing much more complex structures to be built.

Some of these implicit requirements originate because the user wants a solution which is at least as good as the one that can be provided by conventional computers.

We introduced a new feature into the membrane computing. This time the inspiration is not biological, but rather is from the area of conventional computing.

Suppose we want to be able to manipulate the rules of the system during its computation. A number of papers has been written about this but in most of them the rules are predefined in the description of the system. The most natural way to manipulate the rules is to represent them as data, treat this data as rules, and manipulate it as usual in P systems, in the spirit of von Neumann's approach. In membrane systems, the data consists of multisets, so objects should be treated as description of the rules. Informally, a rule j in a region i can be represented by the contents of membranes jL and jR inside i. Changing the contents of regions jL and jR results in the corresponding change of the rule j. We call such P systems polymorphic, by analogy with polymorphic or self-modifying computer programs.

At the same time, if a membrane system is an abstraction inspired by the biological cell, one can view inner regions as an abstraction inspired by the cell nucleus; their contents correspond to the genes encoding the enzymes performing the reactions of the system.

The simplicity of the proposed model is that we consider the natural encoding, i.e., no encoding at all: the multisets describing the rules are represented by exactly themselves. Therefore, we are addressing a problem informally stated by Gh. Păun "Where Is the Nucleus?" by proposing a computational variant based on one simple difference: the rules are taken from the current configuration rather than from the description of the P system itself.

3.4 Insertion-Deletion (P) Systems

We considered models of biocomputing based on insertion and deletion operations of small size [80, 21, 22, 35, 36, 57, 68, 65, 66, 64, 67, 69, 81]. The insertion and the deletion operations originate from the language theory, where they where introduced mainly with linguistic motivation. In general form, an insertion operation means adding a substring to a given string in a specified (left and right) context, while a deletion operation means removing a substring of a given string from a specified (left and right) context. A finite set of insertion-deletion rules, together with a set of axioms provide a language generating device: starting from the set of initial strings and iterating insertion-deletion operations as defined by the given rules we get a language.

In the last years, the study of these operations has received a new motivation from molecular computing, because, from the biological point of view, insertion-deletion operations correspond to mismatched annealing of DNA sequences. As expected, insertion-deletion systems are quite powerful, leading to characterizations of recursively enumerable languages. This is not quite surprising as the corresponding device contains two important ingredients needed for the universality: the context dependency and the erasing ability. However, as it was shown in [80], the context dependency may be replaced by insertion and deletion of strings of sufficient length, in a context-free manner. If the length is not sufficient (less than two) then such systems are decidable and a characterization of them was shown by S.Verlan in [107].

Similar investigations were continued in [5, 4] on insertion-deletion systems with one-sided contexts, i.e. where the context dependency is present only from the left (right) side of all insertion and deletion rules. These articles also give some combinations of rule parameters that lead to systems, which are not computationally complete. However, if these systems are combined with the distributed computing framework of P systems, then their computational power may strictly increase [64, 68].

In [21, 35] we study P systems with context-free insertion and deletion rules of one symbol. We show that this family is strictly included in MAT, however some non-context-free languages may be generated. If Parikh vectors are considered, then the corresponding family equals to PsMAT. When a priority of deletion over insertion is introduced, PsRE can be characterized, but in terms of language generation such systems cannot generate a lot of languages because there is no control on the position of an inserted symbol. If one-sided contextual insertion or deletion rules are used, then this can be controlled and all recursively enumerable languages can be generated.

The same result holds if a context-free deletion of two symbols is allowed.

3.5 Splicing (P) Systems

It is known that H systems are not very powerful, so, a lot of other models introducing additional control elements were proposed. Another extension of H systems was done using the framework of P systems (see [109]). In a formal way, splicing P systems can be considered like a graph, whose nodes contain sets of strings and sets of splicing rules. Every rule permits to perform a splicing and to send the result to some other node. Since splicing P systems generate any recursively enumerable language, it is clear that there are universal splicing P systems.

Like for small universal Turing machines, we are interested in such universal systems that have a small (smallest) number of splicing rules. A first result was obtained by Yu.Rogozhin and S.Verlan in [97] where a universal splicing P system with 8 rules was shown. Similar investigations for P systems with symbol-objects were done in [11, 39] and the latter article constructs a universal antiport P system with 23 rules.

In [38] we provided a new construction for splicing P systems and proved the remarkable fact that 6 splicing rules are powerful enough for the universality. In [34] we presented a series of small universal devices (splicing systems):

- two universal time-varying distributed H systems: of degree 2 with 15 rules and of degree 1 with 17 rules,

- and also three universal splicing test tube systems with 3 or 2 test tubes and 10 rules.

Test tube systems based on splicing, introduced in E. Csuhaj-Varjú et al. in [47] are symbol processing mechanism with components (test tubes) working as splicing schemes in the sense of T. Head, and communicating through redistribution of the contents of the test tubes via filters. These systems with finite initial contents of the tubes and finite sets of splicing rules associated to each component are computationally complete; they characterize the family of recursively enumerable languages. The existence of universal test tube distributed systems was obtained on this basis, hence there is the theoretical possibility to design universal programmable computers with the structure of such a system.

Since 1996, a lot of variants of test tube systems have been introduced and studied, using filtering of the strings by patterns. A natural question is whether or not such systems with other types of filtering mechanisms, based on techniques widely used in laboratory, can be defined. We introduced a new variant of test tube systems, length-separating test tube systems, based on splicing where the communication of the words among the test tubes is based on filtering by their lengths, motivated by the gel electrophoresis laboratory technique [50]. However, we remark that filtering by length can also be done by other methods, like size exclusion chromatography, that permits to separate molecules depending on their size.

Gel electrophoresis is a technique for separation of molecules which is widely used in the laboratory. It is usually performed for analytical purposes at the final stage of the experiment. Its formal counterpart, the length separation, is a standard tool in DNA computing. For example, it is used in the third step of the Adleman's experiment in 1994. There are also algorithms like in F. Guarnieri et al. [58] based on the length separation which use it at the end of the computation in order to confirm or select the result. In Y. Khodor et al. [63], a method called length-only discrimination based on the generate-and-search approach but relying on the length of the sequence is presented and experimentally confirmed.

We use the length separation in another way. We consider a distributed system and we use the length separation for the communication between the components of the system. Since such systems work iteratively, the length separation, used at each step, becomes one of the main ingredients of the model. In an informal way, our model corresponds to the following experiment.

Let us suppose that there is a set of test tubes. Each of these test tubes may transform DNA molecules (cut, ligate, multiply etc). The tubes are selective and they can do their transformations only on specific molecules (for example, in a tube DNA molecules may be cut with a specific enzyme, hence only molecules having a corresponding site will be modified). Taking a tube, we may put some amount of DNA molecules into it. After the transformation, all molecules from a tube are put in a gel electrophoresis. After the separation, the gel is cut at some points corresponding to some molecular lengths. Hence, molecules will be grouped by some length intervals. After that, molecules are extracted from the gel and distributed among other test tubes depending on their molecular length interval.

The above process may be iterated and, since all tubes are organized in a network, some interesting transformations may be done. The initial DNA molecules are put in some fixed tube, and the transformed molecules are collected in the output tube. To separate molecules to be transmitted from one tube to another one, we define different conditions. These conditions are exclusive, namely, each string (molecule) found in a tube can be forwarded to only one tube. All molecules in the test tube are communicated to some tube (depending on the underlying graph of the test tube system, might remain at the original tube). One type of these conditions are variants of communication by fixed (bounded) length where strings of length equal to (or at most equal to or at least equal to) a fixed constant are communicated to another tube. In terms of gel electrophoresis this corresponds to the cut at some specific points depending on the marker molecules.

Other types of conditions involve molecules of maximal and minimal size as well as their negations (not maximal and not minimal). From the gel point of view, this corresponds to the selection of the first or the last molecule from the gel (the other molecules correspond to the negation variant). We study the computational power of these constructs. We show that the length separating test tube systems, even with very restricted size parameters, are able to simulate the Turing machines. This result holds in general case as well as for systems only using communication conditions based on separation of molecules by maximal (resp. minimal) and not maximal (resp. not minimal) length and on the ability checking whether the word (the molecule) differs from the empty word. These results correspond to our expectations, due to the nature of the splicing operation.

If we restrict the communication conditions to select molecules with fixed or bounded length, then the computational power of the corresponding systems is not known. Although using appropriate communication predicates our construction has the power of the Turing machines, this does not help in efficiently solving practical problems. For example, given a particular molecule, can we design a system that will perform a particular transformation on it? Moreover, this transformation should be efficient, i.e., it shall be done in the smallest possible number of steps, involving the smallest number of high-cost operations. This problem is difficult to solve.

Here we provided a theoretical framework that could be used to describe and possibly answer the above questions. However, we mainly focus on the network structure and length filtering; the operation-related improvements remain to be further investigated.

3.6 Reversibility and P Systems

Membrane computing is a formal framework of distributed parallel computing. We studied the reversibility and maximal parallelism of P systems from the computability point of view [32, 24, 25, 37]. The notions of reversible and strongly reversible systems are considered.

The universality is shown for reversible P systems with either priorities or inhibitors, and a negative conjecture is stated for reversible P systems without such control. Strongly reversible P systems without control have shown to only generate sub-finite sets of numbers; this limitation does not hold if inhibitors are used.

Another concept considered is strong determinism which is a syntactic property, as opposed to the determinism typically considered in membrane computing. Strongly deterministic P systems without control only accept sub-regular sets of numbers, while systems with promoters and inhibitors are universal.

3.7 P Systems with Active Membranes

Membrane systems are a convenient framework of describing polynomial-time solutions to certain intractable problems in a massively parallel way. Division of membranes makes it possible to create an exponential space in linear time, suitable for attacking problems in NP and even in PSPACE. Their solutions by so-called P systems with active membranes have been investigated in a number of papers since 2001, later focusing on solutions by restricted systems (see, for example, The Oxford Handbook of Membrane Computing, ed. by G. Paun, G. Rozenberg, A. Salomaa [87]). The description of rules in P systems with active membranes involves membranes and objects; the typical types of rules are object evolution, object communication, membrane dissolution, membrane division.

Our goal was to implement methods of P systems with active membranes in **computer algebra** and particularly, to generalize the approach from decisional problems to the computational ones, by considering a #P-complete (pronounced sharp-P complete) problem of computing the permanent of a binary matrix [5, 14].

Commutative and non-commutative Computer Algebra Systems were analyzed. The systems were analyzed taking into account efficiency, termination of calculations, and some technical details of their implementation. Existing methods of parallel calculations used with Computer Algebra Systems were also examined. We studied a series of problems in the matrix theory (permanent calculation), Grobner base theory (determination if the algebra has the finite dimension, checking if a given set of polynomials forms the Grobner base of the given algebra). We used P-lingua system [95] to simulate elaborated algorithms. We found that the said system does not meet all necessities to simulate monomials manipulations. We proposed to extend it with the mechanism of rule parameterization.

The domain of Computer Algebra is characterized by problems of the high calculation complexity. Therefore, the interest in effective methods of their solution is justified. The results of our research demonstrate that natural calculations make an effective mechanism to solve problems in this domain, in particular, for non-commutative algebras where the computing processes can be infinite.

Other our goal was to implement methods of P systems with active membranes in **mathematical linguistics**.

Solving most problems of natural language processing is based on using certain linguistic resources, represented by corpora, lexicons, etc. Usually, these collections of data constitute an enormous volume of information, so processing them requires much computational resources. A reasonable approach for obtaining efficient solutions is that based on applying parallelism; this idea has been promoted already in 1970's. Many of the stages of text processing (from tokenization, segmentation, lematizing to those dealing with natural language understanding) can be carried out by parallel methods. This justifies the interest to the methods offered by the biologically inspired models, and by membrane computing in particular.

However, there are some issues that by their nature do not allow complete parallelization, yet exactly they are often those "computational primitives" that are inevitably used during solving major problems, like the elementary arithmetic operations are always present in solving difficult computational problems. Among such "primitives" in the computational linguistics we mention handling of the dictionaries, e.g., dictionary lookup and dictionary update. Exactly these problems constitute the subject of our work.

In our approach we speak about dictionary represented by a prefix tree and P systems with active membranes that are a convenient framework of describing computations on trees [17, 16, 46]. In [13, 15, 41, 42, 43] we formalised inflection process for the Romanian language using the model of P systems with cooperative string replication rules, which will make it possible to automatically build the morphological lexicons as a base for different linguistic applications.

3.8 Networks of Evolutionary Processors

Motivated by some models of massively parallel computer architectures (see [53] and [60]), networks of language processors have been introduced in 1997 by E. Csuhaj-Varjú and A. Salomaa [48]. Such a network can be considered as a graph where the nodes are sets of productions and at any moment of time a language is associated with a node. In a *derivation* step, any node derives from its language all possible words as its new language. In a *communication* step, any node sends those words to other nodes that satisfy an output condition given as a regular language, and any node takes those words sent by the other nodes that satisfy an input condition also given by a regular language. The language generated by a network of language processors consists of all (terminal) words which occur in the languages associated with a given node.

Inspired by biological processes, J. Castellanos, C. Martín-Vide, V. Mitrana and J. Sempere introduced in [44] a special type of networks of language processors which are called networks with evolutionary processors because the allowed productions model the point mutation known from biology. The sets of productions have to be substitutions of one letter by another letter or insertions of letters or deletion of letters; the nodes are then called substitution node or insertion node or deletion node, respectively.

It was shown by A. Alhazov et al. in [3] that networks of evolutionary processors are universal in that sense that they can generate any recursively enumerable language and that networks with three nodes are sufficient to get all recursively enumerable languages. The proof uses one node of each type (and intersection with a monoid). Therefore it is a natural question to study the power of networks with evolutionary processors where the nodes have only two types, i. e.,

(i) networks with deletion nodes and substitution nodes (but without insertion nodes),

(ii) networks with insertion nodes and substitution nodes (but without deletion nodes), and

(iii) networks with deletion nodes and insertion nodes (but without substitution nodes).

We investigated the power of such systems and studied the number of nodes sufficient to generate all languages which can be obtained by networks of the type under consideration. We prove that networks of type (i) and (ii) produce only finite and context-sensitive languages, respectively. Every finite, context-sensitive or recursively enumerable language can be generated by a network of type (i) with one node, by a network of type (ii) with two nodes or by a network of type (iii) with two nodes, respectively [19].

Particularly interesting variants of these devices are the so-called hybrid networks of evolutionary processors (HNEPs), where each language processor performs only one of the above operations on a certain position of the words in that node. Furthermore, the filters are defined by some variants of random-context conditions, i.e., they check the presence/absence of certain symbols in the words. These constructs can be considered both language generating and accepting devices, i.e., generating HNEPs (GHNEPs) and accepting HNEPS (AHNEPs).

In [49] E. Chuhaj-Varjú et al. showed that, for an alphabet V, GHNEPs with $27 + 3 \cdot card(V)$ nodes are computationally complete. A significant improvement of the result can be found in [6, 7], where we proved that GHNEPs with 10 nodes (irrespectively of the size of the alphabet) obtain the universal power. Recently [8, 18] we improved this result and showed that any recursively enumerable language can be generated by a GHNEP having 7 nodes and it can be accepted by an AHNEP with the same number of nodes. We also show that the families of GHNEPs and AHNEPs with 2 nodes are not computationally complete. Although the sharpness of the upper bounds is not verified, we considerably improved the previous results. The gap between universality and non-universality for GHNEPs now is very small (it is the same as for the famous PCP problem). In [10] we completed investigation of HNEPs with one node and presented a precise description of languages generated by them.

We considered new variant of HNEP, so called Obligatory Network of Evolutionary Processors (OHNEP shortly) [12]. The differences between HNEP and OHNEP are:

1) in using deletion and substitution operations: a node discards a string if no operations in node are applicable to string (in HNEP

case this string remains in the node),

2) an underlying graph is directed graph (in HNEP case this graph is undirected).

We underline that both differences are natural.

The first one allows us to have the uniform definitions of the operations on a string, as opposed to considering two cases as in HNEPs (it is the set of results of the applications of the operation to all possible positions; the case when there are no such positions yields the empty set by definition).

The second difference, that of generalization of the underlying graph to be directed, is natural from the computational point of view; moreover, since the loops are typically not considered, it also seems relevant from the viewpoint of the biological motivation that the communicating channels are directed.

These differences allow proofing universality of OHNEP with nodes with only one operation, without input and output filters and using only insertion operation at the left end and deletion operation at the right end of a string. This interesting fact stresses the importance of structure of HNEP in order to reach universality.

On the other hand we can avoid substitution operation. Notice that this feature of OHNEP to discard a string if this string does not participate at the operations has counterpart in DNA computing area, TVDH systems also discard strings if they do not participate at splicing operations ([73]). A task to find a minimal number of nodes of universal OHNEP is open. A variant of OHNEP with underlying complete graph is not considered yet.

An implementation of HNEPs and OHNEPs in mathematical linguistics is also interesting task to investigate. The constructions demonstrate that distributed architectures of very small size, with uniform structure and with components based on very simple language theoretic operations are sufficient both to generate and to recognize any recursively enumerable language.

3.9 Other Models of Natural Computing

A number-conserving cellular automaton (NCCA) is a cellular automaton whose states are integers and whose transition function keeps the sum of all cells constant throughout its evolution. It can be seen as a kind of modeling of the physical conservation laws of mass or energy.

We showed a construction method of NCCAs with radius 1/2 [61]. The local transition function is expressed via a single unary function which can be regarded as "flows" of numbers. In spite of the strong constraint, we constructed NCCAs with radius 1/2 that simulate any cellular automata with radius 1/2 or any NCCA with radius 1. We also consider the state complexity of these non-splitting simulations ($4n^2 + 2n + 1$ and $8n^2 + 12n - 16$, respectively). These results also imply existence of intrinsically universal NCCA with radius 1/2.

A reversible logic element is a primitive from which reversible computing systems can be constructed. A rotary element is a typical 2-state 4-symbol reversible element with logical universality, and we can construct reversible Turing machines from it very simply.

There are also many other reversible elements with 1-bit memory. So far, it is known that all the 14 kinds of non-degenerate 2-state 3-symbol reversible elements can simulate a Fredkin gate, and hence they are universal. We showed that all these 14 elements can "directly" simulate a rotary element in a simple and systematic way [84, 85, 86].

Acknowledgments All authors acknowledge the support by the Science and Technology Center in Ukraine, project 4032.

References

[1] L.M.Adleman: *Molecular computation of solutions of combinatorial problems*, Science, 226, pp. 1021–1024, 1994.

[2] A.Alhazov: *Maximally Parallel Multiset-Rewriting Systems: Browsing the Configurations.* In: M.A. Gutiérrez-Naranjo, A. Riscos-Núñez, F.J. Romero-Campero, D. Sburlan: RGNC re-

port 01/2005, University of Seville, *Third Brainstorming Week on Membrane Computing*, Fénix Editora, Sevilla, 2005, pp. 1–10.

[3] A.Alhazov, C.Martín-Vide, Yu.Rogozhin: *On the number of nodes in universal networks of evolutionary processors*, Acta Inf. 43 (2006) pp. 331–339.

[4] A.Alhazov: *Ciliate Operations without Context in a Membrane Computing Framework.* Romanian Journal of Information Science and Technology, vol.10, no.4 (2007), pp. 315–322.

[5] A.Alhazov, L.Burtseva, S.Cojocaru, Y.Rogozhin: *Computing solutions of #P-complete problems by P systems with active membranes.* In: Proceedings of Ninth Workshop on Membrane Computing (WMC9), Edinburgh, UK, July 28 - 31, 2008, pp. 59–70.

[6] A.Alhazov, E.Csuhaj-Varjú, C.Martín-Vide, Yu.Rogozhin: *About Universal Hybrid Networks of Evolutionary Processors of Small size.* In: Pre-proceedings of the 2nd International Conference on Language and Automata, Theory and Applications, LATA 2008, March 13 - 19, 2008, Rovira i Virgili University, Tarragona, Spain. Technical Report of GRLMC No. 36/08, Universitat Rovira i Virgili, Tarragona, Spain (2008), pp. 43–54.

[7] A.Alhazov, E.Csuhaj-Varjú, C.Martín-Vide, Y.Rogozhin: *About Universal Hybrid Networks of Evolutionary Processors of Small Size.* Lecture Notes in Computer Science, Springer, 5196, pp. 28–39, 2008.

[8] A.Alhazov, E.Csuhaj-Varjú, C.Martín-Vide, Y.Rogozhin: *Computational Completeness of Hybrid Networks of Evolutionary Processors with Seven Nodes.* In: Proceedings of the Workshop DCFS 2008, Descriptional Complexity of Formal Systems, Charlottetown, Prince Edward Island, Canada, July 16-18, 2008.

[9] A.Alhazov, M.Margenstern, S.Verlan: *Fast synchronization in P systems. In: Proceedings of Ninth Workshop on Membrane Computing (WMC9)*, Edinburgh, UK, July 28 - 31, 2008, pp. 59–70.

[10] A.Alhazov, Yu.Rogozhin: *About Precise Characterization of Languages Generated by Hybrid Networks of Evolutionary Processors with One Node.* Computer Science Journal of Moldova, 16, no.3 (48) (2008), pp. 364–376.

[11] A.Alhazov, S.Verlan: *Minimization Strategies for Maximally Parallel Multiset Rewriting Systems.* Technical Report of Turku Centre for Computer Science, Turku, Finland, no. 862, (2008).

[12] A.Alhazov, G. Bel-Enguix, Yu.Rogozhin: *Obligatory Hybrid Networks of Evolutionary Processors.* In: Proc. of the First International Conference on Agents and Artificial Intelligents, ICAART 2009, Porto, Portugal, 19-21 January, 2009, pp. 613–618.

[13] A.Alhazov, E.Boian, S.Cojocaru, Yu.Rogozhin: *Modelling Inflections in Romanian Language by P Systems with String Replication.* In: Proc. of the 10th Workshop on Membrane Computing, WMC10, Curtea de Arges (Romania), August 24 - 27, 2009, pp. 116–128.

[14] A.Alhazov, L.Burtseva, S.Cojocaru, Yu.Rogozhin: *Solving PP-Complete and #P-Complete Problems by P Systems with Active Membrane.* Lecture Notes in Computer Science, Springer, 5391 (2009), pp. 108–117.

[15] A.Alhazov, E.Boian, S.Cojocaru, Yu.Rogozhin: *Modelling Inflections in Romanian Language by P Systems with String Replication.* Computer Science Journal of Moldova, vol.17, no.2(50), 2009, pp. 160–178.

[16] A.Alhazov, S.Cojocaru, L.Malahova, Yu.Rogozhin: *Dictionary Search and Update by P Systems with String-Objects and Active Membranes.* International Journal of Computers, Communications and Control, Vol. IV, No. 3 (2009), pp. 206–213.

[17] A.Alhazov, S.Cojocaru, L.Malahova, Yu.Rogozhin: *Dictionary Search and Update by P Systems with String-Objects and Active Membranes.* In: Proc. of the 7th Brainstorming Week on Membrane Computing, Sevilla, Spain, February 2-February 6, 2009. Universidad de Sevilla, RGNC REPORT 1/2009, pp. 1–8.

[18] A.Alhazov, E.Csuhaj-Varjú, C.Martín-Vide, Yu.Rogozhin: *On the size computationally complete hybrid networks of evolutionary processors.* Theoretical Computer Science, Elsevier, 410 (2009), pp. 3188–3197.

[19] A.Alhazov, J.Dassow, C.Martín-Vide, Yu.Rogozhin, B.Truthe: *On Networks of Evolutionary Processors with Nodes of Two Types.* Fundamenta Informaticae, IOS Press, 91(1) (2009), pp. 1–15.

[20] A.Alhazov, R.Freund, M.Oswald, S.Verlan: *Partial Halting and Minimal Parallelism Based on Arbitrary Rule Partitions.* Fundamenta Informaticae, IOS Press, 91(1) (2009), pp. 17–34.

[21] A.Alhazov, A.Krassovitskiy, Yu.Rogozhin, S.Verlan: *P Systems with Minimal Insertion and Deletion.* In: Proc. of the 7th Brainstorming Week on Membrane Computing, Sevilla, Spain, February 2 - February 6, 2009. Universidad de Sivilla, RGNC REPORT 1/2009, pp. 9–21.

[22] A.Alhazov, A.Krassovitskiy, Yu.Rogozhin, S.Verlan: *A Note on P Systems with Small-Size Insertion and Deletion.* In: Proc. of the 10th Workshop on Membrane Computing, WMC10, Curtea de Arges (Romania), August 24 - 27, 2009, pp. 534–537.

[23] A.Alhazov, M.Margenstern, S.Verlan: *Fast Synchronization in P Systems.* In: D.W. Corne, P. Frisco, Gh. P?un, G. Rozenberg, A. Salomaa: Membrane Computing - 9th International Workshop, WMC 2008, Edinburgh, Revised Selected and Invited Papers, Lecture Notes in Computer Science vol. 5391, Springer, 2009, pp. 118–128.

[24] A.Alhazov, K.Morita: *A Short Note on Reversibility in P Systems.* In: Proc. of the 7th Brainstorming Week on Membrane Computing, Sevilla, Spain, February 2-February 6, 2009. Universidad de Sevilla, RGNC REPORT 1/2009, pp. 23–28.

[25] A.Alhazov, K.Morita: *On Reversibility and Determinism in P systems.* In: Proc. of the 10th Workshop on Membrane Comput-

ing, WMC10, Curtea de Arges (Romania), August 24 - 27, 2009, pp. 129–139.

[26] A.Alhazov, I.Petre, V.Rogojin: *The parallel complexity of signed graphs: decidability results and an improved algorithm.* Theoretical Computer Science, 410, Elsevier, (2009, pp. 2308–2315.

[27] A. Alhazov, C.Ciubotaru, Yu.Rogozhin, S.Ivanov: *The Family of Languages Generated by Non-Cooperative Membrane Systems.* In: M. Gheorghe, Th. Hinze, Gh. Păun: *Preproceedings of the Eleventh Conference on Membrane Computing, CMC11*, Jena, Verlag ProBusiness Berlin, 2010, pp. 37–51, and *Lecture Notes in Computer Science* 6501, to appear.

[28] A.Alhazov, C.Ciubotaru, Yu.Rogozhin, S.Ivanov: *The Membrane Systems Language Class.* LA symposium, RIMS Kôkyûroku Series vol. 1691, Kyoto University, 2010, pp. 44–50.

[29] A.Alhazov, C.Ciubotaru, Yu.Rogozhin, S.Ivanov: *Introduction to the Membrane Systems Language Class.* In: Proceedings of the 3rd International Conference "Telecommunications, Electronics and Informatics", ICTEI 2010, vol. II, Chişinău, 2010, pp. 19–24.

[30] A.Alhazov, C.Ciubotaru, Yu.Rogozhin, S.Ivanov: *The Membrane Systems Language Class.* In: Proceedings of the Eighth Brainstorming Week on Membrane Computing, Sevilla, Spain, February 1-5, 2010, RGNC Report 1/2010, Fenix Editora, Sevilla, 2010, pp. 23–36.

[31] A.Alhazov, C.Ciubotaru, Y.Rogozhin, S.Ivanov: *The family of Languages Generated by Non-Cooperative Membrane Systems.* In: Proceedings of the Eleventh International Conference on Membrane Computing. Friedrich Schiller University Jena, Germany, 24-27 August, 2010, pp. 37–52.

[32] A.Alhazov, R.Freund, K.Morita: *Reversibility and Determinism in Sequential Multiset Rewriting.* Unconventional Computation 2010, Tokyo, Lecture Notes in Computer Science, Springer, vol. 6079, 2010, pp. 21–31.

[33] A.Alhazov, S.Ivanov, Y.Rogozhin: *Polymorphic P Systems.* In: Proceedings of the Eleventh International Conference on Membrane Computing. Friedrich Schiller University Jena, Germany, 24-27 August, 2010, pp. 53–66.

[34] A.Alhazov, M.Kogler, M.Margenstern, Yu.Rogozhin, S.Verlan: *Small Universal TVDH and Test Tube Systems.* International Journal of Foundations of Computer Science, 2010 (in press).

[35] A.Alhazov, A.Krassovitskiy, Yu.Rogozhin, S.Verlan: *P systems with minimal insertion and deletion.* Theoretical Computer Science, Elsevier, 2010 (in press).

[36] A.Alhazov, A.Krassovitskiy, Yu.Rogozhin, S.Verlan: *Small Size Insertion and Deletion Systems.* In: Mathematics, Computing, Language, and Life: Frontiers in Mathematical Linguistics and Language Theory, Vol.2. Scientific Applications of Language Methods. World Scientific, 2010 (in press).

[37] A.Alhazov, K. Morita: On Reversibility and Determinism in P Systems. Workshop on Membrane Computing, WMC10, Curtea de Arges, 2009, Lecture Notes in Computer Science, Springer 5957, 2010, pp. 158–168.

[38] A.Alhazov, Y.Rogozhin, S.Verlan: *A Small Universal P Systems.* In: Proceedings of the Eleventh International Conference on Membrane Computing. Friedrich Schiller University Jena, Germany, 24-27 August, 2010, pp. 67–74.

[39] A.Alhazov, S.Verlan: *Minimization Strategies for Maximally Parallel Multiset Rewriting Systems.* arXiv:1009.2706v1 [cs.FL] 14 Sep 2010, http://arxiv.org/abs/1009.2706.

[40] C.H. Bennett, *Logical Reversibility of Computation*, IBM J. Res. Develop. 6, pp. 525–532, 1973.

[41] E.Boian, C.Ciubotaru, S.Cojocaru, A.Colesnicov, V.Demidova, L.Malahov: *P-systems application for solution of some problems in computational linguistics.* Proceedings of the International

Conference ICT+ ”Information and Communication Technologies - 2009”. May 18-21, 2009, Chisinau, Republic of Moldova. pp. 30–33, ISBN 978-9975-66-134-8. (in Romanian)

[42] E.Boian, S.Cojocaru, A.Colesnicov, C.Ciubotaru, L.Malahova: *Using the P systems in computer linguistic applications (in Russian).* In: Proceedings of the International Conference ”Horizons of Applied Linguistics and Linguistics Technologies”, Kiev, Ukraine, 21 - 26 September, 2009. p.53.

[43] E.Boian, S.Cojocaru, V.Macari, G.Magariu, T.Verlan: On simulation of inflection process in Romanian language by P-systems with string replication. In CD: Proceedings of the ECIT2010 - 6th European Conference on Intelligent Systems and Technologies. Iasi, Romania, October 07-09, 2010.

[44] J.Castellanos, C.Martń-Vide, V.Mitrana, J.Sempere: *Solving NP-complete problems with networks of evolutionary processors,* In: Proc. IWANN, Lecture Notes in Computer Science 2084, Springer-Verlag, Berlin, 2001, pp. 621–628.

[45] J.Cocke, M.Minsky: *Universality of tag systems with P=2,* Journal of the ACM, 11(1), 1964, pp. 15–20.

[46] S.Cojocaru, E.Boian: *Determination of inflexional group using P systems.* Computer Science Journal of Moldova, vol.18, no.1(52), 2010, pp. 70–81.

[47] E.Csuhaj-Varjù, L.Kari, G.Păun, *Test Tube distributed system based on splicing*, Computer and AI, 2–3, pp. 211–232, 1996.

[48] E.Csuhaj-Varjú, A.Salomaa: *Networks of parallel language processors,* In: New Trends in formal Language Theory (Gh. Păun, A. Salomaa, Eds.), Lecture Notes in Computer Science 1218, Springer-Verlag, Berlin, 1997, pp. 299–318.

[49] E.Csuhaj-Varjú, C.Martín-Vide, V.Mitrana: *Hybrid networks of evolutionary processors are computationally complete,* Acta Inf. 41 (2005), pp. 257–272.

[50] E.Csuhaj-Varjú, S.Verlan: *On length-separating test tube systems.* Natural computing, Springer, vol.7, no.2, 2008, pp. 167–181.

[51] E.Csuhaj-Varjú, S.Verlan: *Power and Size of Generalized Communicating P Systems with Minimal Interaction Rules.* In: Proc. of the 10th Workshop on Membrane Computing, WMC10, Curtea de Arges (Romania), August 24 - 27, 2009, pp. 547–551.

[52] E.Csuhaj-Varjú, G.Vaszil, S.Verlan: *On Generalized Communicating P Systems with One Symbol.* In: Proceedings of the Eleventh International Conference on Membrane Computing. Friedrich Schiller University Jena, Germany, 24-27 August, 2010, pp. 137–154.

[53] S.E.Fahlmann, G.E.Hinton, T.J.Seijnowski: *Massively parallel architectures for AI: NETL, THISTLE and Boltzmann machines,* In: Proc. AAAI National Conf. on AI, William Kaufman, Los Altos, 1983, pp. 109–113.

[54] C.Ferretti, G.Mauri, C.Zandron: *Nine Test Tubes Generate any RE Language*, personal communication.

[55] R.Freund, S.Verlan: *(Tissue) P systems working in the k-restricted minimally parallel derivation mode.* In: Proceedings of International Workshop on Computing with Biomolecules, August 27th, 2008, Wien, Austria, pp. 43–52.

[56] R.Freund, A.Alhazov, Y.Rogozhin, S.Verlan: *Communication P systems.* In: The Oxford Handbook of Membrane Computing, ed. by Gh.Păun, G.Rozenberg, A.Salomaa, Oxford University Press, 2010.

[57] R.Freund, M.Kogler, Yu.Rogozhin, S.Verlan: *Graph-Controlled Insertion-Deletion Systems.* In: Proceedings DCFS 2010. EPTCS 31, 2010, pp. 88–98, doi:10.4204/EPTCS.31.11.

[58] F.Guarnieri, M.Fliss, C.Bacroft: *Making DNA add.* Science (1996) 273(12):220–223.

[59] T.Head *Formal Language Theory and DNA: An Analysis of the Generative Capacity of Specific Recombinant Behaviors*, Bulletin of Mathematical Biology, Vol. 49, No. 6, pp. 737–759, 1987.

[60] W.D.Hillis: *The Connection Machine.* MIT Press, Cambridge, 1985.

[61] K.Imai, A.Alhazov: *On Universality of Radius 1/2 Number-Conserving Cellular Automata.* Unconventional Computation 2010, Tokyo, Lecture Notes in Computer Science, Springer, vol. 6079, 2010, pp. 45–55.

[62] S.Ivanov, V.Macari: *CUDA in Simulating P Systems.* In: Proceedings of the 3rd International Conference "Telecommunications, Electronics and Informatics" ICTEI 2010, Volume II, pp. 198–201.

[63] Y.Khodor, J.Khodor, T.F.Jr.Knight: *Experimental confirmation of the basic principles of length-only discrimination.* In: Jonoska N, Seeman NC (eds) DNA7. Lecture notes in computer science, vol 2340. Springer-Verlag, Berlin (2002), pp. 223–230.

[64] A.Krassovitskiy, Y.Rogozhin, S.Verlan: *One-sided Insertion and Deletion: Traditional and P Systems Case.* In: Proceedings of International Workshop on Computing with Biomolecules, August 27th, 2008, Wien, Austria, pp. 53–64.

[65] A.Krassovitskiy, Y.Rogozhin, S.Verlan: *Further results on insertion-deletion systems with one-sided contexts.* In: Pre-proceedings of the 2nd International Conference on Language and Automata, Theory and Applications, LATA 2008, March 13 - 19, 2008, Rovira i Virgili University, Tarragona, Spain. Technical Report of GRLMC No. 36/08, Universitat Rovira i Virgili, Tarragona, Spain (2008), pp. 347–358.

[66] A.Krassovitskiy, Y.Rogozhin, S.Verlan: *Further results on insertion-deletion systems with one-sided contexts.* Lecture Notes in Computer Science, Springer, 5196, pp. 333–344, 2008.

[67] A.Krassovitskiy, Y.Rogozhin, S.Verlan: *Computational Power of P Systems with Small Size Insertion and Deletion Rules.* In: Proc. of the International Workshop on The Complexity of Simple Programs, University College Park, Ireland, December 6th and 7th, 2008, pp. 137–148.

[68] A.Krassovitskiy, Yu.Rogozhin, S.Verlan: *Computational Power of P Systems with Small size Insertion and Deletion Rules.* EPTCS 1, 2009, pp. 108–117, doi:10.4204/EPTCS.1.10.

[69] A.Krassovitskiy, Yu.Rogozhin, S.Verlan: *Computational power of insertion-deletion (P) systems with rules of size two.* Natural Computing, Springer, DOI 10.1007/s11047-010-9208-y, 2010 (in press).

[70] M.Margenstern, Yu.Rogozhin: *A Universal Time-Varying Distributed H-System of Degree 2,* Preliminary Proceedings of Forth International Meeting on DNA Based Computers, June 15-19, 1998, University of Pennsylvania, U.S.A., 1998.

[71] M.Margenstern, Yu.Rogozhin: *A Universal Time-Varying Distributed H System of Degree 2*, Biosystems, 52, 1999, pp. 73–80.

[72] M.Margenstern, Yu.Rogozhin: *Generating All Recursively Enumerable Languages with a Time-Varying Distributed H System of Degree 2*, Technical report, Institut Universitaire de Technologie de Metz, 1999, Publications du G.I.F.M.

[73] M.Margenstern, Yu.Rogozhin: *About time-varying distributed H systems,* DNA Computing: 6th International Workshop on DNA-Based Computers, DNA 2000, Leiden, The Netherlands, June 13-17, 2000, LNCS, Springer, Revised Papers (A. Condon, G. Rozenberg, Eds.), 2054 (2000), pp. 53–62.

[74] M.Margenstern, Yu.Rogozhin: *Time-Varying Distributed H systems of Degree 2 Generate All Recursively Enumerable Languages*, in: Where Do Mathematics, Computer Science and Biology Meet (C. Martin-Vide, V. Mitrana, Eds.), Kluwer Academic, Dortrecht, 2000, pp. 399–407.

[75] M.Margenstern, Yu.Rogozhin: *Extended Time-Varying Distributed H Systems - Universality Result*, Proceedings of The 5th World Multi-Conference on Systemics, Cybernetics and Informatics, Industrial Systems, SCI 2001, Orlando, Florida USA, July 22-25, 2001, IX, 2001.

[76] M.Margenstern, Yu.Rogozhin: *Time-Varying Distributed H Systems of Degree 1 Generate All Recursively Enumerable Languages*, In: Words, Semigroups, and Transductions (M. Ito, G. Păaun, S. Yu, Eds.), World Scientific, Singapore, 2001, ISBN 981-02-4739-7, pp. 329–340, Festschrift in Honor of Gabriel Thierrin.

[77] M.Margenstern, Yu.Rogozhin: *A Universal Time-Varying Distributed H System of Degree 1.,* DNA Computing: 7th International Workshop on DNA-Based Computers, DNA7, Tampa, FL, USA, June 10-13, 2001. Revised Papers (N. Jonoska, N. C. Seeman, Eds.), 2340, Springer Verlag, Berlin, Heidelberg, New York, 2002.

[78] M.Margenstern, Yu.Rogozhin, S.Verlan: *Time-Varying Distributed H Systems of Degree 2 Can Carry Out Parallel Computations,* DNA Computing: 8th International Workshop on DNA-Based Computers, DNA8, Sapporo, Japan, June 10-13, 2002. Revised Papers (M. Hagiya, A. Ohuchi, Eds.), 2568 (2002), pp. 326–336.

[79] M.Margenstern, Y.Rogozhin, S.Verlan: *Time-varying distributed H systems with parallel computations: the problem is solved*, DNA Computing: 9th International Workshop on DNA Based Computers, DNA9, Madison, WI, USA, June 1-3, 2003. Revised Papers (J. Chen, J. Reif, Eds.), 2943, Springer, 2004.

[80] M.Margenstern, Gh.Păun, Yu.Rogozhin, S.Verlan: *Context-free insertion-deletion systems.* Theoretical Computer Science, Elsevier, vol.330, issue 2 (2005), pp. 339–348.

[81] A.Matveevici, Yu.Rogozhin, S.Verlan: *Insertion-Deletion Systems with One-Sided Contexts.* Lecture Notes in Computer Science, Springer, vol. 4664 (2007), pp. 205–217.

[82] M.Minsky: *Computations: Finite and Infinite Machines,* Prentice Hall, Englewood Cliffts, NJ, 1967.

[83] V.Mitrana, I.Petre, V.Rogojin: *Accepting splicing systems.* Theoretical Computer Science, Elsevier, 411, 25, 2010, pp. 2414–2422.

[84] K.Morita, Ts.Ogiro, A.Alhazov, Ts.Tanizawa: *Non-degenerate 2-State Reversible Logic Elements with Three or More Symbols Are All Universal.* In: Proceedings of 2nd Workshop on Reversible Computation, July 2nd - 3rd, 2010, Bremen, Germany, pp. 27–34.

[85] Ts. Ogiro, A. Alhazov, Ts. Tanizawa, K. Morita: *Universality of 2-State 3-Symbol Reversible Logic Elements - A Direct Simulation Method of a Rotary Element.* In: Proceedings of the 4th International Workshop on Natural Computing, Himeji, 2009, pp. 220–227.

[86] Ts. Ogiro, A. Alhazov, Ts. Tanizawa, K. Morita: *Universality of 2-State 3-Symbol Reversible Logic Elements - A Direct Simulation Method of a Rotary Element.* Natural Computing, PICT 2, Springer Japan, part 3, 2010, pp. 252–259.

[87] *The Oxford Handbook of Membrane Computing,* ed. by Gh.Păun, G.Rozenberg, A.Salomaa. Oxford University Press, 2010.

[88] A.Păun: *On Time-Varying H Systems,* Bulletin of EATCS, 67, February 1999, pp. 157–164.

[89] Gh.Păun, G.Rozenberg, A.Salomaa: *Computing by splicing*, TCS 168, pp. 321–336, 1996.

[90] Gh.Păun: *DNA computing: distributed splicing systems, Structures in Logic and Computer Science.* A Selection of Essays in Honor of A. Ehrenfeucht (J. Mycielsky, G. Rozenberg, A. Salomaa, Eds.), 1261, Springer Verlag, Berlin, Heidelberg, New York, 1997.

[91] Gh.Păun, G.Rozenberg, A.Salomaa: *DNA Computing: New Computing Paradigms,* Springer Verlag, Berlin, Heidelberg, New York, September 1998, ISBN 3-540-64196-3.

[92] Gh.Păun: *DNA computing based on splicing: universality results,* Proceedings of the Second Internetional Colloquium on Universal Machines and Computations, Metz, France (M. Margenstern, Ed.), I, IUT de Metz, 1998.

[93] Gh.Păun, G.Rozenberg, A.Salomaa, Eds.:*Handbook of Membrane Computing.* Oxford University Press, 2010.

[94] Gh.Păun: Membrane Computing. An Introduction, Springer, 2002.

[95] $http : //www.p - lingua.org/wiki/index.php/Main_Page$

[96] L.Priese, Yu.Rogozhin, M.Margenstern: *Finite H-systems with 3 Test Tubes are not Predictable.* In: Proceedings of Pacific Simposium on Biocomputing, 3, Kapalua, Maui, January 1998, Hawaii, USA (R.Altman, A.Dunker, L.Hanter, T.Klein eds.), World Sci.Publ., Singapure (1998), pp. 545–556.

[97] Yu.Rogozhin, S.Verlan: *On the Rule Complexity of Universal Tissue P Systems.* LNCS, vol. 3850, pp. 356–362, Springer (2006).

[98] Yu.Rogozhin, S.Verlan: *New choice for small universal devices: Symport/Antiport P systems.* EPTCS 1, 2009, pp. 235-242, doi:10.4204/EPTCS.1.23.

[99] G.Rozenberg, A.Salomaa, Eds.:*Handbook of Formal Languages*, vol. 1–3, Springer, 1997.

[100] *P systems webpage.* `http://ppage.psystems.eu/`

[101] *TVDHsim: Time-Varying Distributed H Systems Simulator.* http://lita.sciences.univ-metz.fr/ verlan/

[102] S.Verlan: *Calculs Moleculaires: les Syst'emes Distributes 'a Changement de Phase,* Master Thesis, Universite de Metz, 2001.

[103] S.Verlan: *On Enhanced Time-Varying Distributed H Systems,* Computer Science Journal of Moldova, 10(3), 2002, pp. 263–279, Kishinev.

[104] S.Verlan: *A Frontier Result on Enhanced Time-Varying Distributed H Systems with Parallel Computations,* Preproceedings of DCFS'03, Descriptional Complexity of Formal Systems, Budapest, Hungary, July 12-14, 2003, 2003.

[105] S.Verlan: *Communicating Distributed H Systems with Alternating Filters,* in: Aspects of Molecular Computing. Essays Dedicated to Tom Head on the Occasion of His 70th Birthday (N. Jonoska, Gh. Păun, G. Rozenberg, Eds.), vol. 2950 of LNCS, Springer Verlag, Berlin, Heidelberg, New York, 2004, pp. 367–384.

[106] S.Verlan: *Head Systems and Applications to Bio-Informatics,* Ph.D. Thesis, University of Metz, 2004.

[107] S. Verlan: *On minimal context-free insertion-deletion systems.* In C. Mereghetti, B. Palano, G. Pighizzini, and D. Wotschke, editors, *Seventh International Workshop on Descriptional Complexity of Formal Systems, June 30 - July 2, 2005 Como, Italy. Proceedings.*, 285-292, 2005. Technical repport no. 06-05, University of Milan. In publication in *Journal of Automata Languages and Combinatorics.*

[108] S.Verlan: *Look-Ahead Evolution for P Systems.* In: Proc. of the 10th Workshop on Membrane Computing, WMC10, Curtea de Arges (Romania), August 24 - 27, 2009, pp. 507–513.

[109] S.Verlan, P.Frisco: *Splicing P Systems.* In: The Oxford Handbook of Membrane Computing, ed. by Gh.Păun, G.Rozenberg, A.Salomaa, Oxford University Press, 2010.

[110] S.Verlan, Y.Rogozhin: *New choice for small universal devices: Symport/antiport P systems.* In: International Workshop on The Complexity of Simple Programs, University College Park, Ireland, December 6th and 7th, 2008, pp. 305–314.

A. Alhazov[1,2], E. Boian[1], L. Burtseva[1],
C. Ciubotaru[1], S. Cojocaru[1], A. Colesnicov[1],
V. Demidova[1], S. Ivanov[1,3], V. Macari[1],
G. Magariu[1], L. Malahova[1], V. Rogojin[1,4],
Yu. Rogozhin[1], T. Tofan[1], S. Verlan[1,5], T. Verlan[1]

Received November 1, 2010

[1] Institute of Mathematics and Computer Science
Academy of Sciences of Moldova
5 Academiei str., Chişinău, MD-2028, Moldova

[2] FCS, Department of Information Engineering
Graduate School of Engineering, Hiroshima University
Higashi-Hiroshima 739-8527 Japan

[3] Technical University of Moldova, Faculty of Computers,
Informatics and Microelectronics,
Ştefan cel Mare 168, Chişinău MD-2004 Moldova

[4] Biomedicum Helsinki,
B524a P.O.Box 63 (Haartmaninkatu 8) 00014
UNIVERSITY OF HELSINKI

[5] LACL, Departement Informatique
UFR Sciences et Technologie
Universite Paris XII
61, av. General de Gaulle
94010 Creteil, France

E–mails:
Dr. Artiom Alhazov: `artiom@math.md`,
Dr. Elena Boian: `lena@math.md`,
Dr. Liudmila Burtseva: `burtseva@math.md`,
Dr. Constantin Ciubotaru: `chebotar@math.md`,
Dr.hab. Svetlana Cojocaru: `Svetlana.Cojocaru@math.md`,
Dr. Alexandru Colesnicov: `kae@math.md`,
Valentina Demidova: `demidova@math.md`,
Sergiu Ivanov: `sivanov@math.md`,
Veaceslav Macari: `vmacari@yandex.ru`,
Dr. Galina Magariu: `gmagariu@math.md`,
Ludmila Malahova: `mal@math.md`,
Dr. Vladimir Rogojin: `vladimir.rogojin@helsinki.fi`,
Dr.hab. Yurii Rogozhin: `rogozhin@math.md`,
Tatiana Tofan: `ttofan@math.md`,
Dr.hab. Sergey Verlan: `verlan@univ-paris12.fr`,
Tatiana Verlan: `tverlan@math.md`

Computer Science Journal of Moldova, vol.**18**, no.2(53), 2010

Membrane Systems Languages Are Polynomial-Time Parsable

Artiom Alhazov Constantin Ciubotaru Sergiu Ivanov
Yurii Rogozhin

Abstract

The focus of this paper is the family of languages generated by transitional non-cooperative P systems without further ingredients. This family can also be defined by so-called time yields of derivation trees of context-free grammars. In this paper we prove that such languages can be parsed in polynomial time, where the degree of polynomial may depend on the number of rules and on the size of the alphabet.

1 Introduction

Membrane computing is a theoretical framework of parallel distributed multiset processing. It has been introduced by Gheorghe Păun in 1998, and has been an active research area; see [6] for the comprehensive bibliography and [4],[3] for a systematic survey. Membrane systems are also called P systems.

The configurations of membrane systems (with symbol objects) consist of multisets over a finite alphabet, distributed across a tree structure. Therefore, even such a relatively simple structure as a word (i.e., a sequence of symbols) is not explicitly present in the system. To speak of languages as sets of words, one first needs to represent them in membrane systems, and there are a few ways to do it.

One of the most elegant ways is to do all the processing by multisets, and regard the order of sending the objects in the environment as their order in the output word. In case of ejecting multiple symbols in the same step, the output word is formed from any of their permutations. One can say that this approach also needs an implicit observer, but at least this observer only inspects the environment and it is, in some sense, the simplest possible one.

Following [2], in this paper we are interested in the case when the rules are non-cooperative, i.e., all objects evolve independently. A number of results have been established in [2]. For instance, it was shown that one membrane is enough, and a characterization of this family was given via derivation trees of context-free grammars. Next, three normal forms were given for the corresponding grammars. It was than shown that the membrane systems language family lies between regular and context-sensitive families of languages, and it is incomparable with linear and with context-free languages. Then, the lower bound was strengthened to $REG \bullet \mathtt{Perm}(REG)$. An example of a considerably more "difficult" language was given than the lower bound mentioned above. The membrane systems language family was also shown to be closed under union, permutations, erasing/renaming morphisms. It is not closed under intersection, intersection with regular languages, complement, concatenation or taking the mirror image.

In attempt to lower the known upper bound (semilinear context-sensitive) of these languages, we show here that the word membership problem can be solved in polynomial time.

2 Definitions

Consider a finite set V. The set of all words over V is denoted by V^*, the concatenation operation is denoted by $\bullet$ (which is written only when necessary) and the empty word is denoted by λ. Any set $L \subseteq V^*$ is called a language. For a word $w \in V^*$ and a symbol $a \in V$, the number of occurrences of a in w is written as $|w|_a$. We write $w[i]$ to denote the i-th symbol of w, $1 \leq i \leq |w|$. The permutations of a word $w \in V^*$ are $\mathtt{Perm}(w) = \{x \in V^* \mid |x|_a = |w|_a \forall a \in V\}$. We denote the set of all permutations of the words in L by $\mathtt{Perm}(L)$, and we extend this notation to families of languages. We use FIN, REG, LIN, CF, MAT, CS, RE to denote finite, regular, linear, context-free, matrix without appearance checking and with erasing rules, context-sensitive and recursively enumerable families of languages, respectively. The family of languages generated by extended (tabled) interactionless L systems is denoted by $E(T)0L$. Notation $SLIN$ stands for the semilinear languages. We denote by **P** the family of languages recognizable by Turing machines in polynomial time. For more formal language preliminaries, we refer the reader to [5].

A multiset over V is a mapping $M : V \to \mathbb{N}$; $M(a)$ is multiplicity of a in M. For $V = \{a_1, \cdots, a_m\}$, we may write M as $\left\{a_1^{M(a_1)}, \cdots, a_m^{M(a_m)}\right\}$, omitting missing elements. The size $|M|$ of a multiset is $\sum_{i=1}^{m} M(a_i)$. We use the extension of the set notations to multisets; for instance, $M_1 \subseteq M_2$, $M_1 \cup M_2$ and $M_1 \setminus M_2$ mean $M_1(a) \leq M_2(a)$, $M_1(a) + M_2(a)$ and $\max(M_1(a) - M_2(a), 0)$ for the multiplicities of all symbols a, respectively. Multisets in membrane computing are typically represented by strings; in this paper we use the set notations described above, to be able to distinguish between multisets and strings.

2.1 Transitional P systems

A membrane system is defined by a construct

$$\begin{array}{rcl} \Pi & = & (O, \mu, w_1, \cdots, w_m, R_1, \cdots, R_m, i_0), \text{ where} \\ O & & \text{is a finite set of objects,} \\ \mu & & \text{is a hierarchical structure of membranes,} \\ w_i & & \text{is the initial multiset in region } i,\ 1 \leq i \leq m, \\ R_i & & \text{is the set of rules of region } i,\ 1 \leq i \leq m, \\ i_0 & & \text{is the output region.} \end{array}$$

The membranes are bijectively labeled by $1, \cdots, m$, the interior of each membrane defines a region; the environment is referred to as region 0. When languages are considered, $i_0 = 0$ is assumed.

The rules of a membrane system have the form $u \to v$, where u is a non-empty multiset over O and v is a multiset over $(O \times Tar)$. The target indications from $Tar = \{here, out\} \cup \{in_j \mid 1 \leq j \leq m\}$ are written as a subscript, and target *here* is typically omitted. In case of non-cooperative rules, u is a multiset of size 1.

The rules are applied in maximally parallel way: no further rule should be applicable to the idle objects. In case of non-cooperative systems, the concept of maximal parallelism is the same as in L systems: all objects evolve by the associated rules in the corresponding regions (except objects a in regions i such that R_i does not contain any rule $a \to u$, but these objects do not contribute to the result). The choice of rules is non-deterministic.

A configuration of a P system is a construct which contains the information about the hierarchical structure of membranes as well as the contents of every membrane at a definite moment of time. The process of applying all rules which are applicable in the current configuration and thus obtaining a new configuration is called a transition. A sequence of transitions is called a computation. The computation halts when such a configuration is reached that no rules are applicable. The result of a (halting) computation is the *sequence* of objects sent to the environment (all the permutations of the symbols sent out in the same time are considered). The language $L(\Pi)$ generated by a P system Π is the union of the results of all computations. The family of languages generated by non-cooperative transitional P systems with at most m membranes is denoted by $LOP_m(ncoo, tar)$. If the number of membranes is not bounded, m is replaced by $*$ or omitted. If the target indications of the form in_j are not used, tar is replaced by out.

Example 1 *To illustrate the concept of generating languages, consider the following P system:*

$$\Pi = (\{a,b,c\}, [_1\ \ ]_1, \{a^2\}, \left\{\{a\} \to \emptyset, \{a\} \to \{a, b_{out}, c_{out}^2\}\right\}, 0).$$

Each of the two symbols a has a non-deterministic choice whether to be erased or to reproduce itself while sending a copy of b and two copies of c into the environment. Therefore, the contents of region 1 can remain a^2 for an arbitrary number $m \geq 0$ of steps, and after that at least one copy of a is erased. The other copy of a can reproduce itself for another $n \geq 0$ steps before being erased. Each of the first m steps, two

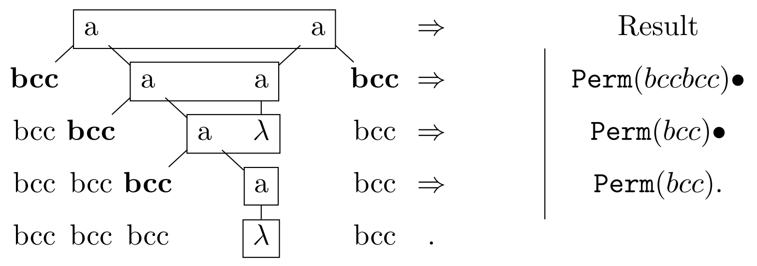

a a ⇒ Result
bcc a a bcc ⇒ Perm(bccbcc)•
bcc bcc a λ bcc ⇒ Perm(bcc)•
bcc bcc bcc a bcc ⇒ Perm(bcc).
bcc bcc bcc λ bcc .

Figure 1. An example of a computation of a P system from Example 1. The lines are only used to hint how the rules are applied.

copies of b and four copies of c are sent out, while in each of the next n steps, only one copy of b and two copies of c are ejected. Therefore, $L(\Pi) = (\texttt{Perm}(bccbcc))^*(\texttt{Perm}(bcc))^*$.

3 Parsability

We first recall a few existing results.

Lemma 1 *Random access machines can be simulated by Turing machines with polynomial slowdown.*

This result will lead to a much simpler proof of the main result.

Lemma 2 *[2] $LOP_*(ncoo, tar) = LOP_1(ncoo, out)$.*

This result means one membrane is enough. Such membrane systems only have one working region, and the destination of the objects in right hand side of the rules may only be *here* and *out*.

Lemma 3 *[2] Any non-cooperative P system can be transformed into an equivalent one such that all objects evolve by some rules (objects not participating in left-hand side of any rule are never produced).*

This condition implies that the system only halts if there are no objects inside the system. Hence, the evolution of any object inside the system eventually leads to some number (possibly zero) of objects in the environment.

Lemma 4 *([2]) Any non-cooperative P system can be transformed into an equivalent one such that the initial contents is $w_1 = \{S\}$, and*

- *S does not appear in the right-hand side of any rule, and*
- *R_1 has no erasing rules, except possibly $\{S\} \to \emptyset$.*

This result means that no object can be erased, except the axiom which may only be erased immediately.

We now proceed to the main result.

Theorem 1 *$LOP_*(ncoo, tar) \subseteq \mathbf{P}$.*

Proof. The proof consists of three parts. First, a few known results are used to simplify the statement of the theorem. Second, a finite-state automaton (with transitions labeled by multisets of terminals) of polynomial size is constructed. Third, acceptance problem is reduced to a search problem in a graph of a polynomial size.

Thanks to Lemma 1, the rest of the proof can be explained at the level of random access machines.

Due to Lemma 2, we assume that an arbitrary membrane system language L is given by a one-membrane system $\Pi = ([_1 \ ]_1, O, w_1, R_1)$.

It is known from Lemma 3 that the condition specified in it does not restrict the generality. Hence, from now we assume that every object A inside the system corresponds to at least one rule that rewrites A.

Without restricting generality, we also assume the normal form specified in Lemma 4. In this case, it is clear that if $w \in T^n$, then during any computation of Π generating w, the number of objects inside the system can never exceed $\max(n, 1)$.

We now build a finite automaton $A = (Q, \Sigma, q_0, \delta, F)$ such that any word $w' \in T^{\leq n}$ is accepted by A if and only if $w' \in L(\Pi)$. Accepting by an automaton with transitions labeled by multisets is understood as follows: a transition labeled by a multiset of weight k can be followed if the multiset composed of the next k input symbols equals the transition label; in this case these input symbols are read.

We define Q as the set of multisets of at most $\max(n, 1)$ objects, Σ as the set of multisets of at most n objects, q_0 is the singleton multiset $\{S\}$, and $F = \{\emptyset\}$. It only remains to define the transition mapping δ of A. We say that $q' \in \delta(q, s)$ if $[_1\ q\ ]_1 \Rightarrow [_1\ q'\ ]_1 s$. It is known (see, e.g., [1]) that computing all transitions from a configuration with k objects takes polynomial time with respect to k; here, $k \leq \max(n, 1)$ (and the degree of such a polynomial does not exceed $|R_1|$), and, moreover, the number of configurations reachable in one step is also polynomial.

Notice that $|Q|$ is polynomial with respect to n (and the degree of such a polynomial does not exceed $|O|+1$).[1] Hence, building A from n and Π can be done in polynomial time, and, moreover, the size of the description of A is also polynomial. Of course, it is sufficient to only examine the reachable states of A.

Running each transition $q' \in \delta(q, s)$ of A on w can be actually done in time $O(|s|)$; however, there are two problems. Firstly, A is non-deterministic, and secondly, A may have transitions labeled by an empty multiset, and removing empty multiset transitions or non-determinism might need too much time or space, or even increase its size too much. Instead, we reduce parsing by A to a graph reachability problem.

Consider a graph $\Gamma = (V, U)$, where $V = \{0, \cdots, n\} \times Q$ and U consists of such transitions $((i, q), (j, q'))$ that $i \leq j$ and $q' \in \delta(q, s)$, where s equals the multiset consisting of $w[i+1], \cdots, w[j]$.

Finally, $w \in L = L_t(G)$ if and only if $w \in L(A)$, and $w \in L(A)$ if and only if there is a path from $(0, q_0)$ to (n, e) in Γ. Note: alternatively, search in A incrementally by prefixes of w. $\square$

[1] Indeed, multisets of size $\leq n$ over O bijectively correspond to multisets of size exactly n over $O \cup \{\lambda\}$. Let $|O| = m$. Moreover, multisets of size n over $O \cup \{\lambda\}$ correspond to n-combinations of $m+1$ possible elements with repetition. For $n > 0$, their number is $|Q| = \binom{n+m}{n} \leq n^{m+1}$.

It is known from [2] that membrane systems language family is included in the family of context-sensitive languages, see also Lemma 4. In [2] one also claims that membrane systems language family is **semilinear**. No formal proof is given, but there is an almost immediate observation that such language is letter-equivalent to that generated by the context-free language with the same rules (languages are letter-equivalent if for every word in one of them there is a word in the other one with the same multiplicities of all symbols; indeed, the difference is only in the order of output). Semilinearity thus follows from Parikh theorem. By Theorem 1, we improve the upper bound:

Corollary 1 $LOP_*(ncoo, tar) \subseteq CS \cap SLIN \cap \mathbf{P}$.

4 An Example

Consider a word $w = babbaa$ and a P system

$$\begin{aligned} \Pi &= ([_1\ ]_1, \{S', S, a, b\}, \{S'\}, R), \text{ where} \\ R &= \{p : \{S'\} \to \{S\},\ q : \{S\} \to \{S^2\},\ r : \{S\} \to \{a, b\}_{out}\}. \end{aligned}$$

Only objects S, S' are productive inside the system, and only objects a, b may be sent outside. Since $|w| = 6$, we only need to examine multisets over S, S' of size up to 6 elements (28 in total). However, out of them only $\{S'\}$, $\{S\}$, $\emptyset$, $\{S^2\}$, $\{S^4\}$, $\{S^6\}$ are reachable. The finite automaton would look as follows (for simplicity of the picture, we wrote i instead of $\{a^i, b^i\}$ as labels):

$$\begin{tikzcd}
{} \arrow[r] & \{S'\} \arrow[d, "0"] & \emptyset & & \{S^6\} \arrow[ll, "6"'] \arrow[dl, shift left, "4"] \arrow[dll, "5"] \arrow[loop below, "3"] \\
& \{S\} \arrow[ur, "1"] \arrow[r, "0"'] & \{S^2\} \arrow[u, "2"] \arrow[r, shift right, "0"'] \arrow[loop below, "1"] & \{S^4\} \arrow[l, shift right, "3"'] \arrow[ul, "4"] \arrow[ur, shift left, "1"] \arrow[loop below, "2"] &
\end{tikzcd}$$

We now check the word w:

- states after reading λ: $\{S'\}$, $\{S\}$, $\{S^2\}$, $\{S^4\}$;

- states after reading *ba*: $\emptyset$, $\{S^2\}$, $\{S^4\}$, $\{S^6\}$;
- states after reading *babbaa*: $\emptyset$, $\{S^4\}$. The input is accepted.

5 Conclusions

We have shown that there exists an algorithm deciding the word membership problem of membrane systems languages in polynomial time with respect to the length of the word. The degree of such polynomial may depend on the number of rules and on the size of the alphabet. Hence, the position of the membrane systems language family in the language family hierarchy is between $REG \bullet \texttt{Perm}(REG)$ and $CS \cap SLIN \cap \mathbf{P}$.

Acknowledgments Artiom Alhazov gratefully acknowledges the support of the Japan Society for the Promotion of Science and the Grant-in-Aid for Scientific Research, project 20·08364. All authors acknowledge the support by the Science and Technology Center in Ukraine, project 4032.

References

[1] A. Alhazov: Maximally Parallel Multiset-Rewriting Systems: Browsing the Configurations. In: M.A. Gutiérrez-Naranjo, A. Riscos-Núñez, F.J. Romero-Campero, D. Sburlan: RGNC report 01/2005, University of Seville, *Third Brainstorming Week on Membrane Computing*, Fénix Editora, Sevilla, 2005, 1–10.

[2] A. Alhazov, C. Ciubotaru, Yu. Rogozhin, S. Ivanov: The Family of Languages Generated by Non-Cooperative Membrane Systems. In: M. Gheorghe, Th. Hinze, Gh. Păun: *Preproceedings of the Eleventh Conference on Membrane Computing, CMC11*, Jena, Verlag ProBusiness Berlin, 2010, 37–51, and *Lecture Notes in Computer Science* **6501**, to appear.

[3] Gh. Păun, G. Rozenberg, A. Salomaa, Eds.: *Handbook of Membrane Computing*. Oxford University Press, 2010.

[4] Gh. Păun: Membrane Computing. An Introduction, Springer, 2002.

[5] G. Rozenberg, A. Salomaa, Eds.: *Handbook of Formal Languages*, vol. 1-3, Springer, 1997.

[6] P systems webpage. `http://ppage.psystems.eu/`

A. Alhazov[1,2], C. Ciubotaru[1], S. Ivanov[1,3],
Yu. Rogozhin[1]

Received November 1, 2010

[1] Institute of Mathematics and Computer Science
Academy of Sciences of Moldova
5 Academiei str., Chişinău, MD-2028, Moldova
E–mail: {`artiom, chebotar, sivanov, rogozhin`}`@math.md`

[2] FCS, Department of Information Engineering
Graduate School of Engineering
Hiroshima University,
Higashi-Hiroshima 739-8527 Japan

[3] Faculty of Computers,
Informatics and Microelectronics,
Technical University of Moldova,
Ştefan cel Mare 168, Chişinău MD-2004 Moldova

Computer Science Journal of Moldova, vol.**18**, no.2(53), 2010

Minimal Parallelism and Number of Membrane Polarizations

Artiom Alhazov

Abstract

It is known that the satisfiability problem (SAT) can be efficiently solved by a uniform family of P systems with active membranes with two polarizations working in a maximally parallel way. We study P systems with active membranes without non-elementary membrane division, working in minimally parallel way. The main question we address is what number of polarizations is sufficient for an efficient computation depending on the types of rules used.

In particular, we show that it is enough to have four polarizations, sequential evolution rules changing polarizations, polarizationless non-elementary membrane division rules and polarizationless rules of sending an object out. The same problem is solved with the standard evolution rules, rules of sending an object out and polarizationless non-elementary membrane division rules, with six polarizations. It is an open question whether these numbers are optimal.

1 Introduction

Membrane computing with symbol-objects is a biologically inspired framework of distributed parallel multiset processing; see [11] for an overwiew and [15] for the comprehensive bibliography. The most addressed questions are completeness (solving every solvable problem) and efficiency (solving hard problems in feasible time). We focus on the latter one.

An interesting class of membrane systems are those with active membranes (see [10]), where membrane division can be used for solving computationally hard problems in polynomial time. Let us mention a few results:

- A *semi–uniform* **solution** to `SAT` using three polarizations and division for non-elementary membranes, [10].
- A *polarizationless* solution, [3].
- Using only division for elementary membranes, with three polarizations, [12].
- A *uniform* solution, with elementary membrane division, [13].
- Using only *two polarizations*, in a uniform way, with elementary membrane division, [4].
- Computational **completeness** of P systems with three polarizations and three membranes, [11].
- Using only *two polarizations* and two membranes, [6].
- Using only *one membrane*, with two polarizations, [5].
- *Polarizationless* systems are complete, with no known bound on the number of membranes, [2].
- Solving `SAT` in a **minimally parallel** way, using non-elementary membrane division (replicating both objects and inner membranes), [7].
- Avoiding polarizations by using rules *changing membrane labels*. Using (up to the best author's knowledge) either cooperative rules or non-elementary division as above, [9].

Given a P system, a rule and an object, whether this rule is applicable to this object in some membrane might depend on both membrane label (that usually cannot be changed) and membrane polarization. Essentially, the number of polarizations is the number of states that can be encoded directly on the membrane.

Minimal parallelism provides less synchronization between the objects, so one might expect the need of a stronger control, i.e., more polarizations. It is not difficult to construct the system in such a way that the rules are global (i.e., the membrane labels are not distinguished), most likely without adding additional polarizations. In this way the results dealing with the number of polarizations can be reformulated in terms of number of membrane labels (in that case, the systems have no polarizations, but the rules are allowed to modify membrane labels).

This paper is an extended version of [1].

2 Preliminaries

2.1 Solvability by P systems with input

Definition 1 *A* P system with input *is a tuple* (Π, Σ, i_Π)*, where (a)* Π *is a P system with working alphabet , with* m *membranes labelled with* $1, \cdots, m$*, and initial multisets* $w_1, \cdots, w_m$ *(over* $O - \Sigma$*) associated with them; (b)* $\Sigma \subseteq O$ *is an (input) alphabet, (c)* i_Π *is the label of a distinguished (input) membrane.*

The *initial configuration* of (Π, Σ, i_Π) with an input multiset w over Σ is

$$(\mu, w_1, \cdots, w_{i_\Pi} \cup w, \cdots, w_m).$$

We call (Π, Σ, i_Π) a *decisional* P system with input if there exists two distinguished objects $\mathtt{yes}, \mathtt{no} \in O$ and for any valid input (see *cod* function in the definition below) all its computations send to the environment exactly one object, either `yes` (in this case the computation is called an *accepting* one) or `no`. Moreover, (Π, Σ, i_Π) is called *confluent* if for any valid input all its computations halt in the same configuration.

Definition 2 *Consider a decision problem* $X = (I_X, \theta_X)$*:* I_X *is the set of possible instances of* X *and* θ_X *is a boolean function over* I_X*. We say that* X *is solvable in polynomial time by a uniform family of P systems* $\mathbf{\Pi} = (\Pi(n))_{n \in \mathbb{N}}$ *if the following conditions hold:*

- *The family* $\mathbf{\Pi}$ *is polynomially constructible, i.e., there exists a deterministic Turing machine constructing the system* $\Pi(n)$ *from* n *in polynomial time.*
- *There exists a pair* (s, cod) *of polynomial-time computable functions mapping every instance* $u \in I_X$ *of the problem* X *into a natural number and a multiset (over the alphabet of* $\Pi(s(u))$*), respectively. The instance* u *is to be solved by a system* $\Pi(s(u))$ *with the multiset* $cod(u)$ *placed in the input membrane, as described below.*
- *The family* $\mathbf{\Pi}$ *is polynomially bounded with respect to* (X, cod, s)*, i.e., there exists a polynomial function* $p(n)$ *such that for each* $u \in I_X$ *every computation of the system* $\Pi(s(u))$ *with input* $cod(u)$ *is halting in at most* $p(s(u))$ *steps.*
- *The family* $\mathbf{\Pi}$ *is sound with respect to* (X, cod, s)*, i.e., for each* $u \in I_X$ *if there exists an accepting computation of* $\Pi(s(u))$ *with input* $cod(u)$*, then* $\theta_X(u) = 1$.
- *The family* $\mathbf{\Pi}$ *is complete with respect to* (X, cod, s)*, i.e., for each* $u \in I_X$ *if* $\theta_X(u) = 1$*, then every computation of* $\Pi(s(u))$ *with input* $cod(u)$ *is an accepting one.*

2.2 P systems with active membranes

Definition 3 *A P system* with active membranes *is a P system with the working alphabet* O*, with the set* H *of membrane labels, with the set* E *of polarizations, and with the rules of the following forms:*

(a) $[\ a \rightarrow u\ ]_h^e$ *for* $a \in O$, $u \in O^*$, $h \in H$ *and* $e \in E$*. These are object evolution rules. An object* $a \in O$ *in the region associated with a membrane with label* h *and polarization* e *evolves to a multiset* $u \in O^*$.

(b) $a[\ \ ]_h^e \rightarrow [\ b\ ]_h^{e'}$ *for* $a, b \in O$, $h \in H$ *and* $e, e' \in E$*. These are send–in communication rules. An object* a *from the region immediately outside a membrane with label* h *and polarization* e

is introduced in this membrane, transformed into b and changing the polarization of the membrane to e'.

(c) $[\ a\ ]_h^e \rightarrow [\ \ ]_h^{e'} b$ *for* $a, b \in O$, $h \in H$ *and* $e, e' \in E$. *These are send–out communication rules. An object a is sent out from the region associated with membrane with label h and polarization e to the region immediately outside, transformed into b and changing the polarization of the membrane to e'.*

(d) $[\ a\ ]_h^e \rightarrow a$ *for* $a, b \in O$, $h \in H$ *and* $e \in E$. *These are dissolution rules. A membrane with label h and polarization e is dissolved in reaction with an object a, transformed into b. The skin is never dissolved.*

(e) $[\ a\ ]_h^e \rightarrow [\ b\ ]_h^{e'} [\ c\ ]_h^{e''}$ *for* $a, b, c \in O$, $h \in H$ *and* $e, e', e'' \in E$. *These are division rules for elementary membranes. An elementary membrane can be divided into two membranes with the same label, possibly with different polarizations, possibly transforming some objects.*

Generally, rules of type (a) are executed in parallel, while at most one rule out of all rules of types (b), (c), (d), (e) can be applied to the same membrane in the same step. We also speak about the sequential version

$$(a_s'')\ [\ a\ ]_h^e \rightarrow [\ u\ ]_h^{e'} \text{ for } a \in O,\ u \in O^*,\ h \in H \text{ and } e, e' \in E.$$

of rules (a) (let us use $''$ to indicate that the rule is allowed to change the polarization of the membrane) and their modifications $(b_0), (c_0), (d_0), (e_0)$, (a_{0s}'), $(b_0'), (c_0'), (e_0')$ (here, 0 represents that the rules neither distinguish polarization nor change it, while $'$ means that the rule is allowed to change membrane label).

2.3 Minimal parallelism

In [8], the *minimal parallelism* has been formalized as follows (throughout this paper, each set R_j is associated to a membrane):

$$\begin{aligned} App(\Pi, C, min) = \{ \; & R' \in App(\Pi, C, asyn) \mid \text{ there is no} \\ & R'' \in App(\Pi, C, asyn) \text{ such that} \\ & (R'' - R') \cap R_j \neq \emptyset \text{ for some } j \text{ with} \\ & R' \cap R_j = \emptyset,\ 1 \leq j \leq h \; \}. \end{aligned}$$

We are not going to define all notations used here. In our context, this definition means that minimally parallel application of rules to a configuration consists of all applicable multisets R' that cannot be extended by a rule corresponding to a membrane for which no rule appears in R'.

There exist different interpretations of minimal parallelism. For instance, the original definition of maximal parallelism introduced in [7] is formalized in [14] and called there *base vector minimal parallelism*:

$$\begin{aligned} App(\Pi, C, min_G) = \{ \; & R''' \in App(\Pi, C, asyn) \mid \text{ there is} \\ & R' \in App(\Pi, C, asyn), \text{ such that } R' \subseteq R''', \\ & |R' \cap R_j| \leq 1 \text{ for all } j,\ 1 \leq j \leq h, \text{ and} \\ & \text{there is no } R'' \in App(\Pi, C, asyn) \text{ such that} \\ & (R'' - R') \cap R_j \neq \emptyset \text{ for some } j \text{ with} \\ & R' \cap R_j = \emptyset,\ 1 \leq j \leq h \; \}. \end{aligned}$$

Without discussing all technicalities, we point out that base vector minimally parallel application of rules consists of all extensions of multisets R', which represent maximally parallel choice of sets R_j used sequentially. Hence, the latter mode is identical to the following:

$$\begin{aligned} \{ \; & R''' \in App(\Pi, C, asyn) \mid \exists R' \in App(\Pi, C, seq_{set}), R' \subseteq R''' \\ & \text{and } \; \nexists R'' \in App(\Pi, C, seq_{set}) : R' \subseteq R'' \; \}, \text{ where} \end{aligned}$$

$$\begin{aligned} App(\Pi, C, seq_{set}) = \{ \; & R' \in App(\Pi, C, asyn) \mid \\ & |R' \cap R_j| \leq 1 \text{ for all } j,\ 1 \leq j \leq h. \end{aligned}$$

In this way, one can first restrict applicable multisets to those having at most one rule corresponding to a membrane, then take maximally parallel ones from them (i.e., those whose extensions do not belong to the same restriction), and finally take their unrestricted extensions.

Luckily, the constructions presented in this paper work equally well for both definitions of minimal parallelism. Indeed, they do not use rules of type (b) or its modifications; hence, in one step a membrane reacts only with objects in the associated region. This means that selection of rules for each membrane is done independently, so different membranes do not compete for objects and the system behaves identically in both modes.

Hence, we can simply follow the basic idea introduced already in [7]: for every membrane, at least one rule - if possible - has to be used.

The following remarks describe applicability, maximal applicability and applying rules, respectively.

- The rules of type (a) may be applied in parallel. At one step, a membrane can be the subject of *only one* rule of types $(a'_{0s}), (a''_s)$ and $(b), (c), (d), (e)$ with their modifications.
- In one step, one object of a membrane can be used by only one rule (non-deterministically chosen), but **for every membrane at least one object** that can evolve by one rule of any form, must evolve (no rules associated to a membrane are applied only if none are applicable for the objects that do not evolve).
- If at the same time a membrane is divided by a rule of type (e) and there are objects in this membrane which evolve by means of rules of type (a), then we suppose that first the evolution rules of type (a) are used, and then the division is produced. Of course, this process takes only one step.

3 Using Rules (a''_s)

The three size parameters of the `SAT` problem are the number m of clauses, the number n of variables and the total number l of occurrences of variables in clauses (clearly, $l \leq mn$: without restricting generality, we could assume that no variable appears in the same clause more than once, with or without negation).

Theorem 1 *A uniform family of confluent P systems with rules* $(a_s''), (c_0)$*,* (e_0) *working in minimally parallel way can solve* `SAT` *with four polarizations in* $O(l(m+n))$ *number of steps.*

Proof. The main idea of the construction is to implement a maximally parallel step sequentially. For this, a "control" object will be changing the polarization, and then an input object or a clause object will be restoring it. Since the input is encoded in l objects, changing and restoring polarization will happen for l times, the counting is done by the "control" object.

Consider a propositional formula in the conjunctive normal form:

$$\begin{aligned}
\beta &= C_1 \vee \cdots \vee C_m, \\
C_i &= y_{i,1} \wedge \cdots \wedge y_{i,l_i},\ 1 \le i \le m, \text{ where} \\
y_{i,k} &\in \{x_j, \neg x_j \mid 1 \le j \le n\},\ 1 \le i \le m, 1 \le k \le l_i, \\
l &= \sum_{i=1}^{m} l_i.
\end{aligned}$$

Let us encode the instance of β in the alphabet $\Sigma(\langle n,m,l\rangle)$ by multisets X, X' of the clause-variable pairs such that the variable appears in the clause without negation, with negation or neither:

$$\begin{aligned}
\Sigma(\langle n,m,l\rangle) &= \{v_{j,i,1,s} \mid 1 \le j \le m,\ 1 \le i \le n,\ 1 \le s \le 2\}, \\
X &= \{(v_{j,i,1,1}, 1) \mid x_i \in \{y_{j,k} \mid 1 \le k \le l_j\}, \\
&\quad 1 \le j \le m,\ 1 \le i \le n\}, \\
X' &= \{(v_{j,i,1,2}, 1) \mid \neg x_i \in \{y_{j,k} \mid 1 \le k \le l_j\}, \\
&\quad 1 \le j \le m,\ 1 \le i \le n\}.
\end{aligned}$$

We construct the following P system:

$$\begin{aligned}
\Pi(\langle n,m,l\rangle) &= (O, H, E, [\,[\ ]_2^0 [\ ]_3^0\,]_1^0, w_1, w_2, w_3, R), \text{ with} \\
O &= \{v_{j,i,k,s} \mid 1 \le j \le m,\ 1 \le i \le n,
\end{aligned}$$

$$\begin{aligned}
& 1 \le k \le m+n+1,\ 1 \le s \le 4\} \\
\cup\ & \{d_{i,k} \mid 1 \le i \le m+n+1,\ 1 \le k \le 2l\} \\
\cup\ & \{t_{i,k}, f_{i,k} \mid 1 \le i \le n,\ 1 \le k \le l\} \\
\cup\ & \{d_i \mid 1 \le i \le m+n+1\} \cup \{S, Z, \mathtt{yes}, \mathtt{no}\} \\
\cup\ & \{z_k \mid 1 \le k \le (4l+3)n + m(4l+1) + 2\} \\
w_1 =\ & \lambda,\ w_2 = d_1,\ w_3 = z_0,\ H = \{1,2,3\},\ E = \{0,1,2,3\},
\end{aligned}$$

and the rules are listed below. The computation consists of three stages.

1. Producing 2^n membranes with label 2, corresponding to the possible assignments of variables $x_1, \cdots, x_n$ and selecting clauses that are satisfied for every assignment (groups A and C of rules).

2. Checking for all assignments whether all clauses are satisfied (groups B and D of rules).

3. Generating `yes` from the positive answer, and sending it to the environment. Generating `no` from the timeout (during the first two stages the number of steps is counted in the object in membrane with label 3) and sending it to the environment if there was no positive answer (groups E and F of rules).

Stage 1 consists of n cycles and stage 2 consists of m cycles. Each cycle's aim is to process all l objects, i.e., each object counts the number of cycles completed, and in the first stage the clauses are evaluated while in the second stage the presence of each clause is checked.

In the case of maximal parallelism, a cycle could be performed in a constant number of (actually, one or two) steps, while the minimal parallelism cannot guarantee that all objects are processed. The solution used here is the following. A cycle consists of marking (setting the last index to 3 or 4) all l objects one by one while performing the necessary operation, and then unmarking (setting the last index to 1 or 2) all of them. Marking or unmarking an object happens in two steps: the control object changes the polarization from 0 to 1, 2 (to mark) or to 3 (to unmark), and then one of the objects that has not yet been (un)marked is processed, resetting the polarization to 0.

Control objects in membrane 2: select clauses

A1 (for variable i: divide)
$[\ d_i\ ] \to [\ t_{i,0}\ ][\ f_{i,0}\ ]$, $1 \le i \le n$

A2 (process and mark all l objects)
$[\ t_{i,k-1}\ ]^0 \to [\ t_{i,k}\ ]^1$, $1 \le i \le n$, $1 \le k \le l$
$[\ f_{i,k-1}\ ]^0 \to [\ f_{i,k}\ ]^2$, $1 \le i \le n$, $1 \le k \le l$

A3 (prepare to unmark objects)
$[\ t_{i,l}\ ]^0 \to [\ d_{i,0}\ ]^0$, $1 \le i \le n$
$[\ f_{i,l}\ ]^0 \to [\ d_{i,0}\ ]^0$, $1 \le i \le n$

A4 (unmark all l objects)
$[\ d_{i,k-1}\ ]^0 \to [\ d_{i,k}\ ]^3$, $1 \le i \le n$, $1 \le k \le l$

A5 (switch to the next variable)
$[\ d_{i,l}\ ]^0 \to [\ d_{i+1}\ ]^0$, $1 \le i \le n$

Control objects in membrane 2: check clauses

B1 (test if clause i is satisfied)
$[\ d_{n+i}\ ]^0 \to [\ d_{n+i,1}\ ]^2$, $1 \le i \le m$

B2 (process and mark the other $l-1$ objects)
$[\ d_{n+i,k-1}\ ]^0 \to [\ d_{n+i,k}\ ]^1$, $1 \le i \le m$, $1 \le k \le l$

B3 (unmark all l objects)
$[\ d_{n+i,l+k-1}\ ]^0 \to [\ d_{n+i,l+k}\ ]^3$, $1 \le i \le m$, $1 \le k \le l$

B4 (switch to the next clause)
$[\ d_{n+i,2l}\ ]^0 \to [\ d_{n+i+1}\ ]^0$, $1 \le i \le m$

B5 (send a positive answer)
$[\ d_{m+n+1}\ ] \to [\ \ ]S$

Input objects in membrane 2: select clauses

C1 (mark an object)
$[\ v_{j,i,k,s}\ ]^p \rightarrow [\ v_{j,i,k+1,s+2}\ ]^0$,
$1 \leq i \leq m$, $1 \leq j \leq n$, $1 \leq k \leq m$, $k \neq m$, $1 \leq s \leq 2$, $1 \leq p \leq 2$

C2 (a true variable present without negation or a false variable present with negation satisfies the clause)
$[\ v_{j,i,i,s}\ ]^s \rightarrow [\ v_{j,i,i+1,3}\ ]^0$, $1 \leq i \leq m$, $1 \leq j \leq n$, $1 \leq s \leq 2$

C3 (a true variable present with negation or a false variable present without negation does not satisfy the clause)
$[\ v_{j,i,i,3-s}\ ]^s \rightarrow [\ v_{j,i,i+1,4}\ ]^0$, $1 \leq i \leq m$, $1 \leq j \leq n$, $1 \leq s \leq 2$

C4 (unmark an object)
$[\ v_{j,i,k,s+2}\ ]^3 \rightarrow [\ v_{j,i,k,s}\ ]^0$,
$1 \leq i \leq m$, $1 \leq j \leq n$, $2 \leq k \leq m+1$, $1 \leq s \leq 2$

Input objects in membrane 2: check clauses

D1 (check if the clause is satisfied at least by one variable)
$[\ v_{j,i,m+j,1}\ ]^2 \rightarrow [\ v_{j,i,k+1,3}\ ]^0$, $1 \leq i \leq m$, $1 \leq j \leq n$, $1 \leq s \leq 2$

D2 (mark an object)
$[\ v_{j,i,m+k,s}\ ]^1 \rightarrow [\ v_{j,i,k+1,s+2}\ ]^0$,
$1 \leq i \leq m$, $1 \leq j \leq n$, $1 \leq k \leq n$, $1 \leq s \leq 2$

D3 (unmark an object)
$[\ v_{j,i,m+k,s+2}\ ]^3 \rightarrow [\ v_{j,i,k,s}\ ]^0$,
$1 \leq i \leq m$, $1 \leq j \leq n$, $2 \leq k \leq n+1$, $1 \leq s \leq 2$

Control objects in membrane 3

E1 (count)
$[\ z_{k-1}\ ]^0 \rightarrow [\ z_k\ ]^0$, $1 \leq k \leq N = (4l+3)n + m(4l+1) + 2$

E2 (send time-out object)
$[\ z_N\ ] \rightarrow [\ \ ]Z$

Control objects in the skin membrane

F1 (a positive result generates the answer)
$[\ S\ ]^0 \to [\ \texttt{yes}\ ]^1$

F2 (without the positive answer, the time-out generates the negative answer)
$[\ Z\ ]^0 \to [\ \texttt{no}\ ]^0$

F3 (send the answer)
$[\ \texttt{yes}\ ] \to [\ \ ]\texttt{yes}$
$[\ \texttt{no}\ ] \to [\ \ ]\texttt{no}$

Let us now explain how the system works in more details.

Like the input objects, the control objects keep track of the number of cycles completed. The control object also remembers whether marking or unmarking takes place, as well as the number of objects already (un)marked. Moreover, the control object is responsible to pass the "right" information to the objects via polarization: in stage 1, 1 if the variable is true, and 2 if the variable is false; in stage 2, 1 if the clause is already found, and 2 if the clause is being checked for.

During the first stage, an object $v_{j,i,1,s}$ is transformed into $v_{j,i,n+1,t}$, where $t = 1$ if variable x_j satisfies clause C_i, or $t = 2$ if not. The change of the last index from s to t happens when the third index is equal to i. Notice that although only information about what clauses are satisfied seems to be necessary for checking if β is true for the given assignment of the variables, the information such as the number of cycles completed is kept for synchronization purposes, and the other objects are kept so that their total number remains l. The control object d_1 is transformed into d_{n+1}. Stage 1 takes $(4l + 3)n$ steps.

If some clause is not satisfied, then the computation in the corresponding membrane is "stuck" with polarization 2. Otherwise, during the second stage an object $v_{j,i,n+1,t}$ is transformed into $v_{j,i,n+m+1,t}$, while the control object d_{n+1} becomes d_{m+n+1}. Stage 2 takes $m(4l+1)$ steps, plus one extra step to send objects S to skin, if any.

After stage 2 is completed, one copy of S, if any, is transformed into yes, changing the polarization of the skin membrane. In the same time yes, if it has been produced, is sent out, object Z comes to the skin from region 3. If the polarization of the skin remained 0, Z changes to no, which is then sent out. Depending on the answer, stage 3 takes 2 or 4 steps. In either case, the result is sent out in the last step of the computation. □

Notice that membrane labels are not indicated in the rules. This means that the system is organized in such a way that the rules are *global*, i.e., the system would work equally well starting with the configuration $\mu = [\ w_1[\ w_2\ ]_1^0[\ w_3\ ]_1^0\ ]_1^0$, the labels were only given for the simplicity of explanation.

Using the remark in the end of the Introduction, we obtain

Corollary 1 *A uniform family of confluent polarizationless P systems with rules* $(a'_{0s}), (c_0), (e_0)$ *working in minimally parallel way can solve* SAT *with membrane labels of four kinds.*

The statement follows directly from the possibility of rewriting a global rule $[\ a\ ]^e \rightarrow [\ u\ ]^{e'}$ of type (a''_s) in a rule $[\ a\ ]_e \rightarrow [\ u\ ]_{e'}$ of type (a'_{0s}) (which is polarizationless but is able to change the membrane label).

4 Using Rules (a)

An informal idea of this section is to replace rules of type (a''_s) with rules (a) producing additional objects, and rules (c), sending an additional object out to change the polarization.

Theorem 2 *A uniform family of confluent P systems with rules* $(a), (c)$*,* (e_0) *working in minimally parallel way can solve* SAT *with six polarizations in* $O(l(m+n))$ *number of steps.*

Proof. The strategy used in the construction below is similar to that of the previous theorem. However, since the application of the evolution rules no longer changes the polarization of the membrane, the control symbols $d_{i,k}$, $t_{i,k}$, $f_{i,k}$ no longer "operate" in polarization 0, but rather in polarization that toggles between 0 (for even k) and 5 (for odd k), to prevent multiple applications of evolution rules in a row in the same membrane. Moreover, the input objects are actually allowed to evolve in parallel (and the degree of parallelism is chosen non-deterministically), but in the end of both halves of a cycle it is possible to count the number of extra objects produced, to make sure that all l objects have been processed.

For the same propositional formula

$$\begin{aligned}
\beta &= C_1 \vee \cdots \vee C_m, \\
C_i &= y_{i,1} \wedge \cdots \wedge y_{i,l_i},\ 1 \le i \le m,\ \text{where} \\
y_{i,k} &\in \{x_j, \neg x_j \mid 1 \le j \le n\},\ 1 \le i \le m, 1 \le k \le l_i, \\
l &= \sum_{i=1}^{m} l_i.
\end{aligned}$$

and the same encoding of the instance of β in the alphabet $\Sigma(\langle n,m,l\rangle)$ by multisets X, X',

$$\begin{aligned}
\Sigma(\langle n,m,l\rangle) &= \{v_{j,i,1,s} \mid 1 \le j \le m,\ 1 \le i \le n,\ 1 \le s \le 2\}, \\
X &= \{(v_{j,i,1,1}, 1) \mid x_i \in \{y_{j,k} \mid 1 \le k \le l_j\}, \\
&\quad\ 1 \le j \le m,\ 1 \le i \le n\}, \\
X' &= \{(v_{j,i,1,2}, 1) \mid \neg x_i \in \{y_{j,k} \mid 1 \le k \le l_j\}, \\
&\quad\ 1 \le j \le m,\ 1 \le i \le n\}.
\end{aligned}$$

we construct the following P system:

$$\begin{aligned}
\Pi(\langle n,m,l\rangle) &= (O, H, E, [\,[\ ]_2^0 [\ ]_3^0\,]_1^0, w_1, w_2, w_3, R),\ \text{with} \\
O &= \{v_{j,i,k,s} \mid 1 \le j \le m,\ 1 \le i \le n, \\
&\quad\ 1 \le k \le m+n+1,\ 1 \le s \le 4\} \\
&\cup\ \{d_{i,k} \mid 1 \le i \le m+n+1,\ 1 \le k \le 2l\} \\
&\cup\ \{t_{i,k}, f_{i,k} \mid 1 \le i \le n,\ 1 \le k \le l\} \\
&\cup\ \{d_i \mid 1 \le i \le m+n+1\} \cup \{S, Z, \mathtt{yes}, \mathtt{no}\} \\
&\cup\ \{z_k \mid 1 \le k \le (4l+3)n + m(4l+1) + 2\} \\
&\cup\ \{o_{i,j} \mid 0 \le i \le 5,\ 0 \le j \le 5\} \\
w_1 &= \lambda,\ w_2 = d_1,\ w_3 = z_0, \\
&\quad\ H = \{1,2,3\},\ E = \{0,1,2,3,4,5\},
\end{aligned}$$

and the rules are listed below. The computation stages are the same as in the previous proof.

1. Producing 2^n membranes corresponding to the possible variables assignments; selecting satisfied clauses (groups A and C).
2. Checking whether all clauses are satisfied (groups B and D).
3. Generating the answer and sending it to the environment. (groups E and F).

Stage 1 consists of n cycles and stage 2 consists of m cycles. Each cycle's aim is to process all l objects, i.e., each object counts the number of cycles completed, and in the first stage the clauses are evaluated while in the second stage the presence of each clause is checked.

A cycle consists of marking (setting the last index to 3 or 4) all l objects one by one while performing the necessary operation, and then unmarking (setting the last index to 1 or 2) all of them. Marking or unmarking an object generally happens in five steps:

1. the control object produces two "polarization changers",
2. one of them changes the polarization from 0 or 5 to 1, 2 (to mark) or to 3 (to unmark),
3. one of the objects that has not yet been (un)marked is processed, producing a "witness" — yet another "polarization changer",
4. the "witness" switches the polarization to 4,
5. the second "changer" produced in step 1 of this routine changes the polarization to 5 or 0.

Notice, however, that "step" 3 might actually take more than one step (more objects can be (un)marked in parallel, or even in a row, creating a supply of "witnesses"). Step 4 might actually be executed in parallel with the last step of "step" 3 (sending out a previous "witness" while producing more). Finally, "step" 3 might even be skipped if a previous "witness" is already there. What matters is that the whole (un)marking routine takes at most $5l$ steps.

Changing polarization of membrane 2

O1 (change from i to j)
$[\ o_{i,j}\ ]^i \rightarrow [\ \ ]^j\ o_{4,5}$, $0 \leq i \leq 5$, $0 \leq j \leq 5$

O2 (“witnesses” of D2 are “compatible” with “witnesses” of D1; this does not interfere with the rest of the computation)
$[\ o_{1,4}\ ]^2 \rightarrow [\ \ ]^4\ o_{4,5}$

Control objects in membrane 2: select clauses

A1 (for variable i: divide)
$[\ d_i\ ] \rightarrow [\ t_{i,0}\ ][\ f_{i,0}\ ]$, $1 \leq i \leq n$

A2 (process and mark all l objects)
$[\ t_{i,k-1} \rightarrow t_{i,k}o_{0,1}o_{4,5}\ ]^0$, $1 \leq i \leq n$, $1 \leq k \leq l$, k is odd
$[\ f_{i,k-1} \rightarrow f_{i,k}o_{0,2}o_{4,5}\ ]^0$, $1 \leq i \leq n$, $1 \leq k \leq l$, k is odd
$[\ t_{i,k-1} \rightarrow t_{i,k}o_{5,1}o_{4,0}\ ]^5$, $1 \leq i \leq n$, $1 \leq k \leq l$, k is even
$[\ f_{i,k-1} \rightarrow f_{i,k}o_{5,2}o_{4,0}\ ]^5$, $1 \leq i \leq n$, $1 \leq k \leq l$, k is even

A3 (prepare to unmark objects)
$[\ t_{i,l} \rightarrow d_{i,0}\ ]^0$, $1 \leq i \leq n$, if l is even
$[\ f_{i,l} \rightarrow d_{i,0}\ ]^0$, $1 \leq i \leq n$, if l is even
$[\ t_{i,l} \rightarrow d_{i,0}o_{5,0}\ ]^5$, $1 \leq i \leq n$, if l is odd
$[\ f_{i,l} \rightarrow d_{i,0}o_{5,0}\ ]^5$, $1 \leq i \leq n$, if l is odd

A4 (unmark all l objects)
$[\ d_{i,k-1} \rightarrow d_{i,k}o_{0,3}o_{4,5}\ ]^0$, $1 \leq i \leq n$, $1 \leq k \leq l$, k is odd
$[\ d_{i,k-1} \rightarrow d_{i,k}o_{5,3}o_{4,0}\ ]^0$, $1 \leq i \leq n$, $1 \leq k \leq l$, k is even

A5 (switch to the next variable)
$[\ d_{i,l} \rightarrow d_{i+1}\ ]^0$, $1 \leq i \leq n$, if l is even
$[\ d_{i,l} \rightarrow d_{i+1}o_{5,0}\ ]^5$, $1 \leq i \leq n$, if l is odd

Control objects in membrane 2: check clauses

B1 (test if clause i is satisfied)
$[\ d_{n+i} \rightarrow d_{n+i,1}o_{0,2}o_{4,5}\ ]^0$, $1 \leq i \leq m$

B2 (process and mark the other $l-1$ objects)
$[\ d_{n+i,k-1} \to d_{n+i,k}o_{0,1}o_{4,5}\ ]^0$, $1 \leq i \leq m$, $1 \leq k \leq l$, k is odd
$[\ d_{n+i,k-1} \to d_{n+i,k}o_{5,1}o_{4,0}\ ]^0$, $1 \leq i \leq m$, $1 \leq k \leq l$, k is even

B3 (unmark all l objects)
$[\ d_{n+i,l+k-1} \to d_{n+i,l+k}o_{0,3}o_{4,5}\ ]^0$, $1 \leq i \leq m$, $1 \leq k \leq l$, $l+k$ is odd
$[\ d_{n+i,l+k-1} \to d_{n+i,l+k}o_{5,3}o_{4,0}\ ]^0$, $1 \leq i \leq m$, $1 \leq k \leq l$, $l+k$ is odd

B4 (switch to the next clause)
$[\ d_{n+i,2l} \to d_{n+i+1}\ ]^0$, $1 \leq i \leq m$

B5 (send a positive answer)
$[\ d_{m+n+1}\ ]^0 \to [\ \ ]^0 S$

Input objects in membrane 2: select clauses

C1 (mark an object)
$[\ v_{j,i,k,s} \to v_{j,i,k+1,s+2}o_{p,4}\ ]^p$,
$1 \leq i \leq m$, $1 \leq j \leq n$, $1 \leq k \leq m$, $k \neq m$, $1 \leq s \leq 2$, $1 \leq p \leq 2$

C2 (a true variable present without negation or a false variable present with negation satisfies the clause)
$[\ v_{j,i,i,s} \to v_{j,i,i+1,3}o_{s,4}\ ]^s$, $1 \leq i \leq m$, $1 \leq j \leq n$, $1 \leq s \leq 2$

C3 (a true variable present with negation or a false variable present without negation does not satisfy the clause)
$[\ v_{j,i,i,3-s} \to v_{j,i,i+1,4}o_{s,4}\ ]^s$, $1 \leq i \leq m$, $1 \leq j \leq n$, $1 \leq s \leq 2$

C4 (unmark an object)
$[\ v_{j,i,k,s+2} \to v_{j,i,k,s}o_{3,4}\ ]^3$,
$1 \leq i \leq m$, $1 \leq j \leq n$, $2 \leq k \leq m+1$, $1 \leq s \leq 2$

Input objects in membrane 2: check clauses

D1 (check if the clause is satisfied at least by one variable)
$[\ v_{j,i,m+j,1} \to v_{j,i,k+1,3}o_{1,4}\ ]^2$, $1 \leq i \leq m$, $1 \leq j \leq n$, $1 \leq s \leq 2$

D2 (mark an object)
$[\ v_{j,i,m+k,s} \to v_{j,i,k+1,s+2}o_{1,4}\ ]^1$,
$1 \le i \le m,\ 1 \le j \le n,\ 1 \le k \le n,\ 1 \le s \le 2$

D3 (unmark an object)
$[\ v_{j,i,m+k,s+2} \to v_{j,i,k,s}o_{3,4}\ ]^3$,
$1 \le i \le m,\ 1 \le j \le n,\ 2 \le k \le n+1,\ 1 \le s \le 2$

Control objects in membrane 3

E1 (count)
$[\ z_{k-1} \to z_k\ ]^0,\ 1 \le k \le N = (10l+5)n + m(10l+1) + 2$

E2 (send time-out object)
$[\ z_N\ ]^0 \to [\ \ ]^0 Z$

Control objects in the skin membrane

F1 (the first positive result sends the answer)
$[\ S\ ]^0 \to [\ \ ]^1$`yes`

F2 (without the positive result, the time-out sends the negative answer)
$[\ Z\ ]^0 \to [\ \ ]^0$`no`

Let us now explain how the system works in more details. The control objects keep track of the number of cycles completed, whether marking or unmarking takes place, as well as the number of objects already (un)marked. Moreover, the control object is responsible to pass the "right" information to the objects via polarization: in stage 1, by generating $o_{0,1}$ or $o_{5,1}$ if the variable is true, and $o_{0,2}$ or $o_{5,2}$ if the variable is false; in stage 2, $o_{0,1}$ or $o_{5,1}$ if the clause is already found, and $o_{0,2}$ or $o_{5,2}$ if the clause is being checked for.

During the first stage, an object $v_{j,i,1,s}$ is transformed into $v_{j,i,n+1,t}$, where $t = 1$ if variable x_j satisfies clause C_i, or $t = 2$ if not. The change of the last index from s to t happens when the third index is equal to i. The control object d_1 is transformed into d_{n+1}. Stage 1 takes at most $(10l+5)n$ steps (at most $(10l+3)n$ in the case when l is even).

If some clause is not satisfied, then the computation in the corresponding membrane is "stuck" with polarization 2. Otherwise, during the second stage an object $v_{j,i,n+1,t}$ is transformed into $v_{j,i,n+m+1,t}$, while the control object d_{n+1} becomes d_{m+n+1}. Stage 2 takes at most $m(10l+1)$ steps, plus one extra step to send objects S to skin, if any.

After stage 2 is completed, one copy of S, if any, is sent out as `yes`, changing the polarization of the skin membrane. After this time has passed, object Z comes to the skin from region 3. If the polarization of the skin remained 0, Z is sent out as `no`. $\square$

The rules of the system in the proof above are also *global*, so we can again obtain the following

Corollary 2 *A uniform family of confluent polarizationless P systems with rules* $(a), (c_0'), (e_0)$ *working in minimally parallel way can solve* `SAT` *with membrane labels of six kinds.*

5 Conclusions

Since changing membrane polarization controls what rules can be applied, the number of polarizations corresponds to the number of states of this control. Moreover, almost the only way the objects of the system may interact is via changing membrane polarization. Hence, the number of polarizations is a complexity measure deserving attention.

For maximal parallelism it has been proved that two polarizations are sufficient for both universality (with one membrane) and efficiency, while one-polarization systems are still universal (with elementary membrane division and membrane dissolution), but are conjectured not to be efficient.

We proved that efficient solutions of computationally hard problems by P systems with active membranes working in minimally parallel way can be constructed avoiding both cooperative rules and non-elementary membrane division, thus improving results from [7], [9]. For this task, it is enough to have four polarizations, sequential evolution rules changing polarizations, polarizationless elementary membrane division rules and polarizationless rules of sending an object out. One can use the standard evolution and send-out rules, as well as polarizationless elementary membrane division rules; in this case, six polarizations suffice.

The first construction is "almost" deterministic: the only choices the system can make in each cycle is the order in which the input systems are processed. The second construction exhibits a more asynchronous behaviour of the input objects, which, depending on the chosen degree of parallelism, might speed up obtaining the positive answer, but less than by 20% [1]. In this case, controlling polarizations by evolution is still faster than controlling polarizations by communication.

A number of interesting problems related to minimal parallelism remain open. For instance, is it possible to decrease the number of polarizations/labels? Moreover, it presents an interest to study other computational problems in the minimally-parallel setting, for instance, the computational power of P systems with one active membrane working in the minimally parallel way.

Acknowledgements The author gratefully acknowleges the support of the Japan Society for the Promotion of Science and the Grant-in-Aid for Scientific Research, project 20·08364. He also acknowledges the support by the Science and Technology Center in Ukraine, project 4032, and project TIC2003-09319-C03-01 from Rovira i Virgili University.

References

[1] A. Alhazov: Minimal Parallelism and Number of Membrane Polarizations. Preproc. of the *Seventh International Workshop on Membrane Computing*, WMC7 (H.J. Hoogeboom, Gh. Păun, G. Rozenberg, Eds.), Lorentz Center, Leiden, 2006, 74–87.

[1]The maximal total number of steps needed is slightly over $10l(m+n)$; the fastest computation happens if rules C2 are executed in parallel for all input objects, as well as rules C4, D2, D3, saving $lm-1$, $lm-1$, $ln-1$, $ln-1$ steps, respectively. Their total is $2l(m+n)-4$, which is less than (but assymptotically equal to) 1/5 of the worst time.

[2] A. Alhazov: P Systems without Multiplicities of Symbol-Objects. *Information Processing Letters* **100**, 3, 2006, 124–129.

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

[4] A. Alhazov, R. Freund: On the Efficiency of P Systems with Active Membranes and Two Polarizations. *Membrane Computing, International Workshop, WMC 2004*, Milan, Revised Selected and Invited Papers (G. Mauri, Gh. Păun, M.J. Pérez-Jiménez, G. Rozenberg, A. Salomaa, Eds.), *Lecture Notes in Computer Science* **3365**, Springer, 2005, 146–160, and *Fifth Workshop on Membrane Computing (WMC5)* (G. Mauri, Gh. Păun, C. Zandron, Eds.), University of Milano-Bicocca, Milan, 2004, 81–94.

[5] A. Alhazov, R. Freund, A. Riscos-Núñez: One and Two Polarizations, Membrane Creation and Objects Complexity in P Systems. *Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing* (SYNASC'05), IEEE Computer Society, 2005, 385–394, and First International Workshop on *Theory and Application of P Systems*, Timişoara, Romania, 2005 (G. Ciobanu, Gh. Păun, Eds.), 2005, 9–18.

[6] A. Alhazov, R. Freund, Gh. Păun: Computational Completeness of P Systems with Active Membranes and Two Polarizations. *Machines, Computations, and Universality, International Conference, MCU 2004*, Saint Petersburg, Revised Selected Papers (M. Margenstern, Ed.), *Lecture Notes in Computer Science* **3354**, Springer, 2005, 82–92.

[7] G. Ciobanu, L. Pan, Gh. Păun, M.J. Pérez-Jiménez: P Systems with Minimal Parallelism. *Theoretical Computer Science* **378**, 1, 2007, 117–130.

[8] R. Freund, S. Verlan: A Formal Framework for Static (Tissue) P Systems. *Membrane Computing, 8th International Workshop,*

WMC 2007, Thessaloniki, Revised Selected and Invited Papers (G. Eleftherakis, P. Kefalas, Gh. Păun, G. Rozenberg, A. Salomaa, Eds.), *Lecture Notes in Computer Science* **4860**, Springer, 2007, 271–284.

[9] T.O. Ishdorj: Power and Efficiency of Minimal Parallelism in Polarizationless P Systems. *J. Automata, Languages, and Combinatorics* **11**, 3, 2006, 299–320.

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

[11] Gh. Păun: *Membrane Computing. An Introduction.* Springer-Verlag, Berlin (Natural Computing Series), 2002.

[12] Gh. Păun, Y. Suzuki, H. Tanaka, T. Yokomori: On the Power of Membrane Division in P systems. *Theoretical Computer Science* **324**, 1, 2004, 61–85.

[13] M.J. Pérez-Jiménez, A. Romero-Jiménez, F. Sancho-Caparrini: Complexity Classes in Cellular Computing with Membranes. *Natural Computing* **2**, 3, 2003, 265–285.

[14] S. Verlan: *Study of Language-Theoretic Computational Paradigms Inspired by Biology.* Habilitation thesis, Universite Paris Est, Creteil Val de Marne, France, 2010.

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

Artiom Alhazov

Received November 1, 2010

Institute of Mathematics and Computer Science,
Academy of Sciences of Moldova
5 Academiei str., Chişinău, MD-2028, Moldova
E–mail: `artiom@math.md`

FCS, Department of Information Engineering,
Graduate School of Engineering
Hiroshima University,
Higashi-Hiroshima 739-8527 Japan

Computer Science Journal of Moldova, vol.**18**, no.2(53), 2010

On computational properties of gene assembly in ciliates

Vladimir Rogojin

Abstract

Gene assembly in stichotrichous ciliates happening during sexual reproduction is one of the most involved DNA manipulation processes occurring in biology. This biological process is of high interest from the computational and mathematical points of view due to its close analogy with such concepts and notions in theoretical computer science as permutation and linked list sorting and string rewriting. Studies on computational properties of gene assembly in ciliates represent a good example of interdisciplinary research contributing to both computer science and biology. We review here a number of general results related both to the development of different computational methods enhancing our understanding on the nature of gene assembly, as well as to the development of new biologically motivated computational and mathematical models and paradigms. Those paradigms contribute in particular to combinatorics, formal languages and computability theories.

1 Introduction

We survey here a number of major results which address some computational properties of evolved DNA manipulation process happening in mating cells of stichotrichous ciliates [57, 27, 37, 59, 66]. Stichotrichous ciliates belong to Domain Eukaryote, Phylum Ciliophora, Subphylum Intramacronucleata, Class Spirotrichea, Subclass Stichotrichia [46].

DNA manipulation during gene assembly in stichotrichous ciliates represents a beautiful example of a computational process happening in living organisms. Ciliates belong to one of the oldest and most diverse groups of eukaryotic cells [65]. Currently there are known around 8,000 species of ciliates [17]. Two unique features differ ciliates from other eukaryotes: nuclear dualism and possession of hairlike structures on their cellular surfaces which are called *cilia* [17, 58]. Germline and somatic nuclear functions are split between nuclei of two different types, called micro- and macronuclei respectively. Micronuclei keep their genetic data in highly encrypted form: genes are split into fragments separated by non-protein-encoding sequences, those fragments may be shuffled and some of them could be inverted [8]. In the same time macronuclear genome is organized in a very compact form: basically any macronuclear DNA represents one (rarely two) contiguous nucleotide sequences representing genes. Majority of non-stichotrichous ciliates have simpler organization of their micronuclear genome: gene fragments are still separated by non-coding sequences but follow in the orthodox order. When micronuclei get transformed into macronuclei, all non-coding blocks of DNA are being excised and gene fragments get spliced together to form contiguous sequences representing genes [57]. In stichotrichs and several other species of ciliates this process is even more involved due to the necessity to unscramble gene fragments. This process of gene assembly is especially of interest in stichotrichs from the computational point of view, since it could be interpreted formally as a string rewriting and permutation sorting procedure [61, 17].

We present briefly in this article three aspects related to studies of computational properties of gene assembly in ciliates.

First, we consider a restricted versions of the intramolecular operations of gene assembly (called simple and elementary models) which take into account only local intramolecular manipulations with DNA [22, 33, 45]. We describe a number of combinatorial properties of simple and elementary models, including the structure of gene patterns that can be assembled in these restricted models and form of their assembly strategies [32, 44, 33, 56, 63].

Secondly, we address several novel computational models based on gene assembly relying on either contextual molecular recombinations [36] or on non-deterministic sequence matching [4]. We show that some variants of the models are Turing complete, while some others may be used to solve efficiently (albeit only theoretically) computationally intractable (in particular, NP-complete) problems. Moreover, we mention the result where it was shown that the concept of distributed computations from Tissue-like P systems substitutes well the contextual ingredient of molecular operations in the computational model when demonstrating the Turing universality of gene assembly in ciliates [2].

The third aspect, which we investigate in this paper, is related to an algorithmic approach for studying a graph-theoretical notion of parallel complexity of the gene assembly in ciliates [30].

2 Biological basics of gene assembly in ciliates

As it was mentioned above, ciliates posses two unique features: all ciliates have cilia and all of them are in possession of nuclei of two different types. Cilia represent a complex of moving hair-like organelles projecting from the cellular surface. The motion of cilia is synchronized so that they propel efficiently the cell through the aqueous environment and/or direct nutritious particles (like bacteria, algae or other ciliates) into the cell's oral apparatus [17, 58].

Those nuclei that perform germline function in ciliates are called micronuclei. Micronuclei practically do not participate in RNA transcription and most of the time are passive throughout the life cycle of a cell. However, when ciliates breed, micronuclei get activated. When breeding, two ciliate organisms of the same specie exchange their micronuclear genetical information. On the other hand, almost all RNA transcription in ciliates is carried on in macronuclei [17, 59].

There is a big difference in the internal organization of micronuclear and macronuclear genome. Micronuclear DNA are organized on chromosomes. Each micronuclear DNA represents very long molecule, which contains many genes separated by long non-protein-coding spacer nucleotide sequences. Each gene is broken into some number of fragments separated from each other by non-protein-coding blocks [8]. In stichotrichs ciliates and several other species gene fragments are also shuffled throughout the molecule and some of the fragments are inverted [59, 17]. Contrary to micronuclear DNA macronuclear molecules are short and contain usually one, rarely two contiguous nucleotide protein-coding sequences [8].

The internal molecular structure of micronuclear genes suits well for robust preservation of the genetic information for future generations, while macronuclear gene structure is optimized for rapid RNA transcription, what should bring an evolutionary advantage for ciliate organisms [17].

Macronuclei from maternal organisms get disintegrated during sexual reproduction, while some copies of micronuclei from child organisms are being transformed into new macronuclei. During this transformation heavy editing of micronuclear DNA occurs, so that non-protein-coding sequences (called *Internal Eliminated Sequences*, IESs) get eliminated and new macronuclear DNA are being created from the micronuclear DNA as the result of splicing of micronuclear gene fragments (called *Macronuclear Destined Sequences*, MDSs). Thus, this DNA manipulation process is called *gene assembly* [57]. This process is particularly complex in stichotrichous ciliates because gene fragments on the micronuclear DNA are shuffled and some of the fragments are inverted [8].

The order in which MDSs should be spliced to each other to assemble a macronuclear gene is indicated by short nucleotide sequences (called *pointers*) placed on the edges of MDSs. Any two MDSs which stay next to each other in the assembled macronuclear gene share same pointer on their respective edges. In this way, pointers "tell" for each MDS to which other MDSs it should be spliced to. One can think that MDSs that follow directly one another in the macronuclear DNA "point" to each other by means of their pointers. Thus, a micronuclear gene pattern could be interpreted as a linked list data structure, and the gene assembly process could be seen as a list sorting procedure [61].

3 Molecular models for gene assembly

Two molecular models for gene assembly that suggest splicing of MDSs on their common pointers are called intermolecular [39, 42, 43] and intramolecular [20, 21]. As it follows from their names, the intermolecular model considers interaction and exchange of some pieces of DNA between several molecules during gene assembly, while the intramolecular model assumes that all manipulations are being carried on in all DNA independently from each other.

The intermolecular model

The intermolecular model considers three molecular operations [39, 42, 43]:

1. *Intramolecular recombination:* A block of DNA flanked by occurrences of same pointer gets excised in the form of a circular molecule. As the result, two molecules are produced, one is linear which contains the remaining nucleotide sequence, and another is the circular one containing the excised sequence. For details we refer to Figure 1.

2. *Intermolecular recombination:* As the inverse of the intramolecular recombination, a circular molecule gets inserted into a linear molecule if both molecules posses occurrences of the same pointer. The circular molecule is cut at the site of the occurrence of the pointer and gets inserted at the location of another occurrence of the pointer into the linear molecule. See Figure 2.

3. *Intermolecular recombination:* Two linear molecules recombine on their common pointer. As the result, two molecules interchange their tails starting on the common pointer. Note, that this operation is self-reversible, i.e., if applied on its resulting molecules on the same pointer, the initial two molecules can be obtained. See Figure 3.

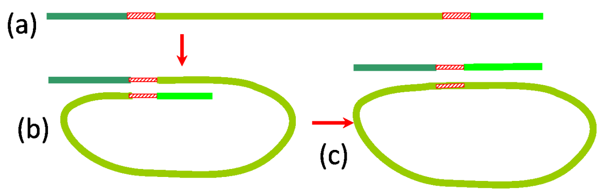


Figure 1. *Intramolecular recombination.* (a) Initial molecule: a pointer is in direct repeat. (b) Folding: the molecule folds forming a loop so that occurrences of the pointer get next to each other. (c) The result: A part of the molecule flanked by the occurrences of the pointer gets excised in the form of a circular molecule.

The intramolecular model

The intramolecular model considers three operations, called *ld*, *hi* and *dlad* [20, 21]:

1. *Loop, direct-repeat excision, ld:* This operation is applicable to a molecule having either an IES flanked by repeating occurrences of a pointer (called *simple ld*) or having a block containing all MDSs flanked by occurrences of a pointer, and this pointer occurs twice with the same orientation. The molecule folds forming a loop, so that occurrences of the pointer get next to each other. Then the recombination happens, and as the result the block flanked by occurrences of the pointer gets excised as a circular molecule, see Figure 4. Note, that this operation resembles the intramolecular recombination from the intermolecular model, except that its simple version does not excise DNA blocks containing any MDS. During gene assembly this operation is used to get rid of non-coding blocks.

2. *Hairpin, inverted-repeat excision/reinsertion, hi:* This operation is applicable to a molecule having two occurrences of the same pointer with opposite orientations. The molecule folds forming a hairpin loop, so that both occurrences of the pointer are brought next to each other and have the same spacial orienta-

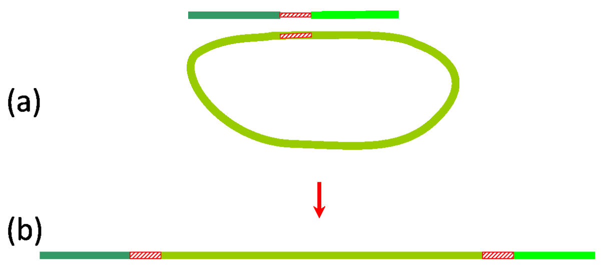


Figure 2. *Intermolecular recombination. The inverse of the intramolecular recombination* (a) Initial molecules: a circular and a linear molecule having occurrences of the same pointer (b) Result: the circular molecule gets inserted into the linear molecule at the site of occurrences of the pointer.

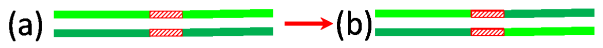


Figure 3. *Intermolecular recombination.* (a) Initially two linear molecules with occurrences of the same pointer. (b) Result: molecules exchange their tails starting at occurrences of the pointer.

tion. Then the recombination is possible, and as the result, piece of the molecule flanked by occurrences of the pointer in the inverted repeat is inverted, see Figure 5. This operation is used throughout the assembly process in order to restore the proper orientation of MDSs.

3. *Double loop, alternating direct-repeat excision-reinsertion, dlad:* This operation can be applied when the molecule has two pointers occurring in an overlapping direct repeat. I.e., when block flanked by occurrences of the same orientation of a pointer overlaps with the block flanked by occurrences of the same orientation of the other pointer. The molecule folds forming a double loop so that occurrences of both pointers get next to each other so that the recombination be possible. In the result of this recombination, non-overlapping parts of blocks flanked by their pointers

exchange their places, see Figure 6. This operation is used in the assembly in order to sort MDSs in proper order.

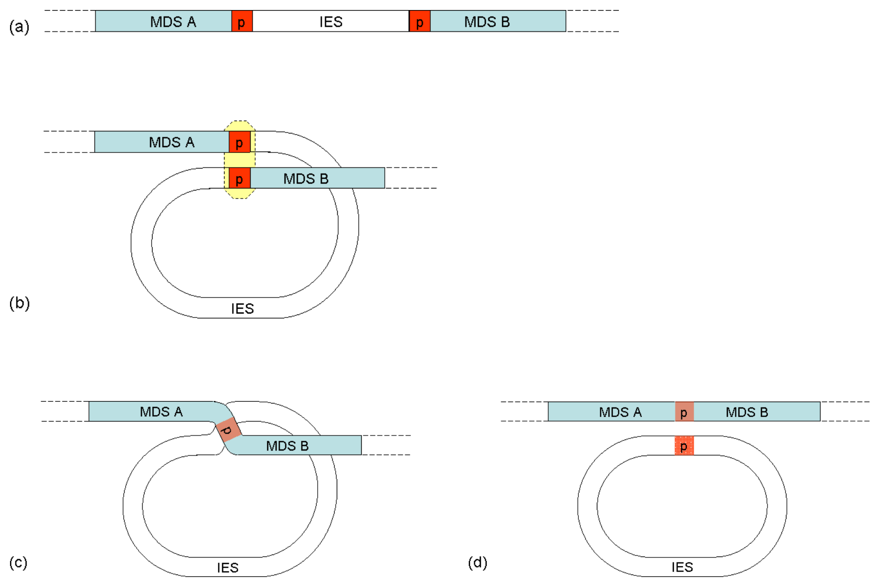


Figure 4. *Simple loop recombination [64].* (a) Initial molecule: an IES flanked by occurrences of pointer p. (b) Loop-folding, alignment of occurrences of pointer p. (c) Recombination by pointer p. (d) Result: the IES is excised in the form of circular molecule, MDS A and MDS B are spliced on the common pointer p in the linear molecule. One occurrence of p is present in the linear molecule and one occurrence of p is present in the circular molecule, but neither of them acts as a pointer.

Note, that contrary to the intermolecular model, the intramolecular operations are not reversible. Application of any of *ld*, *hi* or *dlad* reduces the number of MDSs by gluing two or more MDSs on their common pointers into bigger composite MDSs. A pointer is considered to stop acting as a pointer, when its occurrence gets either inside of an composed IES or of an composed MDS.

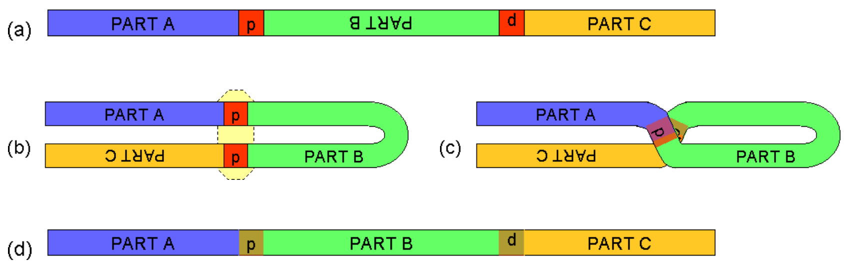


Figure 5. *Hairpin recombination [64].* (a) Initial molecule: one occurrence of p in an orthodox orientation and another in the inverted orientation. (b) Hairpin-folding, alignment of occurrences of pointer p. (c) Recombination by pointer p. (d) Resulting molecule: the orientation of *PART B* is changed, the rest of the molecule is not affected. As a result of the inversion of *PART B* MDSs having pointer p are spliced together. Occurrences of p and their orientations are retained, but neither of the occurrences acts as a pointer.

Simple and elementary intramolecular models

The intramolecular operations allow in their general formulation that DNA blocks participating in the recombination may contain any number of MDSs. However, arguing on the principle of parsimony, *simple intramolecular operations* were introduced in [22] suggesting that all the recombinations are applied "locally". Simple operations follow the same folding and recombination events as the operations in the general intramolecular model, however, blocks of DNA that are being inverted or relocated may contain exactly one MDS. Even further simplification of the intramolecular operations led to so called *elementary operations*, where only the micronuclear (non-composite) MDSs are allowed to be inverted/relocated [33]. I.e., as soon as two MDSs are combined into a composite MDS, this MDS cannot be rearranged by the elementary operations.

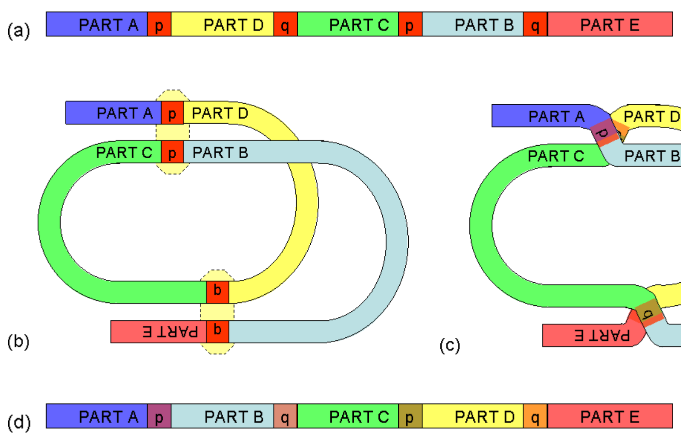


Figure 6. *Double-loop recombination [64].* (a) Initial molecule: blocks flanked by occurrences of pointers p and q respectively overlap, their common nucleotide sequence is *PART C*. (b) The molecule forms a double-loop folding so that occurrences of p and q get aligned for the recombination. (c) Recombination by pointers p and q. (d) Resulting molecule: *PART B* and *PART D* exchange places. As a result of the translocation of *PART B* and *PART D* MDSs having pointer p are spliced together, and MDSs having pointer q are spliced together. Occurrences of p and q remain in the molecule, but none of them act as a pointer.

Template-guided recombination models

Since nucleotide sequences representing pointers are very short (from 2 to 20 base-pairs) and may occur also in the middle of MDSs and IESs [6], there is a need for some kind of guiding mechanism which "helps" ciliates to identify correctly MDSs and to align "real" occurrences of the pointers next to each other and splice "real" MDSs in the right order. The template-guided recombination models suggest such mechanism [7, 60]. These models rely on the concept of templates - macronuclear DNA or RNA remained from maternal organisms. The presence of those molecules in the newly formed macronuclei and their critical role for gene assembly have been shown experimentally [49]. The template-guided recombination represents *triple-splicing*, where two recombining molecules (or different parts of the same molecule) get aligned next to each other and recombine with the help of the template molecule which being as a product of an earlier equivalent recombination, shows the way the alignment and the recombination should be done [7, 60].

In the first template-guided recombination model proposed in [60] a double-stranded DNA serves as a template. This DNA is assumed to be a copy of the assembled gene from the parental macronucleus. In this model, the template DNA is placed in-between recombining DNA. As the result of the recombination, one double stranded DNA whose nucleotide sequence contains the left part of one of the recombining molecules and the right part of the other of the recombining molecules and which matches the template is obtained. During this recombination, the recombining molecules and the template molecule exchange some of their physical parts, thus the resulting DNA contains some parts of the template, and the template is reconstructed to its original form, so that it could be reused for other recombinations. Also, the right part of the first of the recombining molecules and the left part of the second of the recombining molecules that do not much the template get excised. In this template-guided recombination model the parts of the recombining molecules that do not match the template are not spliced together after the recombination, what contradicts both the intermolecular and the intramolecular models.

A modification of the template-guided model from above was suggested in [7]. Instead of double-stranded DNA either a single- or double-stranded RNA serves as the template. During the recombination, the spacial position of the double-stranded template is ”above” the recombining molecules contrary to the previous model. No physical parts of the template get incorporated inside the resulting molecules. Parts of the recombining molecules that do not match the template are also spliced together.

4 Formalizing gene assembly

As we have mentioned above, gene assembly process can be interpreted from the computational point of view as a permutation sorting or string (or multiset of strings) rewriting process. We concentrate in this manuscript on the intramolecular model. Hereby, we present briefly here several formalisms for the intramolecular model at different levels of abstraction. We start from the permutation-based formalism [33], then we continue with MDS-descriptors [20, 21], after that we switch to double occurrence strings [18, 23] and contextual string rewriting rules [41, 47, 51], and finally we present graph-based formalization [18, 23] for the intramolecular model.

Gene assembly as sorting of permutations

The most suitable formalism to handle simple and elementary intramolecular operations is through rewriting rules for signed permutations [33, 45]. A given gene pattern could be represented by a signed permutation which indicates the order and orientation of its micronuclear MDSs. The gene assembly process itself could be interpreted as a sorting of the signed permutation.

We formalize here only the elementary intramolecular model. As the elementary operations rearrange only micronuclear MDSs, this leads to the following formalization of elementary operations:

Definition 1 *1. For each $p \geq 1$, eh_p is defined as follows:*

$$\begin{aligned}
\mathsf{eh}_p(xp\overline{(p+1)}z) &= xp(p+1)z,\\
\mathsf{eh}_p(x\overline{p}(p+1)z) &= xp(p+1)z,\\
\mathsf{eh}_p(x(p+1)\overline{p}z) &= x\overline{(p+1)}\overline{p}z,\\
\mathsf{eh}_p(x\overline{(p+1)}pz) &= x\overline{(p+1)}\overline{p}z,
\end{aligned}$$

where x, z are signed strings over Π_n. We denote $\mathsf{Eh} = \{\mathsf{eh}_p \mid 1 \leq p \leq n\}$.

2. *For each p, $2 \leq p \leq n-1$, ed_p is defined as follows:*

$$\begin{aligned}
\mathsf{ed}_p(xpy(p-1)(p+1)z) &= xy(p-1)p(p+1)z,\\
\mathsf{ed}_p(x(p-1)(p+1)ypz) &= x(p-1)p(p+1)yz,\\
\mathsf{ed}_p(x\overline{p}y\overline{(p+1)}\,\overline{(p-1)}z) &= xy\overline{(p+1)}\,\overline{p}\overline{(p-1)}z,\\
\mathsf{ed}_p(x\overline{(p+1)}\,\overline{(p-1)}y\overline{p}z) &= x\overline{(p+1)}\,\overline{p}\,\overline{(p-1)}yz,
\end{aligned}$$

where x, y, z are signed strings over Π_n. We denote $\mathsf{Ed} = \{\mathsf{ed}_p \mid 1 < p < n\}$.

Since the permutation-based formalism captures only the MDS inversion and relocation events throughout the gene assembly, *LD* operation is not being formalized because it neither inverts, nor relocates MDSs [64].

Example 1 *[33]*

(i) Permutation $\pi_1 = \overline{4}\,\overline{5}\,6\,\overline{1}\,2\,\overline{3}$ is sortable and a sorting composition is $(\mathsf{eh}_4 \circ \mathsf{eh}_5 \circ \mathsf{eh}_2 \circ \mathsf{eh}_1)(\pi_1) = 4\,5\,6\,1\,2\,3$. Permutation $\pi_1' = \overline{4}\,\overline{5}\,6\,\overline{1}\,\overline{2}\,\overline{3}$ is unsortable. Indeed, only $\mathsf{eh}_4 \circ \mathsf{eh}_5$ is applicable to π_1', but it does not sort it.

(ii) There exist permutations with several sorting compositions, even leading to different (cyclically) sorted permutations. One such permutation is $\pi_2 = 2\,4\,1\,3$. Indeed, $\mathsf{ed}_2(\pi_2) = 4\,1\,2\,3$. At the same time, $\mathsf{ed}_3(\pi_2) = 2\,3\,4\,1$.

(iii) There are permutations having both sorting compositions and non-sorting compositions leading to unsortable permutations. If $\pi_3 = 2\,4\,1\,3\,5$, then $\mathsf{ed}_3(\pi_3) = 2\,3\,4\,1\,5$ is a unsortable permutation. However, π_3 can be sorted, e.g., by the following composition: $(\mathsf{ed}_4 \circ \mathsf{ed}_2)(\pi_3) = 1\,2\,3\,4\,5$.

(iv) Applying a cyclic shift to a permutation may render it unsortable. Indeed, permutation $2\,1\,3$ is sortable, while $3\,2\,1$ is not.

Gene assembly as MDS descriptor rewriting process

Here we represent a gene pattern through its sequence of MDSs. Each MDS we represent through its incoming and outgoing pointers (opening and closing pointers respectively), as well as through the sequence of pointers incorporated in the MDS on which micronuclear MDSs spliced to form this composite MDS [20, 21, 4]. Formally, let $\Sigma_P = \{p_1, p_2, \ldots, p_n\}$ be the set of pointers, and $b, e \in \Sigma_P$ be so called *begin* and *end* markers (denoting opening sequence of very first MDS and closing sequence of very last MDS respectively). Then we represent an MDS by a triple $M = (p, u, q)$, where $p \in \Sigma_P \cup \{b\}$, $q \in \Sigma_P \cup \{e\}$ are called *active pointers* and $u \in \Sigma_P^*$ is called the *content*. It is said, that p is an *incoming* and q is an *outgoing* pointer of M. The length of M is denoted as $|M| = |puq|$. We denote the set of all MDSs over Σ_P as $\Sigma_M = \{(p, u, q) | p \in \Sigma_P \cup \{b\}, q \in \Sigma_P \cup \{e\}, u \in \Sigma_P^*\}$. The inversion of MDS $M = (p, u, q)$ we denote as $\overline{M} = (\overline{q}, \overline{u}, \overline{p})$ and set of inverted MDSs as $\overline{\Sigma_M} = \{\overline{M} | M \in \Sigma_M\}$.

The gene pattern we represent by its MDS descriptor – a sequence of MDSs and their orientations. Formally, an MDS descriptor is a string over alphabet $\Sigma_M \cup \overline{\Sigma_M}$.

Note, that the assembled gene is represented by any of the composite MDSs $(b, p_1p_2 \ldots p_n, e)$ and $(\overline{e}, \overline{p_n} \ldots \overline{p_2}\overline{p_1}, \overline{b})$. In this way, a gene assembly process may be interpreted as an MDS descriptor rewriting process that leads to any of the two MDSs from above.

The intramolecular operations we formalize on MDS descriptors as follows:

1. *ld* operation on pointer p is formalized as ld_p:

$$\psi_1(q, u, p)\psi_2(p, v, r)\psi_3 \Rightarrow^{\mathsf{ld}_p} \psi_1(q, upv, r)\psi_3$$

2. *hi* operation on pointer p is formalized as hi_p:
   - $\psi_1(p, u, q)\psi_2(\overline{p}, \overline{v}, \overline{r})\psi_3 \rightarrow^{\mathsf{hi}_p} \psi_1\overline{\psi_2}(\overline{q}, \overline{u}\ \overline{p}\ \overline{v}, \overline{r})\psi_3$;
   - $\psi_1(q, u, p)\psi_2(\overline{r}, \overline{v}, \overline{p})\psi_3 \rightarrow^{\mathsf{hi}_p} \psi_1(q, upv, r)\overline{\psi_2}\psi_3$;

3. *dlad* operation on pointers p and q is formalized as $dlad_{p,q}$:

- $\psi_1(p,u_1,r_1)\psi_2(q,u_2,r_2)\psi_3(r_3,u_3,p)\psi_4(r_4,u_4,q)\psi_5 \Rightarrow^{\mathsf{dlad}_{p,q}}$ $\psi_1\psi_4(r_4,u_4qu_2,r_2)\psi_3(r_3,u_3pu_1,r_1)\psi_2\psi_5$;
- $\psi_1(p,u_1,r_1)\psi_2(r_2,u_2,q)\psi_3(r_3,u_3,p)\psi_4(q,u_4,r_4)\psi_5 \Rightarrow^{\mathsf{dlad}_{p,q}}$ $\psi_1\psi_4\psi_3(r_3,u_3pu_1,r_1)\psi_2(r_2,u_2qu_4,r_4)\psi_5$;
- $\psi_1(r_1,u_1,p)\psi_2(q,u_2,r_2)\psi_3(p,u_3,r_3)\psi_4(r_4,u_4,q)\psi_5 \Rightarrow^{\mathsf{dlad}_{p,q}}$ $\psi_1(r_1,u_1pu_3,r_3)\psi_4(r_4,u_4qu_2,r_2)\psi_3\psi_2\psi_5$;
- $\psi_1(r_1,u_1,p)\psi_2(r_2,u_2,q)\psi_3(p,u_3,r_3)\psi_4(q,u_4,r_4)\psi_5 \Rightarrow^{\mathsf{dlad}_{p,q}}$ $\psi_1(r_1,u_1pu_3,r_3)\psi_4\psi_3\psi_2(r_2,u_2qu_4,r_4)\psi_5$;
- $\psi_1(p,u_1,r_1)\psi_2(q,u_2,p)\psi_4(r_4,u_4,q)\psi_5 \Rightarrow^{\mathsf{dlad}_{p,q}}$ $\psi_1\psi_4(r_4,u_4qu_2pu_1,r_1)\psi_2\psi_5$;
- $\psi_1(p,u_1,q)\psi_3(r_3,u_3,p)\psi_4(q,u_4,r_4)\psi_5 \Rightarrow^{\mathsf{dlad}_{p,q}}$ $\psi_1\psi_4\psi_3(r_3,u_3pu_1qu_4,r_4)\psi_5$;
- $\psi_1(r_1,u_1,p)\psi_2(q,u_2,r_2)\psi_3(p,u_3,q)\psi_5 \Rightarrow^{\mathsf{dlad}_{p,q}}$ $\psi_1(r_1,u_1pu_3qu_2,r_2)\psi_3\psi_2\psi_5$;

For more details on this formalism we refer to [20, 21, 4].

Example 2 *[64]*
Let us consider a micronuclear gene pattern of the actin I gene from Stylonychia lemnae*. Its MDS descriptor is*

$$\delta = (3,4)(4,5)(5,6)(7,8)(\overline{3},\overline{2})(b,2)(6,7)(8,e).$$

Composition $\Phi = \mathsf{ld}_5 \circ \mathsf{ld}_4 \circ \mathsf{hi}_{\overline{7}} \circ \mathsf{hi}_{\overline{2}} \circ \mathsf{hi}_3 \circ \mathsf{dlad}_{6,8}$ *reduces* δ *to MDS* $(b,234567,e)$*. Indeed,*

$$\begin{aligned}
\delta' = \mathsf{dlad}_{6,8}(\delta) &= (3,4)(4,5)(5,6,7)(\overline{3},\overline{2})(b,2)(7,8,e)\\
\delta'' = \mathsf{hi}_3(\delta') &= (\overline{7},\overline{6},\overline{5})(\overline{5},\overline{4})(\overline{4},\overline{3},\overline{2})(b,2)(7,8,e)\\
\delta''' = \mathsf{hi}_{\overline{2}}(\delta'') &= (\overline{7},\overline{6},\overline{5})(\overline{5},\overline{4})(\overline{4},\overline{32},\overline{b})(7,e)\\
\delta^{iv} = \mathsf{hi}_{\overline{7}}(\delta''') &= (b,23,4)(4,5)(5,67,e)\\
\delta^{v} = \mathsf{ld}_4(\delta^{iv}) &= (b,234,5)(5,67,e)\\
\delta^{vi} = \mathsf{ld}_5(\delta^{v}) &= (b,234567,e)
\end{aligned}$$

Gene assembly by contextual intramolecular operations on double occurrence strings

If we abstract from the information about MDSs and concentrate only on pointers occurrences in a gene pattern, then we obtain a string based formalism, where each letter and its sign represent an occurrence of a pointer and its orientation [23]. If we follow the idea that pointer alignment during gene assembly is context-guided, then we may come to the similar concept of contextual intramolecular operations [36].

Formally, set of templates may be represented by so-called *splicing scheme* which is a set of *splicing relations.* A splicing rule is represented by a pair of triplets $(\alpha, p, \beta) \sim (\alpha', p, \beta')$ which means that the recombination at pointer p is possible if and only if, one of its occurrences is flanked by sequences α and β and another of its occurrences is flanked by α' and β'.

The contextual *ld* on pointer p is formalized as del_p: $del_p(xpupy) = xpy$ with respect to splicing scheme R if and only if there is a splicing relation $(\alpha, p, \beta) \sim (\alpha', p, \beta')$ in R such that $x = x'\alpha$, $u = \beta u' = u''\alpha'$ and $y = \beta' y'$.

The contextual *dlad* on pointers p and q is formalized as $trl_{p,q}$: $trl_{p,q}(xpuqypvqz) = xpvqypuqz$ with respect to splicing scheme R if and only if there are splicing relations $(\alpha, p, \beta) \sim (\alpha', p, \beta')$ and $(\gamma, q, \delta) \sim (\gamma', q, \delta')$ in R such that $x = x'\alpha$, $uqy = \beta u' = u''\alpha'$, $vqz = \beta' v'$, $xpu = x''\gamma$, $ypv = \beta y' = y''\gamma'$ and $z = \delta' z'$.

Formalizing contextual *hi* is beyond our scope in this manuscript.

For non-contextual formalizations of *ld*, *hi* and *dlad* we refer to [18, 23].

Gene assembly as graph reduction process

If we abstract from positions of pointers in a gene pattern and concentrate only on their overlapping relations, then we obtain graph-based formalism for gene assembly [18, 23]. We recall that two pointers p and q overlap if and only if the interval flanked by pointer p overlaps (but does not include and is not included) with interval delimited by pointer q. I.e., we say that p and q overlap if in the gene pattern we have either scattered subsequence $pqpq$ or $qpqp$.

Formally, let u be a string representing occurrences of pointers in a gene pattern. We define for it signed overlap graph $G = (V, E, \sigma)$ as follows:

- $V = dom(u)$, i.e., each node p from G corresponds to a pointer p from u;
- $E = \{\{p, q\} | pqpq \leq_s u \text{ or } qpqp \leq_s u\}$, i.e., $\{p, q\}$ is an undirected edge in G if and only if pointers p and q overlap in u;
- $\sigma : V \to \{+, -\}$, where $\sigma(p) = -$ if and only if both occurrences of p have the same sign (i.e., the same orientation) in u.

Operations LD, HI and $DLAD$ are represented on the abstraction level of signed overlap graphs as follows:
Let $G = (V, E, \sigma)$ be a signed overlap graph:

- Operation ld on pointer p is formalized as graph reduction operation gnr: the operation gnr is applicable to a vertex $p \in V$ if $\sigma(p) = -$ and $N(p) = \emptyset$. In this case, $\mathsf{gnr}_p(G) = G - \{p\}$.
- Operation hi on pointer p is formalized as graph reduction operation gpr: the operation gpr is applicable to vertex $p \in V$ if $\sigma(p) = +$. In this case, $\mathsf{gpr}_p(G) = loc_p(G) - \{p\}$, where $loc_p(G) = (V, E, \sigma')$ with $\sigma'(x) = -\sigma(x)$ if x is a neighbor of p, otherwise $\sigma'(x) = \sigma(x)$, for all $x \in V$. Here $-\sigma(x) = -$ if and only if $\sigma(x) = +$.
- Operation $dlad$ on pointers p and q is formalized as graph reduction operation gdr: The operation $\mathsf{gdr}_{p,q}$ is applicable to adjacent vertices $p, q \in V$ if $\sigma(p) = \sigma(q) = -$. In this case, $\mathsf{gdr}_{p,q}(G) = G'$ where $G' = (V \setminus \{p, q\}, E', \sigma)$. Here for all pairs of x and y from $V \setminus \{p, q\}$ such that $x \in N_G(p) \setminus N_G(q)$ and $y \in N_G(q) \setminus N_G(p)$ we have edge $\{x, y\} \in E'$ if and only if x and y are not neighbors in G. For all other pairs x, y from $V \setminus \{p, q\}$ we have $\{x, y\} \in E'$ if and only if $\{x, y\} \in E$.

Example 3 *[64]*

Here we show how an overlap graph G which represent the micronuclear gene pattern of the actin I gene from Stylonychia lemnae, *can be reduced to the empty graph:*

Step 1: *Vertices* 6 *and* 8 *are negative and adjacent in G. In this way, we can apply* $\mathsf{gdr}_{6,8}$. *The resulting graph* $\mathsf{gdr}_{6,8}(G)$ *is represented in Figure 7(a);*

Step 2: *Vertex* 3 *is positive in* $\mathsf{gdr}_{6,8}(G)$. *Then, we can apply* gpr_3. *Vertex* 7 *changes its sign since it is adjacent to* 3. *Then we obtain graph represented in Figure 7(b);*

Step 3: *Vertices* 2 *and* 7 *are positive in* $\mathsf{gpr}_3(\mathsf{gdr}_{6,8}(G))$. *Then we can apply* gpr_2 *and* gpr_7. *The resulting graph is represented in Figure 7(c);*

Step 4: *Finally, only two negative isolated vertices* 4 *and* 5 *remain in* $(\mathsf{gpr}_7\circ \circ\mathsf{gpr}_2 \circ \mathsf{gpr}_3 \circ \mathsf{gdr}_{6,8})(G)$. *By applying* gnr_4 *and* gnr_5 *we reduce the graph to the empty one.*

5 Complexity of gene assembly

Intuitively, by "complexity" of gene assembly one can understand the "effort" needed to assemble a gene. In literature exist several concepts for the complexity of gene assembly related to types of molecular operations involved into the process as well as to the order and the manner in which those operations are applied [64]. In this section we address such gene assembly complexity related topics as simple [22] and elementary [33] gene assembly as well as the parallel complexity of gene assembly [30, 28].

Simple and elementary gene assembly

In its general formulation, intramolecular operations may invert and translocate blocks of DNA containing any number of MDSs. It has

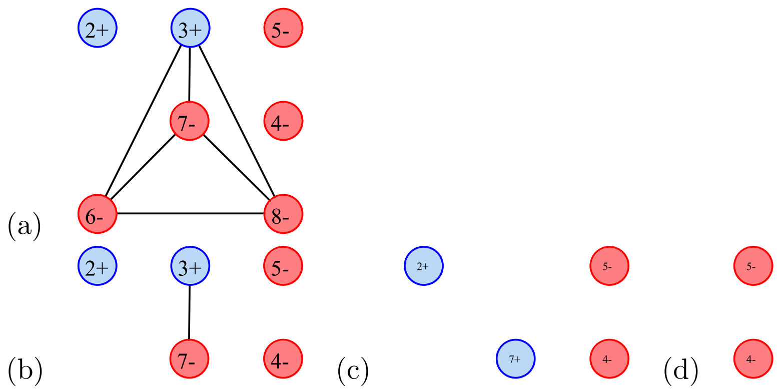


Figure 7. Graph reduction strategy [64]: (a) Graph G which represents the micronuclear gene pattern of the actin I gene from *Stylonychia lemnae*, (b) Graph $\mathsf{gdr}_{6,8}(G)$, (c) graph $\mathsf{gpr}_3(\mathsf{gdr}_{6,8}(G))$, (d) graph $(\mathsf{gpr}_7 \circ \mathsf{gpr}_2 \circ \mathsf{gpr}_3 \circ \mathsf{gdr}_{6,8})(G)$.

been shown that the intramolecular model is complete in a sense that any given hypothetical micronuclear gene pattern can be assembled to the macronuclear gene [17]. The restriction from general intramolecular to simple model has one immediate consequence, simple model is not complete. I.e., there are some hypothetical gene patterns for which there is no assembly strategy consisting of solely simple operations which lead to assembled macronuclear gene. However, simple operations can assemble all currently discovered from ciliates gene patterns [17, 22]. This fact enables the hypothesis that ciliates use simple operations to assemble their genes. Then it might be interesting to characterize those gene patterns that can be assembled by simple operations.

There may exist many different intramolecular assembly strategies applicable to the same micronuclear gene pattern [17]. In general intramolecular model, any assembly strategy applicable to a gene pattern leads to an assembled gene. However, since simple model is not complete, there are gene patterns for which there is no simple assembly strategy leading to the assembled gene. Interestingly, any gene pattern which can be assembled by simple operations to the macronuclear gene has only successful simple strategies (i.e., those leading to the macronuclear gene) [44]. In this way, characterizing those gene patterns that can be assembled by simple strategies is straightforward: just try a simple assembly strategy and see if it assembles successfully a gene pattern to the macronuclear gene. However, the situation is totaly different for elementary operations.

The slight difference between the definitions of simple and elementary operations (elementary *hi* and *dlad* invert/relocate blocks of DNA containing only one non-composite MDS) generates the following important outcome: a gene pattern may have both successful and unsuccessful applicable elementary assembly strategies [33]. This means that characterization of those gene patterns that can be assembled by elementary operations is not as straightforward as it is in the case of simple operations. It may be necessary to try some number of different elementary assembly strategies applicable to a gene pattern before finding a successful one. However, we have found an efficient combinatorial procedure to decide whether a gene pattern may be assembled by elementary operations without actually trying any of elementary assembly strategies [56, 63].

Our decision method is basing on the concept of a *dependency graph* associated to a gene pattern [33]. The dependency graph is a directed graph reflecting the information about the order in which operations can be used in strategies applicable to the gene pattern as well as about those operations that are never used in any strategy for this pattern. In this manuscript we present results addressing gene patterns without inverted MDSs. For characterization of gene patterns with the inverted MDSs we refer to [33].

As it was mentioned above, signed permutations is a suitable formalism to represent gene patterns when working with elementary operations. The dependency graph associated to permutation π is defined as $\Gamma_\pi = (V_\pi, E_\pi)$, where $V_\pi = dom(\pi)$ and

$$E_\pi = \{(1,1),(n,n)\} \cup \{(i,i)|(i+1)(i-1) \leq_s \pi\} \cup$$

$$\cup\{(j,i) | (i-1)j(i+1) \leq_s \pi\}.$$

Also, we denote the subgraph induced from Γ_π by set $T_{\pi,p} = \{q |$there is a path from q to p in $\Gamma_\pi\}$ as $\Gamma_{\pi,p}$.

Example 4 *[64]*
Let $\pi = 1\,10\,3\,5\,7\,12\,2\,9\,4\,11\,6\,13\,8$. *Then*

$$V_\pi = \mathsf{dom}(\pi) = \{1,2,3,4,5,6,7,8,9,10,11,12,13\}$$

and

$E_\pi =$	$\{$	
	$(1,1)$	
	$(10,2)$,	*since* $1\,10\,3 \leq_s \pi$
	$(9,3)$,	*since* $2\,9\,4 \leq_s \pi$
	$(11,5)$,	*since* $4\,11\,6 \leq_s \pi$
	$(13,7)$,	*since* $6\,13\,8 \leq_s \pi$
	$(2,8)$, $(12,8)$	*since* $12\,12\,2\,9 \leq_s \pi$
	$(9,9)$,	*since* $10\,8 \leq_s \pi$
	$(4,10)$,	*since* $9\,4\,11 \leq_s \pi$
	$(3,11)$, $(5,11)$, $(7,11)$,	*since* $10\,3\,5\,7\,12 \leq_s \pi$
	$(6,12)$,	*since* $11\,6\,13 \leq_s \pi$
	$(13,13)$,	*since* $n = 13$
	$\}$	

The corresponding dependency graph $\Gamma_\pi = (V_\pi, E_\pi)$ *is shown in Figure 8.*

Since we consider here gene patterns without inverted MDSs, all gene assembly strategies that we address do not use *hi* operations. I.e., formally we consider Ed-sortable permutations. For dependency graph-based characterization of Eh, Ed-sortable permutations we refer to [33].

Here we present a constructive dependency graph-based characterization of Ed-sortable permutations from [56]. We use the notion of so-called "forbidden integers" for a permutation, integers on which ed

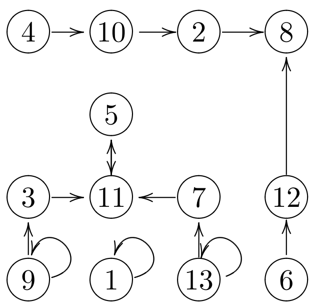


Figure 8. The dependency graph $\Gamma_\pi = (V_\pi, E_\pi)$ of permutation $\pi = 1\,10\,3\,5\,7\,12\,2\,9\,4\,11\,6\,13\,8$ [64]

operations are never used in any ed-strategy applicable to the permutation. I.e., an integer p from $\mathsf{dom}\pi$ for some unsigned permutation π we call forbidden if and only if there is no Ed strategy applicable to π where operation ed_p is used.

We can decide the set of forbidden integers for a permutation π as follows:

Theorem 1 *[56] For a permutation π over Σ_n and $p \in \Sigma_n$, p is forbidden in π if and only if the subgraph $\Gamma_{\pi,p} = (T_{\pi,p}, E_{\pi,p})$ is cyclic or $q-1, q \in T_{\pi,p}$ for some q.*

By $\pi|_U$ we denote a scattered substring of π containing only the letters from set U.

Now, we decide the Ed-sortability of a permutation π as follows:

Theorem 2 *[56]*

Permutation π is sortable if and only if $\pi|_{F(\pi)}$ is sorted.

Example 5 *[64]*

Let π be the permutation from Example 4. By Theorem 1, set $F(\pi) = \{1,3,5,7,9,11,13\}$, since 1, 9 and 13 are in self-loops, 5 and 11 form a cycle, and edges $(9,3)$ and $(13,7)$ belong to Γ_π. Clearly $\pi|_{F(\pi)} = 1\,3\,5\,7\,9\,11\,13$, which is sorted. Thus π is sortable. For instance, composition of ed *operations $\mathsf{ed}_8 \circ \mathsf{ed}_{12} \circ \mathsf{ed}_6 \circ \mathsf{ed}_2 \circ \mathsf{ed}_{10} \circ \mathsf{ed}_4(\pi)$ sorts π.*

This theorem does not provide any information about those ed strategies that sort π. The general form of all sorting strategies was presented in [63].

Parallel gene assembly

By parallel complexity of a gene pattern we understand the minimal number of parallel steps needed to assemble a gene. Formally, the parallelism of gene assembly is analyzed by means of the pointer reduction system. We say that a set of graph reduction operations may be applied in parallel to a graph if and only if these operations may be applied in any order to the graph. It was shown that for an initial graph the set of reduction operations, but not the order of their application determine the resulting graph. All these enables the notion of *parallel reduction* of a graph as a sequence of *parallel steps* leading to the empty graph [30, 28]. Here a parallel step is a set of operations applied in parallel. We represent a parallel reduction formally as $\Phi = S_k \circ \ldots \circ S_1$, where S_i is a set of operations applicable in parallel to graph $S_{i-1} \circ \ldots \circ S_1(G)$ and $\Phi(G)$ is an empty graph. The parallel complexity of Φ is k, we denote it as $\mathcal{C}(\Phi) = k$. By parallel complexity of graph G we understand the minimal complexity among all of its parallel reductions. Formally, the parallel complexity of G is $\mathcal{C}(G) = \min\{\mathcal{C}(R) \mid R \text{ is a parallel reduction of } G\}$.

One of the main open questions related to the topic of parallel gene assembly is whether *"the common finite upper bound for parallel complexities of all signed overlap graphs exists"*. This question was answered only for some particular types of graphs in [28, 31, 30]. The highest parallel complexity for negative trees is 2, and for positive trees is 3. The upper bound for arbitrarily signed trees is unknown. The parallel complexity of negative paths with $2n$ vertices is 1, and of negative paths with $2n + 1$ vertices is 2. The parallel complexity of any path with either $3n$ or $3n + 1$ vertices is 2, and of any positive path with $3n + 2$ vertices is 3. Any positive complete bipartite or tripartite graph has a parallel complexity of at most 3. Upper bounds for parallel complexities of some other types of graphs are also known.

Another open problem for the topic of parallel gene assembly is to find an efficient algorithm to decide the parallel complexity of a signed graph. Currently, the most optimal known algorithm computing the parallel complexity of a graph [5] has time complexity

$$O\left(\frac{n^{n+7/2}}{c^n}\right) \text{ for } c = \frac{e}{\sqrt{2}}.$$

This is an improvement in comparison to the basic brute force algorithm [3] where all applicable parallel strategies are being checked. In this improved algorithm we are considering parallelization of sequential graph reduction strategies. Any sequential strategy that we consider is split into parallel steps in such a manner that the number of parallel steps is minimal for this strategy. Moreover, we do not consider more than one sequential strategy having the same parallelization. This approach enables us to consider instead of many parallel reduction strategies with the same domain of operations just one of them with the lowest complexity. Moreover, we have improved also on the parallel applicability decision procedure. Instead of checking the applicability of all permutations of operations from a set, we have used another approach. We based on the fact that a set of operations S is applicable in parallel to a graph G if and only if for any subset $S' \subseteq S$ sequence of operations $r \circ lex(S')$ is applicable to G, where $r \in S \setminus S'$ and $lex(S')$ is the lexicographical order of operations from S'. In this way, we have to check $k2^{k-1}$ sequences of operations $r \circ lex(S')$ to decide the parallel applicability of S. The complexity estimate of the basic algorithm in [3] grows almost as fast as $(n^n)^2$, while the present estimate of the improved algorithm in [5] grows almost as fast as n^n.

6 Computing with gene assembly

In this section we concentrate on the topic of computing by gene assembly in ciliates. Inspired by the celebrated computational experiment of Adleman with DNA [1], we are interested whether and how we can compute by using evolved DNA manipulation during the gene assembly.

The research in this direction addresses such formal language-based topics as computability, hierarchies of classes, language equations, closure properties, as well as the results on developing methods to solve computationally hard mathematical problems.

Formal languages-related results

One of the first results concerning the computability of gene assembly was obtained for the intermolecular model [39]. The concept of contextual recombinations [54] was used to simulate computations by Turing machines by using intermolecular operations. Basing on contextual version of intermolecular operations an accepting system was defined: a multiset of strings is accepted if in result of application of a sequence of contextual intermolecular operations according to the given splicing scheme [26] a multiset containing the given axiom word is obtained.

Inspired by this result we have shown in a similar manner the Turing universality for the contextual intramolecular operations [36]. Since the intramolecular model operates on a single molecule, we used the formalism of contextual string rewriting rules representing intramolecular operations. Basing on these contextual string rewriting rules we have defined an accepting intramolecular recombination system incorporating the following ingredients: the *splicing scheme*, the *start word* and the *target word*. That system accepts all those words which being concatenated to the start word produce the target word in the result of a sequence of application of contextual string rewriting rules according to the given splicing scheme. Formally, an accepting intramolecular recombination system is denoted as $G = (\Sigma, \sim, \alpha_0, w_t)$, where $(\Sigma, \sim)$ is the splicing scheme, α_0 is the start word and w_t is the target word. G accepts the following language: $L(G) = \{w \in \Sigma^* | \alpha_0 w \Rightarrow^*_{\tilde{R}} w_t\}$. Accepting intramolecular recombination systems were proved to be universal. Instead of using multiple copies of a string as in the case of the intermolecular model we used concatenation of multiple copies of the string. For any simulation of a Turing machine we assumed that we have as many concatenations of copies of the string as needed.

It was shown in [2] that the "contextual ingredient" of the intermolecular model if substituted by the "distribution ingredient" does not decrease the computational power of gene assembly. In [2] there were introduced tissue P systems with ciliate operations. In general, a tissue P system is represented by an undirected graph, where multisets of objects and sets of multiset rewriting (evolution) rules are associated to each node (called *region*, or *membrane*, or *cell*). Rules define the evolution of multisets and communication between neighboring regions through *communication channels* (the graph edges) [48]. The paradigm of P systems was motivated from such biological elements as cellular membranes and membranal structure, biochemical reactions, DNA and RNA manipulations, transmembrane transportation of chemicals and other phenomena in cellular biology [52, 53, 48]. We refer to [53, 62] for the detailed overview on the research topic of P systems. In its generic definition, P systems posses an abstract nature of their objects and evolution rules. In tissue P systems with ciliate operations the objects are strings representing DNA molecules and evolution rules are intermolecular operations with transmembrane communication. It is added to the definition of the intramolecular excision and the intermolecular insertion, the information about the regions-targets for the excised circular molecule and the molecule containing non-excised sequences, as well as the information about regions-sources for the recombining molecules.

Formally, a tissue P system with ciliate operations is defined as a tuple $\Pi = (O, C, R, i_0)$, where O is a finite set of symbols, C is a finite set of regions, R is a finite set of excision/insertion rules and i_0 is the output region. To each region $c \in C$ there are associated finite sets of strings $inf(c)$ presented in infinite number of copies in c and initial finite multiset of strings $cfg(c)$. The strings can be both linear and circular and are defined over alphabet O. Each evolution rule from R has one of the following forms: *intramolecular excision rule* $i \rightarrow_p j/k$ is applicable on a string $upvpw$ (or $\circ upvpw$) in region $i \in C$ and produces strings upw (or $\circ upw$ respectively) and $\circ pv$ in region k; *intermolecular insertion rule* $j/k \rightarrow_p i$ is applicable on pair of strings upw (or $\circ upw$) in region j and $\circ pv$ in region k and produces string $upvpw$ (or $\circ upvpw$ respectively) in region i. Here p is a pointer and u, v, w are linear strings over O. All rules from R are applied in parallel either synchronously or non-synchronously and either in maximally parallel or non-maximally parallel manner (depending on the type of the system). If in result of application of rules from R the system reaches a state where no rule from R could be used, then the multiset of words (or the number of words) in region i_0 is considered to be the *result of computation* of Π. Registers machines were chosen to be simulated by the tissue P systems with ciliate operations in order to prove the P systems Turing universality.

Among other results on gene assembly based on formal languages we can mention language operations inspired by molecular gene assembly operations and their closure properties as well as solutions of language equations [9, 12, 13, 24], language operations inspired by the template-guided DNA recombination [15, 16], generalization of intra- and intermolecular operations as synchronized insertion/deletion on linear strings [9], generalized versions of *ld* and *dlad* operations and families of languages defined by closure under those operations [10, 11].

Computing NP-complete problems

We are wondering how we can use gene assembly in order to solve computationally intractable problems. One of the advantages of gene assembly over DNA computing from the point of view of the experimental implementation might be the fact that such actions as amplification (producing a high number of clones of a DNA molecule) and filtering on the DNA molecules of interest which should be performed "manually" in the lab might be executed by ciliates "autonomously".

In this section we survey research results related to the development of computational methods for solving NP-complete problems [50] by means of gene assembly process. Generally, the gene assembly process is deterministic and confluent, i.e., starting from a gene pattern it always assembles the macronuclear gene (or in case of simple intramolecular operations all assembly strategies applicable to the gene pattern either lead to the assembled gene, or all of the strategies fail). The basic approach for solving instances of NP-complete problems is in checking all the particular solutions for the instance of an NP-complete problem. In order to implement this approach by means of gene assembly, one has to make the gene assembly process non-deterministic and non-confluent, i.e., while starting from the same gene pattern different assembly strategies should produce different resulting molecules. This is achievable if allowing gene patterns with more than two occurrences of the same pointer (and even with several copies of the same MDS) [4]. Then, the computational method for solving NP-complete problems is looking as follows:

1. Encode an instance of the problem as a gene pattern;
2. Let the ciliate amplify the molecule with the encoded instance;
3. Let the gene assembly occur so that all copies of the molecule get assembled by different assembly strategies corresponding to different particular solutions to different resulting molecules;
4. Interpret the resulting molecules as the results of the corresponding particular solutions of the instance.

In [4] we have presented a computational method to solve instances of Hamiltonian Path Problem (HPP) inspired by the famous Adleman's experiment with DNA [1]. For a directed graph $G = (V, E)$ with n vertices and m edges a *hamiltonian path* in G from vertex p to vertex q is a noncyclic path from p to q containing all the vertices from G. The HPP is defined as the problem of deciding whether there exists a hamiltonian path from p to q.

Adleman has solved a small instance of HPP through an experimental assay on its DNA encoding. He implemented the following steps by using biotechnological tools:

Step 1: Generating random paths of the graph;

Step 2: Filtering set of paths generated at *Step 1* so that only paths from p to q remain;

Step 3: Filtering set of paths generated at *Step 2* and living only paths of exactly length n;

Step 4: Filtering set of paths generated at *Step 3* and retaining just paths containing all the vertices of the graph;

Step 5: The paths remaining after *Step 4* are the hamiltonian paths.

The instance of HPP problem was encoded as follows: for each vertex i of the graph there was designed a short single strand DNA sequence O_i of length 20 bp. An edge between vertices i and j was represented by a single strand molecule O_{ij} where the prefix O_{ij} of length 10 was the suffix of O_i, and the suffix of O_{ij} was the prefix of O_j. In his experimental asset, Adleman used a number of copies of molecules O_{ij} for each edge $(i, j) \in E$ and a number of copies $\overline{O_i}$ of complementary strands to O_i for each vertex $i \in V$.

Then, *Step 1* was implemented as follows. All the molecules O_{ij} were let to splice to the complementary sites at molecules $\overline{O_i}$ and $\overline{O_j}$ in a random way. In this way, two molecules O_{ij} and O_{jk} were attached to each other by means of molecule $\overline{O_j}$ and a double-stranded molecule representing path ijk from the graph G was produced. In this manner there were produced double-stranded DNA representing a number of paths from graph G. It was assumed that there were enough number of copies of vertex- and edge-representing molecules provided in the initial assay so that the probability of generating hamiltonian paths was sufficiently high.

Then, *Step 2* was implemented by *polymerase chain reaction*, *Step 3* by gel electrophoresis, and *Step 4* was performed by using a *biotin avidin magnetic beads system* in order to select molecules containing nucleotide sequences O_i for all $i \in V$. In this way, in result of the experiment only the molecules representing hamiltonian paths were obtained.

In this way, Adleman has demonstrated that in principle one can use DNA to compute *in vitro* computationally intractable mathematical problems. Adleman's result has motivated our work [4], where we followed similar principles and have demonstrated (albeit theoretically) how complex DNA manipulations naturally occurring in living cells could be used to compute solutions for NP-complete problems.

We developed several encodings for instances of HPP through *artificial* gene patterns, so that the results of gene assembly are the solutions of the problem, i.e., hamiltonian paths. In this way, the gene assembly is successful if and only if the problem has a solution (i.e., at least one hamiltonian path).

We used the formalism of MDS descriptors in order to encode HPP instances. For a directed graph $G = (V, E)$ we represented each vertex $p \in V$ as a pointer p, and each directed edge $(p, q) \in E$ we represented as an MDS (p, q). A path puq in the graph from vertex p to vertex q via vertices u we represented as a composite MDS (p, u, q), where u is a string over V.

Since a graph may contain more than two edges incident to a vertex p, then we may get in our encoding more than two occurrences of pointer p for the same gene pattern. Gene patterns with any number of occurrences of pointers yield non-deterministic assembly strategies, facilitating in this way the possibility to assemble molecules corresponding to many different paths in the graph. Moreover, we relaxed the conditions under which molecular operation LD could be applied on a molecule: we allowed LD to excise parts of a molecule which contain any number of MDSs.

For each of the following subsets of intramolecular operations LD, HI, DLAD, $\{\mathsf{LD}, \mathsf{DLAD}\}$, $\{\mathsf{LD}, \mathsf{HI}, \mathsf{DLAD}\}$ we have designed artificial gene patterns which could be assembled by the respective subsets of operations to a molecule represented either as (b, p_1up_n, e) or as $(\overline{e}, \overline{p_nup_1}, \overline{b})$, where in our HPP instance we are interested in hamiltonian paths from p_1 to p_n. Among all the assembled MDSs (b, p_1up_n, e) and $(\overline{e}, \overline{p_nup_1}, \overline{b})$ we choose those which contain all pointers $p \in V$ in string p_1up_n and the length of p_1up_n equals n. I.e., we choose the assembled MDSs which correspond to hamiltonian paths in G.

The length of the encoding on gene patterns for an instance of HPP is $O(m)$ for all subsets of intramolecular operations with the exception of only LD operations. Encoding for LD-only strategies has length $O(mn)$.

The result we presented here is purely theoretical. In order to implement this method in a lab, we have to clarify experimentally the following questions: will a ciliate accept our artificially designed gene pattern and can two ciliates assemble our identical gene pattern into two different macronuclear genes according to our model? Since it was demonstrated experimentally, that templates guide the gene assembly process [49], probably we may tweak the gene assembly by designing our own templates [4].

A slightly different approach to solve computationally intractable problems was presented in [34, 35]. Computational methods to solve instances of *Boolean satisfiability problem* (SAT) were developed for both of the intramolecular and intermolecular models. Unlike the result with HPP from above, the formalism of contextual string rewriting rules was used.

7 Discussion

Initially, the research on the computational nature of gene assembly in stichotrichous ciliates was aiming at explaining and understanding this evolved biological DNA manipulation process. A considerable number of biologically related results were obtained. For instance, model- and strategy-independent invariant properties were discovered for gene assembly in [19, 55]. Theoretical models and the experimental evidence were presented for the short *pointer identification problem.* In particular, template-based recombination models were considered in [21, 7] and experimental work was presented in [49] describing the function of maternal RNA-templates on gene assembly. Virtual knot diagrams were presented in [7] as a physical representation of the homologous recombinations of molecule(s) during gene assembly.

Also, the research on computational properties of gene assembly has a great impact on Computer Science and Discrete Mathematics. In particular, new computational modeling techniques and computing paradigms, formal language theoretical results were obtained. Models for gene assembly and models motivated by gene assembly were introduced in terms of permutations, strings, graphs, formal languages and linear algebra. For instance, language generating systems based on gene assembly were introduced [14], non-contextual string rewriting rules were presented in [9, 12, 13, 24] and used to study the closure properties and language equations. The template-guided recombination-based language operations were explored in [10]. Turing-completeness of both inter- and intramolecular operations was demonstrated in [25, 36, 38, 39]. The research related to the concept of parallel gene assembly was launched in [3, 5, 29, 28, 30]. For a recent detailed survey on the research topics on the computational nature of gene assembly we refer to [40]. Also, for a review on some recent results in this area we refer to [64].

Acknowledgments. The work of Vladimir Rogojin is supported by Science and Technology Center in Ukraine, project 4032. Vladimir Rogojin is on leave of absence from Institute of Mathematics and Computer Science of Academy of Sciences of Moldova, Chisinau MD-2028 Moldova.

References

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

[2] Alhazov, A., Ciliate Operations without Context in a Membrane Computing Framework. *Romanian Journal of Information Science and Technology,* **10**(4), (2007), pp. 315–322.

[3] Alhazov, A., Li, C., Petre, I., Computing the graph-based parallel complexity of gene assembly. *Theoretical Computer Science*, to appear (2008).

[4] Alhazov, A., Petre, I., Rogojin, V., Solutions to Computational Problems through Gene Assembly. *Natural Computing*, **7**(3), Springer, (2008), pp. 385–401.

[5] Alhazov, A., Petre, I., and Rogojin, V.. Computing the parallel complexity of signed graphs: an improved algorithm. *Theoretical Computer Science*, doi:10.1016/j.tcs.2009.02.028, (2009).

[6] Alhazov, A., Petre, I., Verlan, S., A sequence-based analysis of the pointer distribution of stichotrichous ciliates. *Biosystems* **101**, (2010), pp. 109–116.

[7] Angeleska, A., Jonoska, N., Saito, M., and Landweber, L.F., RNA-Template Guided DNA Assembly. *Journal of Theoretical Biology* **248**, Elsevier (2007), 706–720.

[8] Chang, W.J., Bryson, P.D., Liang, H., Shin, M.K., Landweber, L., The evolutionary origin of a complex scrambled gene. *Proceedings of the National Academy of Sciences of the US* **102**(42) (2005) 15149–15154.

[9] Daley, M., Ibarra, OH., Kari, L., Closure propeties and decision questions of some language classes under ciliate bio-operations. *Theoretical Computer Science*, **306**(1-3), (2003), pp. 19–38.

[10] Daley, M., Ibara, OH., Kari, L.,I.McQuillan,K.Nakano, The ld and dlad bio-operations on formal languages. *Autom.Lang.Comb.*, **8**(3), (2003), pp. 477–498.

[11] Daley, M., Kari, L., and McQuillan, I., Families of languages defined by ciliate bio-operations. *Theoretical Computer Science*, **320**(1), (2004), pp. 51–69.

[12] Dassow, J., and Holzer, M., Language families defined by a ciliate bio-operation: hierarchies and decidability problems. *International Journal of Foundations of Computer Science*, **16**(4), (2005), pp. 645–662.

[13] Dassow, J., Mitrana, V., and Salomaa, A., Operations and languages generating devices suggested by the genome evolution. *Theoretical Computer Science*, **270**(1-2), (2002), pp. 701–738.

[14] Dassow, J., and Vaszil, G., Ciliate bio-operations on finite string multisets. In: Ibarra OH, Dang Z (Eds.) *DLT 2006, LNCS* **4036**, (2006), pp. 168–179.

[15] Daley, M., and McQuillan, I., Template-guided DNA recombination. *Theoretical Computer Science*, **330**, (2005), pp. 237-250.

[16] Domaratzki, M., Equivalence in template-guided recombination. *Journal of Natural Computing*, **7**(3), (2007), pp. 439-449.

[17] Ehrenfeucht, A., Harju, T., Petre, I., Prescott, D. M., and Rozenberg, G., *Computation in Living Cells: Gene Assembly in Ciliates*, Springer (2003).

[18] Ehrenfeucht, A., Harju, T., Petre, I., Prescott, D. M., and Rozenberg, G., Formal systems for gene assembly in ciliates. *Theoret. Comput. Sci.* **292** (2003) 199–219.

[19] Ehrenfeucht, A., Petre, I., Prescott, D. M., and Rozenberg, G., Circularity and other invariants of gene assembly in ciliates. In: M. Ito, Gh. Păun and S. Yu (eds.) *Words, semigroups, and transductions*, World Scientific, Singapore, (2001) pp. 81–97.

[20] Ehrenfeucht, A., Prescott, D. M., and Rozenberg, G., Computational aspects of gene (un)scrambling in ciliates. In: L. F. Landweber, E. Winfree (eds.) *Evolution as Computation*, Springer, Berlin, Heidelberg, New York (2001) pp. 216–256.

[21] Prescott, D. M., Ehrenfeucht, A., and Rozenberg, G., Molecular operations for DNA processing in hypotrichous ciliates. *Europ. J. Protistology* **37** (2001) 241–260.

[22] Ehrenfeucht, A., Petre, I., Prescott, D. M., and Rozenberg, G., Universal and simple operations for gene assembly in ciliates. In: V. Mitrana and C. Martin-Vide (eds.) *Words, Sequences, Languages: Where Computer Science, Biology and Linguistics Meet*, Kluwer Academic, Dortrecht, (2001) pp. 329–342.

[23] Ehrenfeucht, A., Petre, I., Prescott, D. M., and Rozenberg, G., String and graph reduction systems for gene assembly in ciliates. *Math. Structures Comput. Sci.* **12** (2001) 113–134.

[24] Freund, R., Martin-Vide, C., and Mitrana, V., On some operations on strings suggested by gene assembly in ciliates. *New Generation Computing*, **20**(3), (2002), pp. 279–293.

[25] Hausmann, K., and Bradbury, P. C. (Eds.) *Ciliates: Cells As Organisms.* Vch Pub, (1997).

[26] Head, T., Formal language theory and DNA: an analysis of the generative capacity of specific recombinant behaviours. *Bulletin of Mathematical Biology*, **49**(6), (1987), pp. 737–759.

[27] Hogan, D.J., Hewitt, E.A, Orr, K.E., Prescott, D.M., Müller, K.M., Evolution of IESs and scrambling in the actin I gene in hypotrichous ciliates. *PNAS* 98 (26) (2001) 15101–15106.

[28] Harju, T., Li, C., Petre, I., Parallel Complexity of Signed Graphs for Gene Assembly in Ciliates. *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, **12**(8), (2008) 731–737.

[29] Harju, T., Li, C., Petre, I., Graph Theoretic Approach to Parallel Gene Assembly. *Discrete Applied Mathematics*, **156**(18), Elsevier, (2008), pp. 3416–3429.

[30] Harju, T., Li, C., Petre, I. and Rozenberg, G., Parallelism in gene assembly, *Natural Computing*, 5 (2) (2006) 203–223.

[31] Harju, T., Li, C., Petre, I. and Rozenberg, G., Complexity Measures for Gene Assembly. In: K. Tuyls (Eds.), Proceedings of the *Knowledge Discovery and Emergent Complexity in Bioninformatics* workshop. Springer, *Lecture Notes in Bioinformatics* **4366**, 2007.

[32] Harju, T., Petre, I., and Rozenberg, G., Modelling simple operations for gene assembly. In: J.Chen, N.Jonoska, G.Rozenberg (Eds.) *Nanotechnology: Science and Computation* (2006) 361–376.

[33] Harju, T., Petre, I., Rogojin, V. and Rozenberg, G., Patterns of Simple Gene Assembly in Ciliates, *Discrete Applied Mathematics*, **156**(14), Elsevier, (2008), pp. 2581–2597.

[34] Ishdorj, T.-O., Loos, R., and Petre, I., Computational Efficiency of Intermolecular Gene Assembly. *Fundamenta Informaticae*, **84**(3-4), (2008), pp. 363–373.

[35] Ishdorj, T.-O., and Petre, I., Computing Through Gene Assembly. *Unconventional Computation*, **4618**, (2007), pp. 91–105.

[36] Ishdorj, T.-O., Petre, I. and Rogojin, V, Computational Power of Intramolecular Gene Assembly. *International Journal of Foundations of Computer Science*, **18**(5), World Scientific, (2007), pp. 1123–1136.

[37] Jahn, C. L., and Klobutcher, L. A., Genome remodeling in ciliated protozoa. *Ann. Rev. Microbiol.* **56** (2000), 489–520.

[38] Kari, L., Kari, J., and Landweber, L. F., Reversible molecular computation in ciliates. In: J. Karhumäki, H. Maurer, G. Păun, and G. Rozenberg (eds.), *Jewels are Forever*, Springer, Berlin Heidelberg, New York, (1999), pp. 353–363.

[39] Kari, L., and Landweber, L. F., Computational power of gene rearrangement. In: E. Winfree and D. K. Gifford (eds.) *Proceedings of DNA Bases Computers, V* American Mathematical Society (1999) 207–216.

[40] Brijder, R., Daley, M., Harju, T., Jonoska, N., Petre, I., and Rozenberg, G., Computational nature of gene assembly in ciliates. In: L. Kari, G. Rozenberg (Eds.), *Handbook of Natural Computing*, Springer, (2009), to appear.

[41] Kari, L., and Thierrin, G., Contextual insertion/deletions and computability. *Information and Computation*, **131**, (1996), pp. 47–61.

[42] Landweber, L. F., and Kari, L., The evolution of cellular computing: Nature's solution to a computational problem. In: *Proceedings of the 4th DIMACS Meeting on DNA-Based Computers*, Philadelphia, PA (1998) pp. 3–15.

[43] Landweber, L. F., and Kari, L., Universal molecular computation in ciliates. In: L. F. Landweber and E. Winfree (eds.) *Evolution as Computation*, Springer, Berlin Heidelberg New York (2002).

[44] Langille, M., Petre, I., Simple gene assembly is deterministic. *Fundamenta Informaticae* **72**, IOS Press, (2006), pp. 1–12.

[45] Langille, M., Petre, I., Rogojin, V., Three models for gene assembly in ciliates: a comparison. *Proceedings of BIONETICS 2008*, to appear (2009).

[46] Lynn, D.H., Small, E.B., A Revised Classification of the Phylum Ciliophora Doflein, 1901. *Revista de la Sociedad Mexicana de Historia Natural*, Mexico, 47, (1997), pp. 65–78.

[47] Marcus, S., Contextual Grammars. *Revue Roumaine de Matematique Pures et Appliquees,*, **14**, (1969), pp. 1525–1534.

[48] Martin-Vide, C., Păun, G., Pazos, J., and Rodriguez-Paton, A., Tissue P systems. *Theoretical Computer Science*, **296**(2), pp. 295–326. DOI: 10.1016/S0304-3975(02)00659-X.

[49] Nowacki, M., Vijayan, V., Zhou, Y., Schotanus, K., Doak, T.G., Landweber, L.F., RNA-mediated epigenetic programming of a genome-rearrangement pathway. *Nature* **451**, doi:10.1038/nature06452, (2008) 153–158.

[50] Papadimitriou, C.H., *Computational Complexity.* Addison-Wesley, (1994).

[51] Păun, G., *Marcus Contextual Grammars*, Kluwer, Dordrecht, (1997).

[52] Păun, G., Computing with Membranes, *TUCS Technical Report*, **208**, (1998).

[53] Păun, G., *Membrane Computing: An Introduction.* Springer, (2002).

[54] Păun, G., Rozenberg, G., Salomaa, A., *DNA Computing: New Computing Paradigms.* Springer, (1998).

[55] Petre, I., Invariants of gene assembly in stichotrichous ciliates. *IT*, Oldenbourg Wissenschftsverlag, **3** (2006), pp. 161–167.

[56] Petre, I., and Rogojin, V., Decision Problem for Shuffled Genes, *Information and Computation*, **206**(11), Elsevier, (2008), pp. 1346–1352.

[57] Prescott, D. M., DNA manipulations in ciliates. In: W. Brauer, H. Ehrig, J. Karhumäki, A. Salomaa (eds.) *Formal and Natural Computing: essays dedicated to Grzegorz Rozenberg*, LNCS 2300, Springer (2002) 394–417.

[58] Prescott, D. M., Genome gymnastics: unique modes of DNA evolution and processing in ciliates. *Nature Reviews, Genetics*, **1**(3), (2000), pp. 191–198.

[59] Prescott, D. M., The DNA of ciliated protozoa. *Microbiol. Rev.* **58**(2) (1994) 233–267.

[60] Prescott, D. M., Ehrenfeucht, A., and Rozenberg, G., Template-guided recombination for IES elimination and unscrambling of genes in stichotrichous ciliates. *Journal of Theoretical Biology*, **222**(3), (2003), pp. 323–330.

[61] Prescott, D. M., and Rozenberg, G., How ciliates manipulate their own DNA A splendid example of natural computing. *Natural Computing*, **1**, (2002), pp. 165-183.

[62] Păun, G., Membrane Computing. *Scholarpedia*, **5**(1):9259.

[63] Rogojin, V., Successful Elementary Gene Assembly Strategies. *International Journal of Foundations of Computer Science*, to appear, (2009).

[64] Rogojin, V., Gene Assembly in Stichotrichous Ciliates: Elementary Operations, Parallelism and Computation. *TUCS Dissertations*, **117**, 2009.

[65] Wright, A.-D. G. and Lynn, D. H., Maximum ages of ciliate lineages estimated using a small subunit rRNA molecular clock: Crown eukaryotes date back to the Paleoproterozoic. *Arch Protistenkd*, **148**, (1997), pp. 329-341.

[66] Yao, M.C., Fuller, P., Xi, X., Programmed DNA Deletion As an RNA-Guided System of Genome Defense, *Science* 300 (2003) 1581–1584.

V. Rogojin

Received November 21, 2010

University of Helsinki
Faculty of Medicine
Genome-Scale Biology Research Program
Computational Systems Biology Laboratory
Biomedicum, Helsinki 00014, Finland
Phone: +358 919 125 407
E–mail:*vladimir.rogojin@helsinki.fi*, *vrogojin@vrogojin.net*

Institute of Mathematics and Computer Science
Academy of Sciences of Moldova
5 Academiei str., Chişinău, MD-2028, Moldova

Computer Science Journal of Moldova, vol.**18**, no.2(53), 2010

Recent Developments on Insertion-Deletion Systems[*]

Sergey Verlan

Abstract

This article gives an overview of the recent developments in the study of the operations of *insertion* and *deletion*. It presents the origin of these operations, their formal definition and a series of results concerning language properties, decidability and computational completeness of families of languages generated by insertion-deletion systems and their extensions with the graph-control. The basic proof methods are presented and the proofs for the most important results are sketched.

Keywords: insertion-deletion systems, computational completeness, decidability, graph control, P systems.

1 Introduction

In general form, an insertion operation means adding a substring to a given string in a specified (left and right) context, while a deletion operation means removing a substring of a given string being in a specified (left and right) context. An insertion or deletion rule is defined by a triple (u, x, v) meaning that x can be inserted between u and v or deleted if it is between u and v. Thus, an insertion corresponds to the rewriting rule $uv \to uxv$ and a deletion corresponds to the rewriting rule $uxv \to uv$. A finite set of insertion-deletion rules, together with a set of axioms provides a language generating device: starting from the set of initial strings and iterating insertion or deletion operations as defined by the given rules one gets a language. The size of

[*]This work was supported by Science and Technology Center in Ukraine project Nr. 4032

the alphabet, the number of axioms, the size of contexts and of the inserted or deleted string are natural descriptional complexity measures for insertion-deletion systems.

The idea of insertion of one string into another was firstly considered with a linguistic motivation in [23] and latter developed in [8, 29]. Marcus contextual grammars investigated in above references consider couples $(x,(u,v))$, meaning that words u and v can be adjoined to the word x. This corresponds in some sense to grammars having rules of type $x \to uxv$, *i.e.*, u and v are inserted around the position marked by x. Such grammars are alternative concepts to Chomsky grammars and present the evolution of the descriptive linguistics. Many interesting linguistic properties like ambiguity and duplication can be captured in this framework. The insertion of a string in a specified context was firstly considered in [8].

In [9, 10] the insertion operation and its iterated variant is introduced with a different motivation. The author considers this operation as generalization of Kleene's operations of concatenation and closure [17]. The operation of concatenation would produce a string x_1x_2y from two strings x_1x_2 and y. By allowing the concatenation to happen anywhere in the string and not only at its right extremity a string x_1yx_2 can be produced, *i.e.*, y is inserted into x_1x_2. In [13] the deletion is defined as a right quotient operation which happens not necessarily at the rightmost end of the string. In the same thesis the duality between the insertion and deletion is also highlighted: any insertion system generating a language $\mathcal{L}$ is at the same time a deletion system recognizing $\mathcal{L}$. The operations considered in above works correspond to context-free variants of insertion and deletion operations, because no contexts are used. In the same place several other variants of insertion and deletion are introduced and their closure properties are investigated.

The third inspiration for insertion and deletion operations comes, surprisingly, from the field of molecular biology. In fact they correspond to a mismatched annealing of DNA sequences. We refer to [32] for more details. Such operations are also present in the evolution processes under the form of point mutations as well as in RNA editing, see the discussions in [3], [34] and [32]. This biological motivation of insertion-deletion operations lead to their study in the framework of molecular computing, see, for example, [6], [14], [32], [35].

This article is organized as follows. After a formal definition given in Section 2, the next section describes existing formal proof methods and presents a recent proof technique. Section 4 considers context-free insertion-deletion systems and their links to the previous results in the formal language theory. After that in Section 5 one-sided insertion-deletion systems are considered. Section 6 investigates the graph-control extension of insertion-deletion systems that permits to increase the computational power for non-complete classes. Finally, Section 7 considers the variant where only the insertion operation is used.

2 Formal definition

We do not present here definitions concerning standard concepts of the theory of formal languages and we refer to [33] for more details.

The empty string is denoted by λ. The *length* of the word $x \in V^*$ is the number of symbols which appear in x and it is denoted by $|x|$. The number of occurrences of a symbol $a \in V$ in $x \in V^*$ is denoted by $|x|_a$. If $x \in V^*$ and $U \subseteq V$, then we denote by $|x|_U$ the number of occurrences of symbols from U in x. For a word $w \in V^*$ we define $Perm(w) = \{w' : |w'|_a = |w|_a \text{ for all } a \in V\}$. The length set of a language L is defined as $|L| = \{|x| : x \in L\}$. The length set of a family of languages $\mathcal{F}$ is defined analogously: $N\mathcal{F} = \{|L| : L \in \mathcal{F}\}$.

The family of matrix languages, *i.e.*, the family of languages generated by matrix grammars without appearance checking is denoted by MAT. The family of recursively enumerable languages is denoted by RE. The *Parikh image* of a language family $\mathcal{F}$ is a family of sets of vectors denoted by $Ps\mathcal{F}$ (we assume a fixed ordering on the alphabet $T = \{a_1, \ldots, a_n\}$), and is defined as follows: $Ps(L) = \{(|w|_{a_1}, \ldots, |w|_{a_n}) : w \in L\}$ and $Ps\mathcal{F} = \{Ps(L) : L \in \mathcal{F}\}$.

An *insertion-deletion system* is a construct $ID = (V, T, A, I, D)$, where V is an alphabet, $T \subseteq V$, A is a finite language over V, and I, D are finite sets of triples of the form (u, α, v), $\alpha \neq \lambda$, where u and v are strings over V. The elements of T are *terminal* symbols (in contrast, those of $V - T$ are called nonterminals), those of A are *axioms*, the triples in I are *insertion rules*, and those from D are *deletion rules.* An insertion rule $(u, \alpha, v) \in I$ indicates that the string α can be inserted between u and v, while a deletion rule $(u, \alpha, v) \in D$ indicates that α can be removed from the context (u, v). As stated otherwise, $(u, \alpha, v) \in I$ corresponds to the rewriting rule $uv \to u\alpha v$, and $(u, \alpha, v) \in D$ corresponds to the rewriting rule $u\alpha v \to uv$. We denote by $\Longrightarrow_{ins}$ the relation defined by an insertion rule (formally, $x \Longrightarrow_{ins} y$ iff $x = x_1uvx_2, y = x_1u\alpha vx_2$, for some $(u, \alpha, v) \in I$ and $x_1, x_2 \in V^*$) and by $\Longrightarrow_{del}$ the relation defined by a deletion rule (formally, $x \Longrightarrow_{del} y$ iff $x = x_1u\alpha vx_2, y = x_1uvx_2$, for some $(u, \alpha, v) \in D$ and $x_1, x_2 \in V^*$). We refer by $\Longrightarrow$ to any of the relations $\Longrightarrow_{ins}, \Longrightarrow_{del}$, and denote by $\Longrightarrow^*$ the reflexive and transitive closure of $\Longrightarrow$ (as usual, $\Longrightarrow^+$ is its transitive closure).

The language generated by ID is defined by

$$L(ID) = \{w \in T^* \mid x \Longrightarrow^* w, x \in A\}.$$

The complexity of an insertion-deletion system $ID = (V, T, A, I, D)$ is described by the vector $(n, m, m'; p, q, q')$ called *size*, where

$$\begin{aligned} n &= \max\{|\alpha| \mid (u, \alpha, v) \in I\}, & p &= \max\{|\alpha| \mid (u, \alpha, v) \in D\}, \\ m &= \max\{|u| \mid (u, \alpha, v) \in I\}, & q &= \max\{|u| \mid (u, \alpha, v) \in D\}, \\ m' &= \max\{|v| \mid (u, \alpha, v) \in I\}, & q' &= \max\{|v| \mid (u, \alpha, v) \in D\}. \end{aligned}$$

We also denote by $INS_n^{m,m'}DEL_p^{q,q'}$ corresponding families of insertion-deletion systems. Moreover, we define the total size of the system as the sum of all numbers above: $\psi = n + m + m' + p + q + q'$.

If some of the parameters n, m, m', p, q, q' is not specified, then we write instead the symbol $*$. In particular, $INS_*^{0,0}DEL_*^{0,0}$ denotes the family of languages generated by *context-free insertion-deletion systems.* If one of numbers from the couples m, m' and/or q, q' is equal to zero (while the other is not), then we say that corresponding families have a one-sided context.

We remark that, historically, another complexity measure called *weight* was used for insertion-deletion systems. It corresponds to 4-tuples $(n, \bar{m}; p, \bar{q})$, where $\bar{m} = \max\{m, m'\}$ and $\bar{q} = \max\{q, q'\}$.

3 Basic simulation principles

In this section we show some important properties of insertion-deletion systems, present some normal forms and indicate basic methods for equivalence proofs used in the rest of the chapter.

We start with the presentation of the normal form for insertion-deletion systems.

Definition 3.1. An insertion-deletion system $ID = (V \cup \{\$\}, T, A, I, D \cup D_2)$ of size $(n, m, m'; p, q, q')$ is said to be in the *normal form* if

- for any $(u, x, v) \in I$ it holds $|u| = m$, $|v| = m'$, $|x| = n$,
- for any $(u, x, v) \in D$ it holds $|u| = q$, $|v| = q'$, $|x| = p$,
- for any $(u, x, v) \in D$ it holds that x contains no letters from T,
- the set D_2 is defined as $D_2 = \{(\lambda, \$, \lambda)\}$.

Theorem 3.2. *For any insertion-deletion system ID it is possible to construct a system ID' in normal form and having same size such that $L(ID) = L(ID')$.*

This affirmation is quite obvious. For the first two conditions it is enough to replace any rule having left or right contexts of a smaller size by a group of rules, where the left (resp. right) context is a string over $V \cup \{\$\}$ of the required size. The same holds for the inserted or deleted symbol and axioms. More precisely, the new symbol \$ permits to fill the context of rules and sizes of axioms up to the desired size.

The third condition can be satisfied as follows. For any terminal symbol $t \in T$ a special non-terminal N_t is considered. All rules and axioms involving t are duplicated and t replaced by N_t. This construction ensures that symbol N_t acts like an alias for the symbol t, *i.e.* for any derivation producing $w_1 t w_2$ there is another derivation producing $w_1 N_t w_2$. Hence there is no difference between erasing t or N_t, therefore all deletion rules involving t can be omitted. A formal proof of the theorem can be found in [2].

Example 3.3.

Consider the system $ID = (\{a,b,C\},\{a,b\},\{ab\},I,D)$ of size $(2,1,1;2,1,1)$, with $I = \{(a,aC,b),(a,b,C)\}$ and $D = \{(b,C,b)\}$. Then ID' can be defined as follows: $ID' = (\{a,b,C,\$\},\{a,b\}, \{ab \sqcup\!\sqcup \$\$\}, I', D' \cup \{(\lambda,\$,\lambda)\})$, where $I' = \{(a,aC,b),(a,\$b,C), (a,b\$,C)\}$ and $D' = \{(b,C\$,b),(b,\$C,b)\}$.

Insertion-deletion systems represent a powerful model of computation. If the size of the system is not bounded, then an arbitrary grammar can be simulated.

Theorem 3.4. *For any type-0 grammar $G = (N,T,S,P)$ there is an insertion-deletion system $ID = (V,T,A,I,D)$ such that $L(G) = L(ID)$.*

Proof. Let $V = N \cup \{\#_i : 1 \le i \le |P|\} \cup \{\$\}$. Let $k_1 = \max\{|u|, u \to v \in P\}$ and $k_2 = \max\{|v|, u \to v \in P\}$. Consider $k = \max(k_1,k_2)$. The set A is defined as $A = \{\$^k S \$^k\}$.

For any rule $i : u \to v \in P$ we add insertion rules $(xu, \#_i v, y)$, $x, y \in (N \cup \{\$\})^*$, $|xu| = k$, $|y| = k$, to I and a deletion rule $(x, u\#_i, v)$, $x \in N \cup \{\$\}$ to D. Finally, a rule $(\lambda,\$,\lambda)$ is added to D.

It is not difficult to see that such system simulates G. Indeed, for any derivation $w_1uw_2 \Longrightarrow w_1vw_2$ in G there is a following two-step derivation $\$^k w_1uw_2\$^k \Longrightarrow \$^k w_1u\#_i vw_2\$^k \Longrightarrow \$^k w_1vw_2\$^k$ in ID that simulates the corresponding production of G. If $w \in L(G)$ then the string $\$^k w \$^k$ will be obtained in ID. Additional symbols \$ can be deleted at this moment. So $w \in L(ID)$.

For the converse inclusion it is enough to observe that if an insertion rule $(xu, \#_i v, y)$ is used, then no more insertions inside the corresponding site xu can be done. So, the only way to eliminate the symbol $\#_i$ is to perform the corresponding deletion. Hence the computation in ID can be rearranged in such a way that an insertion is followed by the corresponding deletion. This corresponds to a derivation step in G, which completes the proof. □

As one can see from the previous theorem, the basic idea of grammar simulation by insertion-deletion systems is a construction of a set of related insertion and deletion rules that shall be used in some specified sequence thus performing a grammar rule simulation. Usually, insertion rules introduce new non-terminal symbols in the string which can be deleted only by corresponding deletion rules (like symbols $\#_i$ in the theorem above). If the correct sequence is not performed, then some non-terminal symbols that cannot be deleted will remain in the string. In the subsequent sections different variants of this method are shown permitting to decrease the size of used insertion and deletion rules.

3.1 The method of direct simulation

A simulation of type-0 grammars by insertion-deletion systems is the main method permitting to prove the computational completeness of insertion-deletion systems. However, when several such results are established, it is much easier to prove the computational completeness by simulating another insertion-deletion systems. For example:

Theorem 3.5. *$INS_1^{1,1}DEL_2^{0,0} = RE$.*

Sketch of Proof. The proof may be done by simulating insertion-deletion systems of size $(1,1,1;1,1,1)$ which are known to be computationally complete, see [35, 36]. In this case it is enough to show how a deletion rule (a,b,c), $a,b,c \in V$ can be simulated using insertion and deletion rules of size $(1,1,1;2,0,0)$. Let $a \neq b \neq c$. Then a deletion rule (a,b,c) with label i may be simulated by a sequence of the following rules: $\{(a, \}_i, b), (b,]_i, c), (a, [_i, \}_i), ([_i, \{_i, \}_i), ([_i, K_i, \{_i,)\} \subseteq I$ and $(\lambda, \{_i\}_i, \lambda), (\lambda, K_i b, \lambda), (\lambda, [_i]_i, \lambda) \subseteq D$. The simulation is performed as follows (we underline the inserted symbols):

$$\begin{aligned} w_1abcw_2 &\Longrightarrow_{ins} w_1a\underline{\}_i}bcw_2 \Longrightarrow_{ins} w_1a\underline{[_i}\}_ibcw_2 \Longrightarrow_{ins} w_1a[_i\}_ib\underline{]_i}cw_2 \\ &\Longrightarrow_{ins} w_1a[_i\underline{\{_i}\}_ib]_icw_2 \Longrightarrow_{ins} w_1a[_i\underline{K_i}\{_i\}_ib]_icw_2 \Longrightarrow_{del} \\ &\Longrightarrow_{del} w_1a[_iK_ib]_icw_2 \Longrightarrow_{del} w_1a[_i]_icw_2 \Longrightarrow_{del} w_1acw_2. \end{aligned}$$

The idea behind the simulation is the following. Symbols $[_i$ and $]_i$ delimit the deletion site. Symbol K_i performs the deletion of b, while symbols $\}_i$ and $\{_i$ ensure that K_i is inserted only once after $[_i$ (hence only one b can be deleted). If all the above steps are not performed, then some of additional symbols will remain in the string, hence it will never become terminal. This is a common method of simulation: the working (insertion or deletion) site is delimited by special symbols in order to avoid interactions between several such sites and inside the site the sequence of insertions and deletions permits to simulate exactly one application of the corresponding rule. All additional symbols are related in such a way that the whole sequence of insertions and deletions shall be performed in order to eliminate all of them.

We remark that it would be wrong to simulate a deletion rule (a, b, c) by only rules $\{(a, [_i, b), (b,]_i, c), ([_i, K_i, b)\} \subseteq I$ and (λ, K_ib, λ), $(\lambda, [_i]_i, \lambda) \subseteq D$, because it is possible to erase several symbols b, which leads to a wrong computation:

$$\begin{aligned} w_1abbcw_2 &\Longrightarrow_{ins} w_1a\underline{[_i}bbcw_2 \Longrightarrow_{ins} w_1a[_ibb\underline{]_i}cw_2 \Longrightarrow_{ins} \\ &\Longrightarrow_{ins} w_1a[_i\underline{K_i}bb]_icw_2 \Longrightarrow_{del} w_1a[_ib]_icw_2 \Longrightarrow_{ins} \\ &\Longrightarrow_{ins} w_1a[_i\underline{K_i}b]_icw_2 \Longrightarrow_{del} w_1a[_i]_icw_2 \Longrightarrow_{del} w_1acw_2. \end{aligned}$$

$\square$

The above approach is very powerful and it permits to establish the computational completeness of the corresponding class of insertion-deletion systems in a much easier way. For example, the proof of Theorem 3.5 in [32] (Theorem 6.3) takes more than two pages. The method is quite generic, in order to use it one should find a computational complete class of insertion-deletion systems having same insertion or deletion parameters. Then, in order to prove the computational completeness, it is sufficient to simulate corresponding deletion or insertion operation. This is significantly easier than the simulation of a Chomsky grammar because only left-hand or only right-hand side of a production $u \rightarrow v$ shall be simulated.

Most of the recent results about the universality of insertion-deletion systems are obtained using this technique.

4 Context-free insertion-deletion systems

In this section we present an important class of insertion-deletion systems: systems with context-free rules. This permits to bridge recent results with early investigations from [9] and [13] giving answers to old questions from this area.

4.1 Computational completeness results

We start with the sketch of the proof of the computational completeness for context-free insertion-deletion systems. More details can be found in [24].

Theorem 4.1. *$INS_*^{0,0}DEL_*^{0,0} = RE$.*

Sketch of proof. Let $G = (N, T, S, P)$ be type-0 Chomsky grammar where N, T are disjoint alphabets, $S \in N$, and P is a finite subset of rules of the form $u \to v$ with $u, v \in (N \cup T)^*$ and u contains at least one letter from N. We assume all rules from P labeled in a one-to-one manner with elements of a set M, disjoint of $N \cup T$.

We construct the following context-free insertion-deletion system: $\gamma = (N \cup T \cup M, T, \{S\}, I, D)$, where

$$I = \{(\lambda, vR, \lambda) \mid R : u \to v \in P,\ R \in M,\ u, v \in (N \cup T)^*\},$$
$$D = \{(\lambda, Ru, \lambda) \mid R : u \to v \in P,\ R \in M,\ u, v \in (N \cup T)^*\}.$$

Two rules $(\lambda, vR, \lambda) \in I, (\lambda, Ru, \lambda) \in D$ as above are said to be *M-related.*

We have the equality $L(G) = L(\gamma)$.

The inclusion $L(G) \subseteq L(\gamma)$ is obvious: each derivation step $x_1ux_2 \Longrightarrow x_1vx_2$, performed in G by means of a rule $R : u \to v$, can be simulated in γ by an insertion operation step $x_1ux_2 \Longrightarrow_{ins} x_1vRux_2$ which uses the rule $(\lambda, vR, \lambda) \in I$, followed by the deletion operation $x_1vRux_2 \Longrightarrow_{del} x_1vx_2$ which uses the rule $(\lambda, Ru, \lambda) \in D$.

Consider now the inclusion $L(\gamma) \subseteq L(G)$. The idea of the proof is to transform any terminal derivation in γ into one in which any two consecutive (odd, even) derivations steps simulate one production in G. Because the labels of rules from P precisely identify a pair of M-related insertion-deletion rules, and the elements of M are nonterminal symbols for γ, every terminal derivation with respect to γ must involve the same number of insertion steps and deletion steps; moreover, these steps are performed by using pairs of M-related rules from I and D.

For every terminal derivation in γ it is possible to construct an equivalent derivation, using the same rules in a different order, and having only matching pairs of consecutive rules, *i.e.* odd steps $w_i \Longrightarrow_{ins} w_{i+1}$ are performed by a rule $(\lambda, vR, \lambda) \in I$, while even steps $w_{i+1} \Longrightarrow w_{i+2}$ are performed by using the M-related rule $(\lambda, Ru, \lambda) \in D$. Clearly, two consecutive steps of a derivation in γ which use M-related rules $(\lambda, vR, \lambda) \in I, (\lambda, Ru, \lambda) \in D$, correspond to a derivation step in G which uses the rule $R : u \to v$. This implies the inclusion $L(\gamma) \subseteq L(G)$. □

The context control of a type 0 grammar does not really disappear in the corresponding insertion-deletion system (as constructed in Theorem 4.1 above). It rather changes its form, becoming a rigid synchronization of insertions and deletions. In other terms, if a word u represents the context of a word v in a "context-sensitive production" $R : u \to v$, then in the corresponding insertion-deletion system the word v will also be conditioned by the later occurrence of u in a successful derivation (hence u is yet again the context of v). This condition is enforced by the newly introduced symbol R which acts as a "remote context binder". The fact that the context u "seems" to appear after the context-controlled v is of no importance, reflecting the reversal of generative process of the grammar.

Let us denote by $L \leftrightarrows {}^{L_1}_{L_2}$ the operation of insertion-deletion that inserts words from L_1 into L or deletes words belonging to L_2 from L and by $L \leftrightarrows^* {}^{L_1}_{L_2}$ its reflexive and transitive closure. Then the following representation of RE holds:

Theorem 4.2. *Any language $L \in RE$ can be represented in the following form $L = \left(\{S\} \leftrightarrows^* {}^{L_1}_{L_2}\right) \cap T^*$, where L_1 and L_2 are two finite languages and T is an alphabet.*

In the proof of Theorem 4.1, the length of inserted or deleted strings is not bounded, but a bound can be easily found by controlling the length of strings appearing in the rules of the starting type-0 grammar:

Theorem 4.3. *$INS_3^{0,0}DEL_3^{0,0} = RE$.*

Proof. Let $G = (N, T, S, P)$ be type-0 Chomsky grammar in Kuroda normal form. Then, the rules of the context-free insertion-deletion system constructed in the proof of Theorem 4.1 are of the form $(\lambda,\ \alpha,\ \lambda)$ with $|\alpha| \leq 3$, hence $RE \subseteq INS_3^0DEL_3^0$. □

The total size of the system provided by the proof of Theorem 4.1 is 6. We can improve by one this result, by decreasing by one either the length of the inserted strings or the length of the deleted strings. These proofs can be done by a direct simulation of systems of size $(3, 0, 0; 3, 0, 0)$ using the method presented in Section 3.1.

Theorem 4.4. *[24] $INS_3^{0,0}DEL_2^{0,0} = RE$.*

A counterpart of this result is also true: we can trade-off the length of inserted and deleted strings.

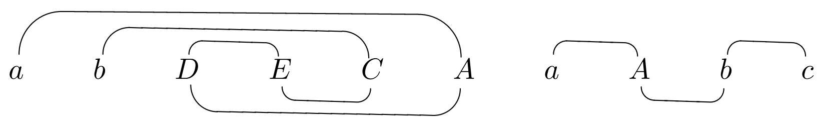
Theorem 4.5. *[24] $INS_2^{0,0}DEL_3^{0,0} = RE$.*

4.2 Non-completeness results

We show below that the above complexity parameters for context-free insertion-deletion systems are optimal. If one of the parameters is further decreased, then the language generated by such systems is included in the family of context-free languages.

The main idea used to obtain this result is that the non-terminal alphabet can be omitted, hence, the deletion can also be omitted.

This can be argued as follows. Consider a derivation of $w \in T^*$ starting from an empty word. Let us mark the corresponding insertion pairs by an overline and the corresponding deletion pairs by an underline. For example, suppose that we insert aA, after that bC in position 1, DE in position 2, aA in position 6 and bc in position 8. After that suppose that we delete EC, DA and Ab. Then the corresponding marking will be as follows (the resulting word is $w = abac$):



We may interpret symbols as labeled graph nodes and lines as edges. In this case we obtain a graph. It is easy to observe that this graph consists of a set of disjoint linear paths and/or cycles. Indeed, for each node, at most two edges corresponding to an insertion and a deletion may be drawn. Let us also label edges corresponding to insertions by i and edges corresponding to deletions by d. If we take the example above, we obtain:

$$a \overset{i}{\text{——}} A \overset{d}{\text{——}} D \overset{i}{\text{——}} E \overset{d}{\text{——}} C \overset{i}{\text{——}} b$$
$$a \overset{i}{\text{——}} A \overset{d}{\text{——}} b \overset{i}{\text{——}} c$$

We may suppose that the first and the last edge of a path are marked with i. If this is not the case, we add an additional node labeled by λ and we connect this node with the last node by a path labeled by i. In particular, a path containing only one letter a (corresponding to an insertion of a) will be written as $\lambda \overset{i}{\text{—}} a$. Hence, each path consists of sequences of one insertion followed by one deletion.

We observe that for a derivation of a word $w \in T^*$ there can only be paths of the following 4 types: (1) paths that start with a letter $a \in T$ and that end with a letter $b \in T$; (2) Paths that have at one end a terminal letter a and at the other end λ; (3) paths that have λ at both ends; (4) Cycles.

We remark that in Case 1 the path leads to the word ab (*i.e.*, contributes to the production of the subword ab of w), in the second case the path produces the letter a and in the last two cases the path generates the empty word.

Without loss of generality, we may suppose that there are no paths of type 3 and 4, because by eliminating the corresponding insertions and deletions we obtain the same word.

Suppose that we have a path marked by over- and underlines as above. We shall understand by an interior of the path the set of all positions that are underlined. In the example above, all positions between D and the first A form the interior of the path. It is clear that no other path (of type 1 and 2) may be situated in the interior of some path, because in this case the corresponding deletion cannot be performed. Consequently, all paths are independent of each other, and we may group rules corresponding to each path and compute paths one after another. Moreover, each path contributes to at most two terminal symbols of the resulting word. Therefore, the computation consists of insertion of terminal symbols corresponding to paths ends as well as of deletion of terminal symbols.

Moreover, we can show that it is possible to precompute all possible paths. This may be done by using the following observation. We may assume that each path p has the following property: if $A \overset{i}{-} B$ belongs to p, then p does not contain an insertion that has A in the left-hand side ($A \overset{i}{-} X$) or B in the right-hand side ($Y \overset{i}{-} B$). This assertion is obvious, because if p contains such a pair, for example $p = \cdots \overset{d}{-} A \overset{i}{-} X \overset{d}{-} \cdots \overset{d}{-} A \overset{i}{-} B \cdots$, then we may eliminate the subpath between two A's by obtaining an equivalent path (that leads to the same ends) $p' = \cdots \overset{d}{-} A \overset{i}{-} B \cdots$. So, the length of each path is bounded by $2 \cdot card(V)$, and we may precompute all possible paths.

In a similar manner it can be proved that the nonterminal alphabet is not relevant even in the general case. See [37] for details.

Lemma 4.6. *Let $ID = (V, T, A, I, D)$ be a context-free insertion-deletion system of size $(2,0,0;2,0,0)$. Then it is possible to construct a system $ID_2 = (T, T, A_2, I_2, D_2)$ of size $(2,0,0;2,0,0)$ such that $L(ID) = L(ID_2)$.*

Moreover, if we consider that the initial system is in the normal form, then there are no deletions of terminal symbols. Hence we obtain that it is sufficient to consider insertion-only systems as $INS_2^{0,0}DEL_2^{0,0} \subseteq INS_2^{0,0}DEL_0^{0,0}$.

We can describe insertion-deletion systems of size $(2,0,0;0,0,0)$ by the following context-free grammar, which is a particular case of a more general result for systems of size $(*,1,1;0,0,0)$ given in [32].

Let $ID = (T,T,A,I,\emptyset)$ be an insertion-deletion system of size $(2,0,0;0,0,0)$. We construct the following context-free grammar $G = (\{Z,S\},T,Z,P)$. Define $P = P_A \cup P_I \cup \{S \to \lambda\}$, where

$$\begin{aligned} P_A &= \{Z \to Sa_1Sa_2S\dots Sa_nS \mid a_1a_2\dots a_n \in A\}, \\ P_I &= \{S \to SaSbS \mid (\lambda, ab, \lambda) \in I\} \cup \{S \to SaS \mid (\lambda, a, \lambda) \in I\}. \end{aligned}$$

It is clear that $L(G) = L(ID)$. Indeed, symbol S marks all possible insertion positions and permits the simulation of insertion rules as well.

Consequently, we obtain:

Theorem 4.7. *$INS_2^{0,0}DEL_2^{0,0} = INS_2^{0,0}DEL_0^{0,0} \subset CF$.*

Proof. The strictness of the inclusion follows from the fact that insertion-deletion systems of size $(2,0,0;0,0,0)$ cannot generate the language $L = \{a^*b^*\}$. Indeed, consider an arbitrary system $ID = (T,T,A,I,\emptyset)$. It is easy to observe that for each word w that belongs to $L(ID)$, words $\{x^*wx^* \mid (\lambda,x,\lambda) \in I\}$ belong to $L(ID)$. Therefore, if we suppose that $L(ID)$ is not finite, then $I \neq \emptyset$, and then for any word $w \in L(ID)$, there are words $\{x^*wx^* \mid (\lambda,x,\lambda) \in I\}$ in $L(ID)$. It is easy to see that L does not have such a property. $\square$

Theorem 4.8. *$INS_2^{0,0}DEL_2^{0,0}$ is incomparable with REG.*

Proof. Previous theorem gives $REG \setminus INS_2^{0,0}DEL_2^{0,0} \neq \emptyset$. It is also clear that the Dyck language D_n may be generated by a context-free insertion system having insertion rules $(\lambda, a_i\bar{a}_i, \lambda)$, $1 \leq i \leq n$. Hence, the assertion is proved. $\square$

From the description above it is clear that languages generated by insertion-deletion systems of size $(2,0,0;2,0,0)$ have a particular structure (below, we denote by $\prod$ the concatenation operation).

Theorem 4.9. *A language L belongs to $INS_2^{0,0}DEL_2^{0,0}$ if and only if it can be represented in the form*

$$L = h\left(T'^* \sqcup\!\sqcup \bigcup_{w=a_1\dots a_n\in A}\prod_{i=1}^{|w|} Da_iD\right),$$

where $A \subseteq T^*$ is a finite set of words, T is an alphabet, D is the Dyck language over an alphabet $T'' \subseteq T$, h is a coding and $T' \subseteq T$.

In a similar way next two results can be obtained. See [37] for more details.

Theorem 4.10. $INS_m^{0,0}DEL_1^{0,0} = INS_m^{0,0}DEL_0^{0,0} \subset CF$, *$m > 0$.*

Theorem 4.11. $INS_1^{0,0}DEL_p^{0,0} \subset REG$ *for any $p > 0$.*

We collect all results above as well as some other results about the computational power of symmetrical insertion-deletion systems in Table 1.

Table 1. Results on symmetrical insertion-deletion systems

Size	Family	Ref.	Size	Family	Ref.
$(1,2,2;1,1,1)$	RE	[14, 32]	$(3,0,0;2,0,0)$	RE	[24]
$(1,2,2;2,0,0)$	RE	[14, 32]	$(1,1,1;2,0,0)$	RE	[32]
$(2,1,1;2,0,0)$	RE	[14, 32]	$(2,0,0;1,1,1)$	RE	[20]
$(1,1,1;1,2,2)$	RE	[35]	$(1,1,1;1,1,1)$	RE	[35]
$(2,1,1;1,1,1)$	RE	[35]	$(2,0,0;2,0,0)$	$\subsetneq CF$	[37]
$(3,0,0;3,0,0)$	RE	[24]	$(m,0,0;1,0,0)$	$\subsetneq CF$	[37]
$(2,0,0;3,0,0)$	RE	[24]	$(1,0,0;p,0,0)$	$\subsetneq REG$	[37]

5 One-sided insertion-deletion systems

In this section we present results about insertion-deletion systems with one-sided context, *i.e.*, of size $(n,m,m';p,q,q')$ where either $m+m' > 0$

and $m * m' = 0$, or $q + q' > 0$ and $q * q' = 0$, *i.e.*, one of numbers in some couple is equal to zero.

One-sided insertion-deletion systems present features common to both contextual and context-free insertion-deletion systems. More precisely, an insertion rule having an empty left (or right) context can be applied any number of times like in the case of context-free rules. However, while a context-free insertion can happen anywhere in the string, in the case of a one-sided insertion the context indicates the place where the insertion can happen. Similar properties are exposed by deletion rules.

Example 5.1.

Consider a system $ID = (T, T, \{a\}, I, \emptyset)$, where $T = \{a, b, c, d\}$ and I is defined as follows: $I = \{(a, b, \lambda), (b, c, \lambda), (c, d, \lambda), (d, a, \lambda)\}$.

Let L be the language generated by ID ($L = L(ID)$). It is clear that L can be defined by the following formulas:

$$L = L_1 \quad L_1 = aL_2^* \quad L_2 = bL_3^* \quad L_3 = cL_4^* \quad L_4 = dL_1^*$$

By substituting L_i, for $2 \leq i \leq 4$ into the description of L_{i-1} we obtain:

$$L_1 = a(b(c(dL_1^*)^*)^*)^*$$

Let $R = \{(abcd)^*(dcb)^*\}$. Consider the language $L'' = L \cap R$. Consider the word $w = abcddbc$ from R. This word is generated in L as follows (we underline the inserted symbol):

$$a \Longrightarrow a\underline{b} \Longrightarrow a\underline{b}b \Longrightarrow ab\underline{c}b \Longrightarrow ab\underline{c}cb \Longrightarrow abc\underline{d}cb \Longrightarrow abc\underline{d}dcb$$

We observe that the generation of the second part of w, the subword dcb, is related to the generation of its first part $abcd$, because every letter is inserted two times: first for the second part and after that for the first part. It is also clear that this is the only way to generate the subword dcb. Moreover, it can be easily seen that such a generation leads to a one-to-one correspondence between $abcd$ and dcb. Now, taking w it is possible to insert a after the first letter d and to continue in a similar manner as before and so on, which gives $w_n = (abcd)^n(dcb)^n$, $n \geq 1$. It is also possible to obtain more copies of $abcd$ by performing insertions of four corresponding letters after d, c, b or a in the first part of w_n. Hence, we finally obtain $L'' = \{(abcd)^i(dcb)^j, j \leq i\}$, which is a non-regular context-free language (by the inverse morphism $\{abcd \to x, dcb \to y\}$ it becomes the well known language $\{x^iy^j, 1 \leq j \leq i\}$). Since the intersection of two regular languages would be regular, we obtain that L is a non-regular context-free language.

5.1 Computational completeness results

Generally, computational completeness proofs for one-sided insertion-deletion systems take into account the above behavior and ensure that additional symbols that potentially can be inserted more than one time are inserted exactly once. This property is usually satisfied by introducing groups of insertion and deletion rules of a special form that can act only if a specified pattern is present in the string. If the pattern is compromised by inserting or deleting more than one additional symbol, then the whole group of rules will fail and non-terminal symbols will remain in the string; moreover, it can be guaranteed that these symbols cannot be eliminated anymore.

The proofs are based on simulation of insertion-deletion systems from Sections 3 and 4 which are known to generate all RE languages. The proof technique is very similar to the one from Theorem 3.5.

We remark that by symmetry, all results for classes $INS_n^{m,m'}DEL_p^{q,q'}$ are also true for classes $INS_n^{m',m}DEL_p^{q',q}$.

We give the sketch of proof for the following theorem.

Theorem 5.2. $INS_1^{1,2}DEL_1^{1,0} = RE$.

Sketch of Proof. The proof is based on the simulation of insertion-deletion systems of size $(1,1,1;1,1,1)$ in normal form. Hence, it is sufficient to show how a deletion rule (a,x,b), with $a,b,x \in V$, may be simulated by using rules of the target system, *i.e.*, insertion rules of type $(a',x',b'c')$ and deletion rules of type (a'',y,λ).

Since the system is in normal form, we may assume that $ab \neq \lambda$. Moreover, we may assume that the system has no insertion rules of the form $(a,b,b), a,b \in V$. If this is the case then we replace every such rule by two insertion rules (a, X, b), (a, b, X), and one deletion rule (b, X, b), where X is a new nonterminal.

A deletion rule $i : (a, x, b)$, where i is the label of the rule, is simulated by two insertion rules (x, X_i, b), (a, D_i, xX_i) and three deletion rules (D_i, x, λ), (D_i, X_i, λ), (a, D_i, λ).

Symbols D_i and X_i act like left and right parentheses that surround x before deleting it. The simulation is performed as follows. First, two insertions are performed:

$$w_1axbw_2 \Longrightarrow_{ins} w_1ax\underline{X_i}bw_2 \Longrightarrow_{ins} w_1a\underline{D_i}xX_ibw_2,$$

and then x is deleted:

$$w_1aD_ixX_ibw_2 \Longrightarrow_{del} w_1aD_iX_ibw_2.$$

At this moment symbols X_i and D_i are deleted:

$$w_1aD_iX_ibw_2 \Longrightarrow_{del} w_1aD_iw_2 \Longrightarrow_{del} w_1abw_2.$$

Hence, every derivation in an insertion-deletion system having the size $(1, 1, 1; 1, 1, 1)$ can be carried out in a system of size $(1, 1, 2; 1, 1, 0)$. On the other hand, we observe that once being inserted, the nonterminals X_i, D_i can be erased only by the rules shown above. Moreover, if they are not deleted, then no symbol can be inserted at the right of a or at the left of b. The rule (D_i, x, λ) can delete at most one x as the pair D_ix is followed by X_ib and $b \neq x$. Thus, there is a one-to-one correspondence between the original and the new systems, which implies that the theorem statement holds. $\square$

In a similar way the results from Table 2 are obtained. We remark that last three results are counterparts of the first three results, where the sizes for insertion and deletion are interchanged. However, in general, systems where insertion parameters are $1, 1, 0$ are simpler than systems having deletion parameters $1, 1, 0$. This is due to the fact that it is easier to control a repeated insertion of symbols by using deletion than a repeated deletion of symbols by using insertion. In the latter case, special "barrier" symbols shall be inserted in order to delimit exactly one symbol to be deleted.

Table 2. Computationally complete one-sided insertion-deletion systems

Size	Ref.	Size	Ref.
(1,1,2;1,1,0)	[20]	(1,1,0;1,1,2)	[26]
(2,0,2;1,1,0)	[20]	(1,1,0;2,0,2)	[26]
(2,0,1;2,0,0)	[20]	(2,0,0;2,0,1)	[26]
(1,2,0;1,0,2)	[21]		

5.2 Non-completeness results

In what follows we show that there are classes of one-sided insertion-deletion systems that are not computationally complete.

We start with the following result.

Theorem 5.3. $REG \setminus INS_1^{1,0}DEL_1^{1,1} \neq \emptyset$.

Sketch of Proof. Consider the regular language $L = \{(ba)^+\}$. We claim that there is no insertion-deletion system ID of size (1,1,0;1,1,1) such that $L(ID) = L$. We can suppose that ID is in normal form.

Let $w_f \in (ba)^+$ be a word generated by ID. Now consider an arbitrary ba block of w_f ($w_f = \beta ba\gamma$, $\beta, \gamma \in (ba)^*$) and take its letter a. Since there are no rules deleting terminal symbols in ID this letter is either inserted by an insertion rule or it was a part of an axiom. We may omit the latter case by taking a derivation that produces a string that is long enough. Now suppose that this letter was inserted using a rule $(z, a, \lambda) \in I$, $z \in V$:

$$w \Longrightarrow^* w_1 z w_2 \Longrightarrow w_1 z a w_2 \Longrightarrow^* \beta ba\gamma = w_f. \tag{1}$$

This means that:

$$\begin{aligned} w_1 z &\Longrightarrow^* \beta b \\ a w_2 &\Longrightarrow^* a\gamma \end{aligned} \tag{2}$$

Now we remark that symbol a might be inserted twice:

$$w \Longrightarrow^* w_1 z w_2 \Longrightarrow w_1 z a w_2 \Longrightarrow w_1 z a a w_2. \qquad (3)$$

From (3) and (2) we obtain:

$$w \Longrightarrow^* w_1 z a a w_2 \Longrightarrow^* \beta b a a \gamma$$

which is a contradiction. □

In way similar to Theorem 5.3 it is possible to show several non-completeness results for one-sided insertion-deletion systems. Table 3 summarizes these results. We remark that systems having smaller parameters, like systems of size $(1,1,0;1,1,0)$ are also not complete.

Table 3. Computationally non-complete one-sided insertion-deletion systems

Size	Witness language	Reference
(1,1,0;1,1,1)	$(ba)^+$	[20]
(1,1,1;1,1,0)	$a^n b^n$, $n \geq 0$	[26]
(1,1,0;2,0,0)	$(ba)^+$	[19]
(2,0,0;1,1,0)	$(ba)^+$	[19]

Moreover, in the case of systems of size $(1,1,0;1,1,0)$ it is possible to show that the language generated by such insertion-deletion systems is a particular subclass of the family of context-free languages.

This class of languages is non-trivial because even a smaller subclass, $INS_1^{1,0}DEL_0^{0,0}$, contains non-regular context-free languages, see Example 5.1.

Theorem 5.4. *$INS_1^{1,0}DEL_0^{0,0} \cap (CF \setminus REG) \neq \emptyset$.*

In [19] it is shown that the effect of deletion rules can be precomputed. This gives the following result.

Theorem 5.5. *$INS_1^{1,0}DEL_1^{1,0} \subset CF$.*

6 Graph-controlled insertion-deletion systems

In previous sections it was shown that there are classes of insertion-deletion systems that cannot generate RE. Making an analogy to context-free grammars, a natural extension of insertion-deletion systems using the graph-controlled or programmed approach can be done. Such model introduces states (or labels of the program) associated to every insertion or deletion rule. The transition is performed by applying corresponding rule and choosing the new state (thus the rule to be applied) among a specific set of rules. Another definition of this model in the style of [30] or [5] can be done. This definition supposes that there are disjoint groups of insertion and deletion rules (corresponding to *membranes* from [30] or *components* from [5]). The transition is performed by firstly choosing and applying one of applicable rules from the current group and switching to the next group indicated in the rule description.

6.1 Formal definition

A *graph-controlled insertion-deletion system* is a construct

$$\Pi = (V, T, A, H, I_0, I_f, R) \text{ where}$$

- V is a finite alphabet,
- $T \subseteq V$ is the *terminal alphabet*,
- $A \subseteq V^*$ is a finite set of *axioms*,
- H is a set of labels associated (in a one-to-one manner) to the rules in R,
- $I_0 \subseteq H$ is the set of *initial labels*,
- $I_f \subseteq H$ is the set of *final labels*, and
- R is a finite set of rules of the form $l : (r, E)$ where r is an insertion or deletion rule over V and $E \subseteq H$.

As it is common for graph controlled systems, a configuration of Π is represented by a pair (w, i), where i is the label of the rule to be applied and w is the current string. A transition $(w, i) \Rrightarrow (w', j)$ is performed if there is a rule $l : ((u, \alpha, v)_t, E)$ in R such that $w \Longrightarrow_t w'$ by the insertion/deletion rule $(u, \alpha, v)_t$, $t \in \{ins, del\}$, and $j \in E$. The result of the computation consists of all terminal strings reaching a final label from an axiom and the initial label, *i.e.*,

$$L(\Pi) = \{w \in T^* \mid (w', i_0) \Rrightarrow^* (w, i_f) \text{ for some } w' \in A,\ i_0 \in I_0,\ i_f \in I_f\}.$$

We will use another rather similar definition for a graph-controlled insertion-deletion system, thereby assigning groups of rules to *components* of the system:

A *graph-controlled insertion-deletion system with k components* is a construct

$$\Pi = (k, V, T, A, H, i_0, i_f, R) \text{ where}$$

- k is the number of components,
- V, T, A, H are defined as for graph-controlled insertion-deletion systems,
- $i_0 \in [1..k]$ is the initial component,
- $i_f \in [1..k]$ is the final component, and
- R is a finite set of rules of the form $l : (i, r, j)$ where r is an insertion or deletion rule over V and $i, j \in [1..k]$.

The set of rules R may be divided into sets R_i assigned to the *components* $i \in [1..k]$, *i.e.*, $R_i = \{l : (r, j) \mid l : (i, r, j) \in R\}$; in a rule $l : (i, r, j)$, the number j specifies the *target component* where the string is sent from component i after the application of the insertion or deletion rule r. A configuration of Π is represented by a pair (w, i), where i is the number of the *current* component (initially i_0) and w is the current string. We also say that w is *situated* in component i. A transition $(w, i) \Rrightarrow (w', j)$ is performed as follows: first, a rule $l : (r, j)$

from component i (from the set R_i) is chosen in a non-deterministic way, the rule r is applied, and the string is moved to component j; hence, the new set from which the next rule to be applied will be chosen is R_j. More formally, $(w,i) \Rrightarrow (w',j)$ if there is $l : ((u,\alpha,v)_t, j) \in R_i$ such that $w \Longrightarrow_t w'$ by the rule $(u,\alpha,v)_t$; we also write $(w,i) \Rrightarrow_l (w',j)$ in this case. The result of the computation consists of all terminal strings situated in component i_f reachable from the axiom and the initial component, *i.e.*,

$$L(\Pi) = \{w \in T^* \mid (w', i_0) \Rrightarrow^* (w, i_f) \text{ for some } w' \in A\}.$$

It is not difficult to see that graph-controlled insertion-deletion systems with k components are a special case of graph-controlled insertion-deletion systems. Without going into technical details, we just give the main ideas how to obtain a graph-controlled insertion-deletion system from a graph-controlled insertion-deletion system with k components: for every $l : ((u,\alpha,v)_t, j) \in R_i$ we take a rule $l : (i, (u,\alpha,v)_t, Lab(R_j))$ into R where $Lab(R_j)$ denotes the set of labels for the rules in R_j; moreover, we take $I_0 = Lab(R_{i_0})$ and $I_f = Lab(R_{i_f})$. Finally, we remark that the labels in a graph-controlled insertion-deletion system with k components may even be omitted, but they are useful for specific proof constructions. On the other hand, by a standard powerset construction for the labels (as used for the determinization of non-deterministic finite automata) we can easily prove the converse inclusion, *i.e.*, that for any graph-controlled insertion-deletion system we can construct an equivalent graph-controlled insertion-deletion system with k components.

We define the *communication graph* of a graph-controlled insertion-deletion system with k components to be the graph with nodes $1,\ldots,k$ having an edge between node i and j if and only if there exists a rule $l : ((u,\alpha,v)_t, j) \in R_i$. In [30], 5.5, special emphasis is laid on graph-controlled insertion-deletion systems with k components whose communication graph has a tree structure, as we observe that the presentation of graph-controlled insertion-deletion systems with k components given above in the case of a tree structure is rather similar to the definition of insertion-deletion P systems as given in [30]; the main differences are that in P systems the final component i_f contains no rules and corresponds with the root of the communication tree; on the other hand, in graph-controlled insertion-deletion system with k components, each of the axioms can only be situated in the initial component i_0, whereas in P systems we may situate each axiom in various different components.

Throughout the rest of this section we shall only use the notion of graph-controlled insertion-deletion systems with k components, as they are easier to handle and sufficient to establish computational completeness in the proofs of our main results presented in the succeeding section. By $GCID_k(ins_n^{m,m'}, del_p^{q,q'})$ we denote the family of languages $L(\Pi)$ generated by graph-controlled insertion-deletion systems with at most k components and insertion and deletion rules of size at most $(n, m, m'; p, q, q')$. We replace k by $*$ if k is not fixed. The letter "G" is replaced by the letter "T" to denote classes whose communication graph has a *tree structure*. Some results for the families $TCID_k(ins_n^{m,m'}, del_p^{q,q'})$ can directly be derived from the results presented in [19, 30], for the corresponding families of insertion-deletion P systems $ELSP_k(ins_n^{m,m'}, del_p^{q,q'})$, yet the results we present in the succeeding section either reduce the number of components for systems with an underlying tree structure or else take advantage of the arbitrary structure of the underlying communication graph thus obtaining computational completeness for new restricted variants of insertion and deletion rules.

Example 6.1.

Consider the following graph-controlled insertion-deletion system $\Pi = (3, T, T, \lambda, H, 1, 1, R)$, with $T = \{a, b, c\}$, $H = \{1, 2, 3\}$ and $R = R_1 \cup R_2 \cup R_3$, where $R_1 = \{1 : ((\lambda, a, \lambda)_{ins}, 2)\}$, $R_2 = \{2 : ((\lambda, b, \lambda)_{ins}, 3)\}$, $R_3 = \{3 : ((\lambda, c, \lambda)_{ins}, 1)\}$.

The system is inserting consecutively a, b and c. Therefore it is clear that $L(\Pi) = \{w \in \{a, b, c\}^* : |w|_a = |w|_b = |w|_c\}$, which is not a context-free language.

We remark that using two nodes, it is possible to similarly generate the non-regular language $L = \{w \in \{a, b\}^* : |w|_a = |w|_b\}$. The communication graph has the form of a tree in this case.

6.2 Results

We start with the following result from [1].

Theorem 6.2. *$PsTCID_*(ins_1^{0,0}, del_1^{0,0}) \subseteq PsGCID_*(ins_1^{0,0}, del_1^{0,0}) = PsMAT$.*

However, in terms of the generated language such systems are not very powerful. Like in the case of context-free insertion-deletion systems there is no control on the position of insertion. Hence, the language $L = \{a^*b^*\}$ cannot be generated, for insertion strings of any size. Hence we obtain:

Theorem 6.3. *$REG \backslash GCID_*(ins_n^{0,0}, del_1^{0,0}) \neq \emptyset$, for any $n > 0$.*

However, there are non-context-free languages that can be generated by such systems (even without deletion). From Example 6.1 we obtain:

Theorem 6.4. *$GCID_*(ins_1^{0,0}, del_0^{0,0}) \setminus CF \neq \emptyset$.*

A more general inclusion holds:

Theorem 6.5. *[1] $GCID_*(ins_n^{0,0}, del_1^{0,0}) \subset MAT$, for any $n > 0$.*

Next theorem shows that graph-controlled insertion-deletion systems are strictly more powerful than ordinary insertion-deletion systems of the same size.

Theorem 6.6. *[21] $TCID_5(ins_1^{1,0}, del_1^{1,0}) = RE$.*

The proof is based on the following idea. Any rule $AB \to CD$ of a type-0 grammar in Kuroda normal form can be simulated in 4 stages: (1) erasing A, (2) erasing B, (3) inserting D and (4) inserting C. Every operation can be done by a dedicated component with the help of an additional symbol that marks the position before A and that is used in all operations. A typical computation may look as follows:

$$w_1ABw_2 \Rrightarrow w_1P_iABw_2 \Rrightarrow w_1P_iBw_2 \Rrightarrow w_1P_iw_2 \Rrightarrow \\ w_1P_iDw_2 \Rrightarrow w_1P_iCDw_2 \Rrightarrow w_1CDw_2$$

Other rules of the grammar can be simulated in a similar manner. We leave technical details that can be consulted in [21].

In a similar way it is possible to obtain a characterization of RE languages by the family $TCID_5(ins_1^{1,0}, del_1^{0,1})$, *i.e.* with contexts for insertion and deletion on different sides. Taking also into account the symmetrical cases we get:

Corollary 6.7. $TCID_5(ins_1^{1,0}, del_1^{0,1}) = TCID_5(ins_1^{0,1}, del_1^{1,0}) = TCID_5(ins_1^{0,1}, del_1^{0,1}) = RE.$

Using a similar technique it is possible to prove following theorems (see [22]).

Theorem 6.8. $TCID_5(ins_1^{1,0}, del_2^{0,0}) = TCID_5(ins_1^{0,1}, del_2^{0,0}) = RE.$

Theorem 6.9. $TCID_5(ins_2^{0,0}, del_1^{1,0}) = TCID_5(ins_2^{0,0}, del_1^{0,1}) = RE.$

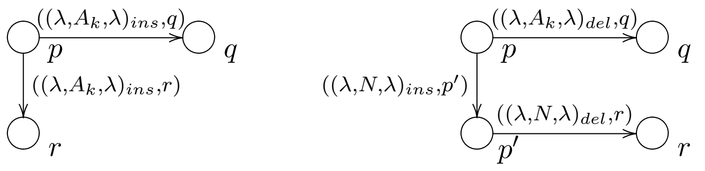
However, in some cases graph-controlled insertion-deletion systems are still not complete.

Theorem 6.10. *[22]* $REG \setminus GCID_*(ins_2^{0,0}, del_2^{0,0}) \neq \emptyset.$

6.3 Graph-controlled insertion-deletion systems with priorities

A further control can be added to graph-controlled insertion-deletion systems by introducing a priority of deletion over insertion, *i.e.*, if deletion and insertion rules are applicable, then one of deletion rules will be chosen. This condition can also be viewed as a particular case of the graph-controlled insertion-deletion systems if the latter have rules with appearance checking. We denote by $TCID_k(ins_n^{m,m'} < del_p^{q,q'})$ the families of languages generated by corresponding classes.

Using priorities it is possible to further decrease the length of contexts needed for computational completeness. It is quite astonishing that insertion-deletion systems that insert or delete one symbol in a context-free manner can generate $PsRE$. In case of general communication graph this is particularly easy to see: jumping to an instruction of a register machine corresponds to switching to the associated component, and the entire construction is a composition of graphs shown in Fig. 1. The decrement instruction works correctly because of priority of deletion over insertion. A configuration $(p, x_1, \cdots, x_n)$ of a register machine is encoded by strings $Perm(pA_1^{x_1} \cdots A_n^{x_n})$.



$((\lambda,A_k,\lambda)_{ins},q)$
p
q
$((\lambda,A_k,\lambda)_{ins},r)$
r
$((\lambda,A_k,\lambda)_{del},q)$
p
q
$((\lambda,N,\lambda)_{ins},p')$
$((\lambda,N,\lambda)_{del},r)$
p'
r

Figure 1. Simulating (p, A_k+, q, r)(left) and (p, A_k-, q, r) (right).

For the tree-like communication graph, the proof is more sophisticated and needs a communication graph depicted at Fig. 2. The main idea is to use a rule $((\lambda, p, \lambda)_{del}, p_1^+)$ if p is an increment instruction or $((\lambda, p, \lambda)_{del}, p_1^-)$ if p is a decrement instruction and redirect the computation to corresponding components that simulate only one instruction of the register machine. This gives:

Theorem 6.11. *$PsTCID_*(ins_1^{0,0} < del_1^{0,0}) = PsRE$.*

Although the above theorem shows that corresponding systems are quite powerful, they cannot generate RE without control on the place where a symbol is inserted ($REG \backslash GCID_*(ins_n^{0,0} < del_1^{0,0}) \neq \emptyset$ for any $n > 0$, see Theorem 6.3). Once we allow a context in insertion or deletion rules, they can do it.

Theorem 6.12. *$TCID_*(ins_1^{0,1} < del_1^{0,0}) = RE$.*

In a similar way the following result can be obtained.

Theorem 6.13. *$TCID_*(ins_1^{0,0} < del_1^{1,0}) = RE$.*

However in this case the proof is more technical and needs additional components, see [1]. A similar can be done with a context-free deletion of two symbols.

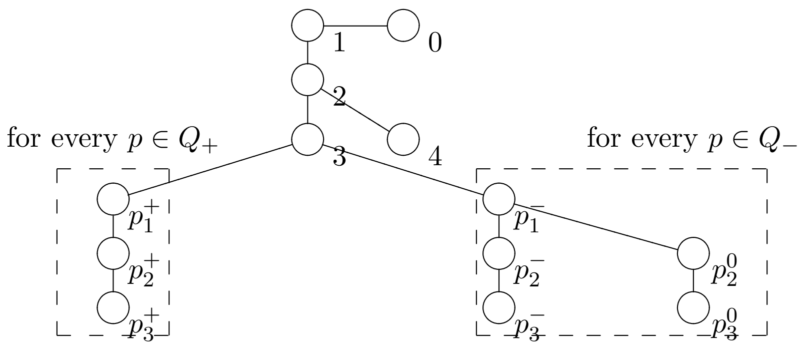


Figure 2. Communication graph for Theorem 6.11. The structures in the dashed rectangles are repeated for every instruction of the register machine.

Theorem 6.14. *$TCID_*(ins_1^{0,0} < del_2^{0,0}) = RE$.*

We mention that the counterpart of Theorem 6.14 obtained by interchanging parameters of insertion and deletion rules is not true, see Theorem 6.3.

7 Using only insertion

In this section we consider systems which only use the operation of insertion, *i.e.*, there are no deletion rules. We shall use the notation $INS_n^{m,m'}$ in order to denote families of languages generated by insertion-only systems. It is known that the classes of *insertion languages* are incomparable with many known language classes. For example, consider a linear language $\{a^nba^n \mid n \geq 1\}$. This language cannot be generated by any insertion system (see Theorem 6.6 in [32]).

In order to be complete it is possible to use some codings to "interpret" the generated strings. In the literature several types of codings were considered. It is possible to consider the following languages as a result for an insertion system I:

1. $h(L(I) \cap R)$, where h is a morphism and R is a special language (as considered in [28, 31]), or

2. $\varphi(h^{-1}(L(I)))$, where h is a morphism and φ is a weak coding (considered in [25, 32, 15]).

We mention that both types of codings are rather simple and can be simulated by a finite state transducer, provided that R is regular. In some cases R is considered to be the Dyck language.

We start with the following representation of regular languages by using insertion systems and star languages. We recall that the family $STAR = \{A^* \mid A \in FIN\}$ of *star* languages is a subfamily of regular languages.

Theorem 7.1. *[31] Any regular language L can be represented in the form $L = h(L' \cap R)$, where h is a weak coding, $L' \in INS_2^{0,0}$, and R is a star language.*

Let W represent the family of weak codings. We mention that the inclusion $REG \subset W(INS_2^{0,0} \cap STAR)$ is proper, because the Dyck language is in $INS_2^{0,0}$.

A similar characterization of context-free languages by the means of insertion systems can be done.

Theorem 7.2. *[18] A language L is context-free if and only if it can be represented in the form $L = \varphi(h^{-1}(L'))$ where $L' \in INS_3^{1,1}$, φ is a weak coding and h is a morphism.*

We remark that it is important to use a coding:

Theorem 7.3. *[32] $INS_*^{1,1} \subsetneq CF$.*

We present below several characterizations of recursively enumerable languages by the means of insertion systems. We start with the following result.

Theorem 7.4. *[15, 27] Each language $L \in RE$ can be written as $L = \varphi(h^{-1}(L'))$, where φ is a weak coding, h is a morphism, and $L' \in INS_3^{3,3}$.*

Sketch. The idea of the proof is to apply “mark and migrate” technique in order to simulate a type-0 grammar. According to this technique, symbols that have been rewritten are marked. In the following a special symbol # called *marking symbol* will be used. We say that a letter a is *marked* in a sentential form waw' if it is followed by #, *i.e.*, $|w'| > 0$, and # is the prefix of w'. For example, in order to simulate a context-free production $A \to BC$, the string $\#BC$ is inserted immediately at the right of A, assuming that A was not marked before. As soon as the derivation of the simulated sentential form is completed, every nonterminal A is marked, and the inverse morphism is applied to the pairs $A\#$.

In order to simulate context-sensitive productions of the form $AB \to CD$, the *migration* of symbols is applied. This means that if a pair AB that should be used by the production is separated by one or more marked symbols, then copies of symbol A are inserted to the right, using the marked symbols as contexts. In this way, the symbol A can migrate to the right and become adjacent to B. When only the terminal symbols are unmarked in the resulted sentential form, the inverse morphism h^{-1} and the weak coding may be applied in order to eliminate marking symbols and nonterminals. □

Corollary 7.5. *[15] Every language $L \in RE$ can be represented in either of the forms $L = L' \setminus R$, $L = L'/R'$, where $L' \in INS_3^{3,3}$, R, R' are regular languages, and $\setminus R, /R'$ denote the left and right quotient with R, R' respectively.*

In a similar way it is possible to obtain a characterization of RE by replacing the inverse morphism h^{-1} by the intersection with a regular language. It is shown in [28] that in order to obtain this characterization it is enough to use strictly $k-$testable languages (denoted by $LOC(k)$), which is a strictly subset of the family of regular languages, for $k \geq 2$. We recall that a language L is a strictly $k-$testable language over T if there are finite sets $Pref, Suf, Int \subseteq T^k$, and for every w, $w \in L$ if and only if (a) the prefix of w of length k belongs to $Pref$, (b) the suffix of w of length k belongs to Suf, and (c) every proper subsequence of w of length k belongs to Int.

Then, the following theorem holds.

Theorem 7.6. *[28] Every language $L \in RE$ can be represented in the form $h(L' \cap R)$, where h is a projection, $L' \in INS_3^{3,3}$, and $R \in LOC(2)$.*

The next theorem considers a different approach showing that insertion systems with context-free rules are quite powerful. Since the *mark and migrate* technique cannot be used in this case, the filtering of sentential forms that have the "proper structure" is performed by an intersection with the Dyck language.

Theorem 7.7. *[31] Every language $L \in RE$ can be represented in the form $L = h(L' \cap \mathcal{D})$, where $L' \in INS_3^{0,0}$, h is a projection, and $\mathcal{D}$ is the Dyck language.*

Finally, we remark that in the case of graph-controlled insertion systems it is possible to decrease the sizes of the contexts.

Theorem 7.8. *[18] Every language $L \in RE$ can be represented in the form $L = \varphi(h^{-1}(L'))$, where φ is a weak coding, h is a morphism, and $L' \in TCID_3(ins_2^{2,2}del_0^{0,0})$.*

8 Bibliographical remarks

Insertion systems, without using the deletion operation, were first considered in [8], however the idea of the context adjoining was exploited long time before by [23]. Context-free insertion systems as a generalization of concatenation were first considered in [9, 10]. A formal language study of both context-free insertion and deletion operations was done in [13], however the operations were considered separately. The articles [7, 11] investigate the power of the insertion and deletion operations. Both operations were first considered together in [16] and related formal language investigations can be found in several places; we mention only [25] and [29]. The biological motivation of insertion-deletion operations leaded to their study in the framework of molecular computing, see, for example, [6], [14], [32], [35]. An interesting study of the deletion operation can be found in [7].

The universality of context-free insertion-deletion systems of size $(2,0,0;3,\ 0,0)$ and $(3,0,0;2,0,0)$ was shown in [24], while the optimality of this result was shown in [37]. The last article suggested to consider the sizes of each context as a complexity measure and not the maximum as it was done before. One-sided insertion-deletion systems were firstly considered in [26] and the graph-controlled variant in [21]. Graph-controlled insertion-deletion systems with priorities were introduced in [1].

Other variants of the insertion operation and different control mechanisms can be found in [13, 12, 4].

References

[1] A. Alhazov, A. Krassovitskiy, Y. Rogozhin, and S. Verlan. P systems with minimal insertion and deletion. In R. Gutiérrez-Escudero, M. A. Gutiérrez-Naranjo, G. Păun, I. Pérez-Hurtado, and A. Riscos-Nunez, editors, *Proc. of Seventh Brainstorming Week on Membrane Computing Sevilla, February 2–6, 2009.* Fénix Editora, Sevilla, 2009, volume I, 9–21. Also accepted to *Theoretical Computer Science.*

[2] A. Alhazov, A. Krassovitskiy, Y. Rogozhin, and S. Verlan. *New Trends in Formal Language Theory Inspired by Natural Computing: Small Size Insertion and Deletion Systems*, Imperial College Press, chapter 9. Mathematics, Computing, Language, and Life: Frontiers in Mathematical Linguistics and Language Theory. 2010, 459–524. In publication.

[3] R. Benne. *RNA-Editing: The Alteration of Protein Coding Sequences of RNA.* Ellis Horwood, Chichester, West Sussex, 1993.

[4] F. Biegler, M. J. Burrell, and M. Daley. Regulated rna rewriting: Modelling rna editing with guided insertion. *Theor. Comput. Sci.*, **387**(2), (2007), 103 – 112. Descriptional Complexity of Formal Systems.

[5] E. Csuhaj-Varjú and J. Dassow. On cooperating/distributed grammar systems. *Elektronische Informationsverarbeitung und Kybernetik*, **26**(1/2), (1990), 49–63.

[6] M. Daley, L. Kari, G. Gloor, and R. Siromoney. Circular contextual insertions/deletions with applications to biomolecular computation. In *SPIRE/CRIWG*. 1999, 47–54.

[7] M. Domaratzki and A. Okhotin. Representing recursively enumerable languages by iterated deletion. *Theoretical Computer Science*, **314**(3), (2004), 451–457.

[8] B. Galiukschov. Semicontextual grammars. *Matem. Logica i Matem. Lingvistika*, (1981), 38–50. Tallin University, (in russian).

[9] D. Haussler. *Insertion and Iterated Insertion as Operations on Formal Languages*. Ph.D. thesis, Univ. of Colorado at Boulder, 1982.

[10] D. Haussler. Insertion languages. *Information Sciences*, **31**(1), (1983), 77–89.

[11] M. Ito, L. Kari, and G. Thierrin. Insertion and deletion closure of languages. *Theor. Comput. Sci.*, **183**(1), (1997), 3–19.

[12] M. Ito and R. Sugiura. n-insertion on languages. In N. Jonoska, G. Paun, and G. Rozenberg, editors, *Aspects of Molecular Computing*. Springer, 2004, volume 2950 of *Lecture Notes in Computer Science*, 213–218.

[13] L. Kari. *On Insertion and Deletion in Formal Languages*. Ph.D. thesis, University of Turku, 1991.

[14] L. Kari, G. Păun, G. Thierrin, and S. Yu. At the crossroads of dna computing and formal languages: Characterizing RE using insertion-deletion systems. In *Proc. of 3rd DIMACS Workshop on DNA Based Computing*. Philadelphia, 1997, 318–333.

[15] L. Kari and P. Sosík. On the weight of universal insertion grammars. *Theor. Comput. Sci.*, **396**(1-3), (2008), 264–270.

[16] L. Kari and G. Thierrin. Contextual insertions/deletions and computability. *Information and Computation*, **131**(1), (1996), 47–61.

[17] S. C. Kleene. Representation of events in nerve nets and finite automata. In C. Shannon and J. McCarthy, editors, *Automata Studies*, Princeton University Press, Princeton, NJ. 1956, 3–41.

[18] A. Krassovitskiy. On the power of insertion P systems of small size. In *Proc. of Seventh Brainstorming Week on Membrane Computing.* 2009, 29–44.

[19] A. Krassovitskiy, Y. Rogozhin, and S. Verlan. Computational power of insertion-deletion (P) systems with rules of size two. Accepted to *Natural Computing.* 2010, in publication.

[20] A. Krassovitskiy, Y. Rogozhin, and S. Verlan. Further results on insertion-deletion systems with one-sided contexts. In C. Martín-Vide, F. Otto, and H. Fernau, editors, *Language and Automata Theory and Applications, Second International Conference, LATA 2008, Tarragona, Spain, March 13-19, 2008. Revised Papers.* Springer, 2008, volume 5196 of *Lecture Notes in Computer Science*, 333–344.

[21] A. Krassovitskiy, Y. Rogozhin, and S. Verlan. One-sided insertion and deletion: Traditional and P systems case. In E. Csuhaj-Varjú, R. Freund, M. Oswald, and K. Salomaa, editors, *International Workshop on Computing with Biomolecules, August 27th, 2008, Wien, Austria.* Druckerei Riegelnik, 2008, 53–64.

[22] A. Krassovitskiy, Y. Rogozhin, and S. Verlan. Computational power of P systems with small size insertion and deletion rules. In T. Neary, D. Woods, A. K. Seda, and N. Murphy, editors, *Proceedings International Workshop on The Complexity of Simple Programs, Cork, Ireland, 6-7th December 2008.* 2009, volume 1 of *EPTCS*, 108–117.

[23] S. Marcus. Contextual grammars. *Revue Roumaine de Mathmatique Pures et Appliques*, **14**, (1969), 1525–1534.

[24] M. Margenstern, G. Păun, Y. Rogozhin, and S. Verlan. Context-free insertion-deletion systems. *Theoretical Computer Science*, **330**(2), (2005), 339–348.

[25] C. Martín-Vide, G. Păun, and A. Salomaa. Characterizations of recursively enumerable languages by means of insertion grammars. *Theoretical Computer Science*, **205**(1-2), (1998), 195–205.

[26] A. Matveevici, Y. Rogozhin, and S. Verlan. Insertion-deletion systems with one-sided contexts. In J. O. Durand-Lose and M. Margenstern, editors, *Machines, Computations, and Universality, 5th International Conference, MCU 2007, Orléans, France, September 10-13, 2007, Proceedings.* Springer, 2007, volume 4664 of *Lecture Notes in Computer Science*, 205–217.

[27] K. Onodera. A note on homomorphic representation of recursively enumerable languages with insertion grammars. *Transactions of Information Processing Society of Japan*, **44**(5), (2003), 1424–1427.

[28] K. Onodera. New morphic characterizations of languages in Chomsky hierarchy using insertion and locality. In A. H. Dediu, A.-M. Ionescu, and C. Martín-Vide, editors, *Language and Automata Theory and Applications, Third International Conference, LATA 2009, Tarragona, Spain, April 2-8, 2009.* Springer, 2009, volume 5457 of *Lecture Notes in Computer Science*, 648–659.

[29] G. Păun. *Marcus Contextual Grammars.* Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[30] G. Păun. *Membrane Computing. An Introduction.* Springer–Verlag, 2002.

[31] G. Păun, M. J. Pérez-Jiménez, and T. Yokomori. Representations and characterizations of languages in Chomsky hierarchy by means

of insertion-deletion systems. *Int. J. Found. Comput. Sci.*, **19**(4), (2008), 859–871.

[32] G. Păun, G. Rozenberg, and A. Salomaa. *DNA Computing: New Computing Paradigms.* Springer, 1998.

[33] G. Rozenberg and A. Salomaa. *Handbook of Formal Languages, 3 volumes.* Springer Verlag, Berlin, Heidelberg, New York, 1997.

[34] W. D. Smith. DNA computers in vitro and in vivo. In R. Lipton and E. Baum, editors, *Proceedings of DIMACS Workshop on DNA Based Computers.* Amer. Math. Society, 1996, DIMACS Series in Discrete Math. and Theoretical Computer Science, 121–185.

[35] A. Takahara and T. Yokomori. On the computational power of insertion-deletion systems. In M. Hagiya and A. Ohuchi, editors, *DNA Computing, 8th International Workshop on DNA Based Computers, DNA8, Sapporo, Japan, June 10-13, 2002, Revised Papers.* 2002, volume 2568 of *Lecture Notes in Computer Science*, 269–280.

[36] A. Takahara and T. Yokomori. On the computational power of insertion-deletion systems. *Natural Computing*, **2**(4), (2003), 321–336.

[37] S. Verlan. On minimal context-free insertion-deletion systems. *Journal of Automata, Languages and Combinatorics*, **12**(1-2), (2007), 317–328.

Sergey Verlan,

Received June 10, 2010

LACL, University of Paris Est,
61, av gen. de Gaulle
94010 Creteil, France
Phone: +33145176600
E–mail: $verlan@univ-paris12.fr$

Institute of Mathematics and Computer Science,
Academy of Sciences of Moldova,
Academiei, 5, MD-2028, Chisinau, Moldova

Simulator of P-Systems with String Replication Developed in Framework of P-Lingua 2.1*

Veaceslav Macari, Galina Magariu, Tatiana Verlan

Abstract

In this paper we present beta version of simulator for P-systems with string replication rules. This simulator is developed according to P-Lingua ideology and principles of the P-Lingua 2.1 development environment. Format for presentation of rules with replications in P-Lingua language is proposed. The already known solutions by means of P-systems with string replication for two problems are used to demonstrate the work with the simulator: the SAT problem and inflections generation problem.

Keywords: P-system, string object, string replication rule, simulator, P-Lingua

1 Introduction

Membrane computing is a rapidly developing scientific trend. The scientists, working in the domain and those who applies results obtained in the domain for practical problems solution, need some tools for verification, demonstration and proof of their theoretical ideas and results. Taking into consideration fast development of the domain and constant appearance of new types of P-systems, the idea to create the development environment, which can be extended as new types of P-systems appear, seems to be successful [1]. Simulators developed in the framework of P-Lingua 2.1 support several types of P-systems: transition P-system model, symport/antiport P-system model, active

*The authors acknowledge the support of the Science and Technology Center in Ukraine, project 4032 “Power and efficiency of natural computing: neural-like P (membrane) systems”.

membranes P-system model (with membrane division rules and with membrane creation rules), probabilistic P-system model, stochastic P-system model. As new models have been included, new simulators have been developed inside the pLinguaCore library, providing at least one simulator for each supported model [2]. P-systems working with string objects using replication rules are not covered by the P-Lingua 2.1 soft, but considered to be powerful means for solution of some problems in different domains (e.g. SAT problem, Hamiltonian Path Problem, some linguistic problems, etc. [3], [4], [5]). So, in this paper we present beta version of simulator for P-systems with string replication rules. This simulator is developed according to P-Lingua ideology and principles of the P-Lingua 2.1 development environment.

2 Model of P-System with String Replication Rules

The formal definition of membrane P-systems with replications is given in [6]. A P-system with string objects and input is a construct

$$\Pi = (O, \Sigma, \mu, M_{l_1}, \ldots, M_{l_p}, R_{l_1}, \ldots, R_{l_p}, i_0), \tag{1}$$

where
O – a finite alphabet;
Σ – a sub-alphabet, $\Sigma \subseteq O$;
μ – a membrane structure defined as a rooted tree with nodes labeled $1, \ldots, p$, the interior of each membrane defines a region;
M_{l_i} – multiset of strings, initially present in region l_i, $1 \leq i \leq p$;
R_{l_i} – set of rules of region l_i, $1 \leq i \leq p$;
l_{i_0} – label of input region, $1 \leq i_0 \leq p$.

A replication rule has the following structure:

$$a \to (u_1, t_1)||(u_2, t_2)|| \ldots ||(u_k, t_n), \tag{2}$$

where
$a \in O^+$,

$u_j \in O^*$, $1 \leq j \leq n$,
$t_j \in \{out, here\} \cup \{in_{l_i} \mid 1 \leq i \leq p\}$, $1 \leq j \leq n$.

It is the string rewriting rule with string replication and target indication.

Application of a rule $a \rightarrow (u_1, t_1)||(u_2, t_2)|| \dots ||(u_k, t_n)$ from the set of rules R_{l_k} transforms any string of the form $w_1 a w_2$ from region l_k into n strings $w_1 u_1 w_2, w_1 u_2 w_2, \dots, w_1 u_n w_2$, where $w_1, w_2 \in O^*$. The resulting string $w_1 u_j w_2$ should be sent to the region specified by t_j:

- if $t_j = here$, the resulting string remains in the region l_k;
- if $t_j = out$, the resulting string is sent out of the region associated with membrane with label l_k to the region immediately outside;
- if $t_j = in_{l_i}$, the resulting string is sent into the region associated with membrane with label l_i, which has to be immediately nested into the region l_k.

(If $k = 1$ we have the usual string rewriting rule with target indication.)

The initial configuration contains the input string(s) over Σ in region i_0 and the multisets of strings M_i in regions i. The rules of the system are applied in parallel to all strings in the system. It may occur that several rules are applicable to a string. But really only one of them can be applied. The rule which will be applied, is determined in non-deterministic way. The computation consists in non-deterministic application of the rules to the strings in regions. If some string can be involved into the computing process, it has to be involved.

The computation halts when no rules are applicable. The result of the computation is the set of all words sent out of the outermost region into environment.

3 P-Lingua Format for P-Systems with String Replication Rules

Since the P-Systems with String Replication had not been implemented within the scope of P-Lingua 2.1, the format for replication rules representation have been developed by the authors. When developing the format, the authors tried to keep principles and notations accepted in the domain and in the framework of P-Lingua 2.1. The authors acknowledge A.Alhazov for fruitful discussions on the question. The developed format is described below.

3.1 Special Symbols

There is a set of symbols of keyboard and reserved words in P-Lingua 2.1, which are used for specific aim when a P-system is described in P-Lingua format [2] (e.g. $@mu$, def, *call*, $\#$). We use the special symbols predefined in P-Lingua with the same purpose as it was designed by the developers of P-Lingua. But, for the P-system with string replication there was a need to introduce some additional symbols and additional functionality for the existing ones.

Thus, the following notations are admitted:

- symbol " ' " is used now to present not only the label of membrane, but the target membrane as well;
- symbol "||" is introduced as the sign for operation of replication;
- the new reserved word "*here*" shows, that the resulting string will remain in the current region;
- the new reserved word "*out*" shows, that the resulting string will be sent out of the current region into the immediately encompassing region.

3.2 Rules with Replication

P-systems with replications operate with strings. To represent a string one has to use a sequence of alphabet symbols which are tied by sign of concatenation and are bracketed in $<$ and $>$. For example, $< c.A\{1,2\}.Beta >$ is the string from three alphabet symbols c, $A_{1,2}$, $Beta$. To write the empty string, we use $< \# >$.

3.2.1 Membrane to Which a Rule Belongs and Target Membrane

To indicate membrane to which a rule belongs, we write symbol "' " and membrane label in the left part of the rule after the string. For example, if a rule belongs to membrane with label 1, we write:

```
<a.b.c>'1 -->
```

To indicate the target membrane, we use the same scheme for the right part of the rule if the target is in_{l_i}. When the target is *out* or *here*, we write (after the string) the symbol "' " and the respective reserved word *out* or *here*. For example, the following rule for membrane with label 1

$$a \rightarrow (abc, here)||(dd, in_2)||(f, out) \tag{3}$$

we present as

```
<a>'1 --> <a.b.c>'here  || <d.d>'2 || <f> 'out
```

3.2.2 Rules Presentation

According to general view of a replication rule $a \rightarrow (u_1, t_1)||(u_2, t_2)||\dots$ $\dots||(u_k, t_k)$, it consists of a left part and a list of right parts with the symbol of replication "||" between them.

$$Left\ part \rightarrow Right\ part\ ||\ Right\ part\ ||\ \dots\ ||\ Right\ part \tag{4}$$

Left part has the following format:

$$s'h, \tag{5}$$

where
s – a non-empty string, e.g. `<a.alpha.bb>`;
h – the label of the membrane to which the rule belongs.

Each Right part can have one of the following three formats:

1)
$$s'here, \tag{6}$$
where

– s is a string, possibly empty, e.g. `<a.beta.abd>` or $< \# >$;

– the reserved word "*here*" shows, that the resulting string will remain in the current region;

2)
$$s'out, \tag{7}$$
where

– s is a string, possibly empty, e.g. `<a.beta.abd>` or $< \# >$;

– the reserved word "*out*" shows, that the resulting string will be sent out of the current region into the immediately encompassing region;

3)
$$s'h, \tag{8}$$
where

– s is a string, possibly empty, e.g. `<a.beta.abd>` or $< \# >$;

– h is the label of target membrane, which specifies the region into which the resulting string will be placed and which has to be immediately nested into the current membrane.

Examples of rules:

1. Suppose that we have the following rule for membrane 1:

$$\beta \cdot e \rightarrow (\phi_1, here) \mid\mid (\phi_2, in_3) \mid\mid (\lambda, out) \tag{9}$$

In Plingua format it looks like:

```
<beta.e>'1 --> <fi{1}>'here || <fi{2}>'3 || <#>'out;
```

2. Suppose that we have the following rule for membrane 2:

$$d \rightarrow (\mu \cdot k, out) \tag{10}$$

In Plingua format it looks like:

```
<d>'2 --> <mu.k>'out;
```

4 Examples of Solution Implementation to Some Problems

4.1 A Solution to SAT

Satisfiability problem (SAT) definition is the following: Given a Boolean expression E in conjunctive normal form (CNF), to decide if there is some assignment to the variables in E such that E is true.

Let us consider a solution to the SAT problem, using P-systems with string replication, given in [4].

Suppose we are given a fomula $E = C_1 \wedge C_2 \wedge \ldots \wedge C_m$ where C_1, C_2, …, C_m are disjunctions, and the variables involved are x_1, x_2, …, x_n. The following P-system with string replication is proposed for the problem solution:

$$\Pi = (V, V, \mu, M_1, M_2, \ldots, M_{m+1}, R_1, R_2, \ldots, R_{m+1}), \qquad (11)$$

where
$V = \{a_i, t_i, f_i \mid 1 \leq i \leq n\}$,
$\mu = [_1\ [_2\ \cdots\ [_{m+1}\ ]_{m+1}\ ]_2\ ]_1$;
$M_{m+1} = \{a_1\}$, $M_i = \{\lambda\}$, $1 \leq i \leq m$,
$R_{m+1} = \{a_i \rightarrow (t_i a_{i+1}, here) \mid\mid (f_i a_{i+1}, here) \mid 1 \leq i \leq n-1\}$
$\quad \cup \{a_n \rightarrow (t_n, out) \mid\mid (f_n, out)\}$,
$R_j = \{t_i \rightarrow (t_i, out) \mid x_i$ is present in $C_j, 1 \leq i \leq n\}$
$\quad \cup \{f_i \rightarrow (f_i, out) \mid \neg x_i$ is present in $C_j, 1 \leq i \leq n\}$, $1 \leq j \leq m$.

For demonstration simplicity we take the case, when the values for m and n are not so large: $m = 3$, $n = 4$, $E = (x_1 + \neg x_2)(\neg x_2 + x_3 + \neg x_4)\neg x_3$.

Code of the program for this problem solution written in P-Lingua (file with extension *.pli*) is the following:

```
@model<string_replication>

def SAT() {

@mu = [ [ [ [ ]'4]'3]'2]'1;
@ms(4) = <a{1}>;
/* rules for membrane 1 */
<t{1}>'1 --> <t{1}>'out;
<f{2}>'1 --> <f{2}>'out;
/* rules for membrane 2 */
<t{3}>'2 --> <t{3}>'out;
<f{2}>'2 --> <f{2}>'out;
<f{4}>'2 --> <f{4}>'out;
/* rules for membrane 3 */
<f{3}>'3 --> <f{3}>'out;
/* rules for membrane 4 */
<a{1}>'4 --> <t{1}.a{2}>'here || <f{1}.a{2}>'here;
<a{2}>'4 --> <t{2}.a{3}>'here || <f{2}.a{3}>'here;
<a{3}>'4 --> <t{3}.a{4}>'here || <f{3}.a{4}>'here;
<a{4}>'4 --> <t{4}>'out || <f{4}>'out;
}

def main() {
call SAT();
}
```

The solution is got in 7 steps ($m + n$ steps).

The initial configuration is shown in Annex 1, Fig. 1. There is the only string a_1 in the region associated with label 4. During the first 4 steps all possible sets of values for variables x_1, x_2, x_3, x_4 are generated in the region associated with label 4.

Step 1. (See Annex 1, Fig. 2.) Due to replication two strings are generated in the region associated with label 4 (for different values of the variable x_1: t_1 – for the value *true* and f_1 – for the value *false*): strings $< t_1.a_2 >$, $< f_1.a_2 >$.

Step 2. (See Annex 1, Fig.3.) Four strings are generated in the region associated with label 4 for different values of the variables x_1 and x_2: $< t_1.t_2.a_3 >$, $< f_1.t_2.a_3 >$, $< t_1.f_2.a_3 >$, $< f_1.f_2.a_3 >$.

Step 3. (See Annex 1, Fig. 4.) Eight strings are generated in the region associated with label 5 for different values of the variables x_1,

x_2 and x_3.

Step 4. (See Annex 1, Fig. 5.) 16 strings are generated for different values of all 4 variables x_1, x_2, x_3, x_4 and they are sent out of the region associated with label 4 into the region associated with label 3.

Step 5. (See Annex 1, Fig. 6.) During the steps 5 – 7 the sets of variables values are filtered: only those sets leave the region, for which the given expression gets the value *true*. At the step 5 the sets are filtered by possible values of the third disjunction C_3 which in our case is negation of the variable x_3: only sets with value *false* for the variable x_3 leave the region associated with label 3.

Step 6. (See Annex 1, Fig. 7.) At the step 6 the sets from the region associated with label 2 are filtered by possible values of the second disjunction C_2 which in our case is $\neg x_2 + x_3 + \neg x_4$: only sets, for which this disjunction is equal to *true*, are sent out of the region associated with label 2.

Step 7. (See Annex 1, Fig. 8.) At the step 7 the sets from the region associated with label 1 are filtered by possible values of the first disjunction C_1 which in our case is $x_1 + \neg x_2$: only the sets, for which this disjunction is equal to *true*, are sent out of the region associated with label 1. And after that the computation halts: the resulting sets, for which the formula $E = (x_1 + \neg x_2)(\neg x_2 + x_3 + \neg x_4)\neg x_3$ gets the value *true*, are in the environment.

4.2 A Solution to the Problem of Inflections Generation in Romanian Language

Inflection in natural language means a change in the form of a word, usually modification or affixation, signalling change in such grammatical functions as tense, voice, mood, person, gender, number, or case. The inflection process goes on according to some set of rules (different rules for different groups of words). The number of such rules varies for different natural languages. The rules can be algorithmized, so the process of inflections generation gets computer aiding.

When modeling the process of inflections generation by P-systems, there appears the possibility to construct in parallel way all the inflections not only for one word, but all the inflections for some group of words which have specific common characteristics, i.e. belong to one inflectional model (e.g. neuter noun, in Romanian, inflectional model 3).

In the article [5] there is defined the P-system with string replication performing the inflection process (including vowel/consonant alternation with the assumption that alternating subword is present in the input word in just one occurrence). According to this definition we construct the P-system for the words of one inflectional model for nouns in Romanian – masculine, inflectional model 5. Below we demonstrate the work of the P-system on the example of two nouns – *"brad"* (engl. *"firtree"*) and *"caid"* (engl. *"kaid"*), which belong to this inflectional model. For better understanding of P-system rules, let us consider the list of inflected words for the noun *"brad"*. Taking into account that the Romanian forms for nominative and accusative cases coincide, as well as for the genitive and dative ones, we consider the reduced paradigm:

brad, brad, brad, bradul, bradului, bradule,
brazi, brazi, brazi, brazii, brazilor, brazilor.

Since one part of inflections is formed without alternation and another part – with alternation, we have two sublists of endings for the nouns of inflectional model 5:

$F_1 = \{, , , ul, ului, ule\}$ and $F_2 = \{i, i, i, ii, ilor, ilor\}$.

So in this case we have the number of sublists of endings $s = 2$; the number of alternations is $m - 1 = 1$, then we have $m = 2$.

Now we can construct the P-system which models the inflection process for considered inflection group according to the definition given in [5].

The words *"brad"* and *"caid"* (followed by the symbol "#") we place into the input region. The P-system looks like the following:

$$\Pi = (O, \Sigma, \mu, brad\#, \lambda, \lambda, R_1, R_2, R_3, 1), \qquad (12)$$

where
$\Sigma = V \cup \{\#\}$,

$O = \Sigma \cup E$,
$\mu = [\ [\ ]_2\ [\ ]_3\ ]_1$,
$E = \{\#_2\} \cup \{A_{11}, A_{12}, A_{21}, A_{22}\}$,
$V = \{a, b, \ldots, z\}$,
$R_1 = \{\# \to A_{12} \mid\mid (\#_2, in_2)\} \cup \{A_{21} \to (\lambda, in_3)\} \cup$
$\{A_{12} \to (\lambda, out) || (\lambda, out) || (\lambda, out) || (ul, out) || (ului, out) || (ule, out)\} \cup$
$\{A_{22} \to (i, out) || (i, out) || (i, out) || (ii, out) || (ilor, out) || (ilor, out)\}$
$R_2 = \{d \to (zA_{21}, out\}$,
$R_3 = \{\#_2 \to (A_{22}, out)\}$.

The P-system has $1 + (s - 1)m$ membranes, then there are 3 membranes in our case. Membrane with label 1 is intended for endings adding. The inflections corresponding to subset F_1 are generated in 2 steps. At the second step they are sent out to the environment. Additionally, in the P-system there are m membranes for each other subset (these membranes are intended for alternations implementation). In our case there are two membranes intended for implementation of single alternation: membranes with labels 2 and 3. The number of steps necessary for performing one alternation is equal to 2. When there are several alternations, they are implemented subsequently inside a word but in parallel for different words. So, the solution is got in $2m + 1$ steps regardless of the number of words placed in the input region; then there are 5 steps for our case.

Code of the program for this problem solution written in P-Lingua (file with extension *.pli*) is the following:

```
@model<string_replication>

def Inflection() {
@mu = [ []'2 []'3]'1;
@ms(1) = <b.r.a.d.diez>,<c.a.i.d.diez>;

 /* rules for membrane 1 */

/* <diez> -> <A{1,2}> || <diez{2}> in 2; */
      <diez>'1 --> <A{1,2}>'1 || <diez{2}>'2;
/* <A{2,1}> -> <#> in 3; */
      <A{2,1}>'1 --> <#>'3;
/* <A{1,2}> -> <#> out || <#> out || <#> out || <u.l> out
|| <u.l.u.i> out || <u.l.e> out; */
      <A{1,2}>'1 --> <#>'out || <#>'out || <#>'out ||
      <u.l>'out || <u.l.u.i>'out || <u.l.e>'out;
/* <A{2,2}> -> <i> out || <i> out || <i> out || <i.i> out
|| <i.l.o.r> out || <i.l.o.r> out; */
      <A{2,2}>'1 --> <i>'out || <i>'out || <i>'out ||
      <i.i>'out || <i.l.o.r>'out || <i.l.o.r>'out;

/* rule for membrane 2 */
/* <d> -> <z.A{2,1}> out;  */
      <d>'2 --> <z.A{2,1}>'out;

/* rule for membrane 3 */
/* <diez{2}> -> <A{2,2}> out;  */
      diez{2}>'3 --> <A{2,2}>'out;
}

def main() {
     call Inflection();

}
```

In Annex 2, Fig. 9 there is the initial configuration of the P-system as it looks in the simulator console.

Step 1. (See Annex 2, Fig. 10.) Since we have two sublists of endings, for each input word two strings are generated – two strings stay in the region 1 and two strings enter the region 2:

- strings in the region 1 are responsible for generation of inflections with endings from subset F_1, which are supposed to be formed without alternation;
- strings in the region 2 are responsible for generation of inflections with endings from subset F_2. One can see that the string is

marked by the special symbol $diez_2$. It has index equal to 2 that shows that the string corresponds to subset F_2 (the subset with index equal to 2).

Step 2. (see Annex 2, Fig. 11.) For strings from region 1 the replicative substitutions are performed and the generated inflections with endings from subset F_1 are sent out to the environment. For strings from region 2 the alternation is carried out (letter *"d"* is changed by letter *"z"*), the marked letter $A_{2,1}$ is added to show that the first alternation was carried out and the resulting strings go to region 1.

Step 3. (see Annex 2, Fig. 12.) The marked letter $A_{2,1}$ in the strings in region 1 is dropped and the resulting strings are placed in region 3.

Step 4. (see Annex 2, Fig. 13.). The symbol $diez_2$ in the strings from region 3 is replaced by the marked letter $A_{2,2}$ and the resulting strings are sent out to region 1. The second index of the marked letter is equal to m, it means that all alternations are carried out and now the endings have to be added.

Step 5. (see Annex 2, Fig. 14.) For strings from region 1 the replicative substitutions are performed and generated inflections with endings from subset F_2 are sent out to the environment. The computation halts: the resulting strings corresponding to all inflections of the input words are present in the environment.

Due to massive parallelism the process of all inflections generation for the words *"brad"* and *"caid"* was implemented in 5 steps. The user is able to place into the input region several words from this inflection group – the number of steps for all inflections generation will be the same.

5 Conclusion and Future Work

The simulator for P-systems with String replications is being developed according to the ideology and in the framework of pLinguaCore. So a set of new classes and methods were added to pLinguaCore in order to support string objects as they are used in P-systems with replication rules. At the same time a set of classes and methods already existing in PLinguaCore were modified to fit string replication P-systems.

The main idea of P-Lingua extension was to use already implemented ideology and to make a separate program flow at the same time. Therefore, new P-system implementation uses base types of P-Lingua framework but the code is separated starting from parsing of P-Lingua program, XML generation, XML loading and simulation with intermediary results and final solution output.

For the implementation of P-Lingua strings replication model, changes in parsing of string objects and rules were made.

A new model definition – "string_replication" – was added in order to make the P-Lingua soft to start working with strings replication P-system.

As far as when applying the rules there is an active work with string objects in P-systems with replications, the internal representation of string objects was changed (compared with that as it is made in PLingua 2.1). String object is represented as a list of objects (not as a simple string), each of which containing: alphabet object name, alphabet object indexes, alphabet object multiplicity. Due to this the work with rules and membrane content during simulation is essentially simplified.

The following classes were added to realize rules with replication: StringsCellLikePsystem, ReplicationLeftHandRule, StringsReplicationRightHandRule, StringsReplicationCellLikeRule, ReadStringsReplicationRule (for reading replication rules from XML file).

The subsequent work supposes implementation of other types of rules for string objects, first of all, splicing operation. Moreover, the more sophisticated means are planned to be offered for user of the simulator: for visualization of P-system (for these models) current configuration and for control of evolving process. Undoubtedly that our experience in the work on strings replication P-systems simulator will serve the good base for elaboration of simulators for other types of P-systems.

Annex 1

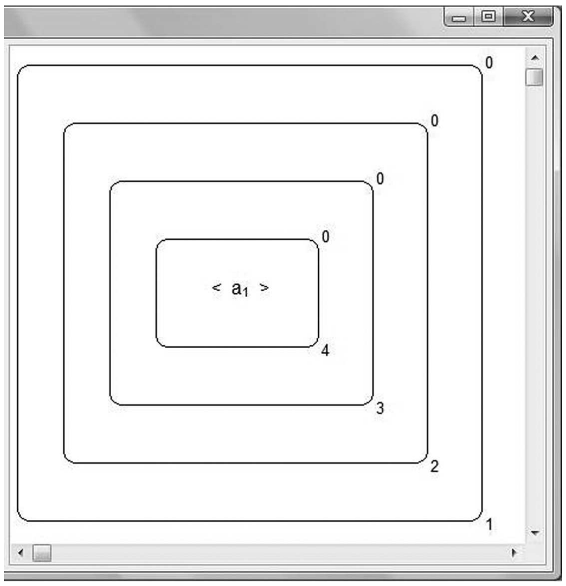


Figure 1. The initial configuration of the P-system for problem SAT solution

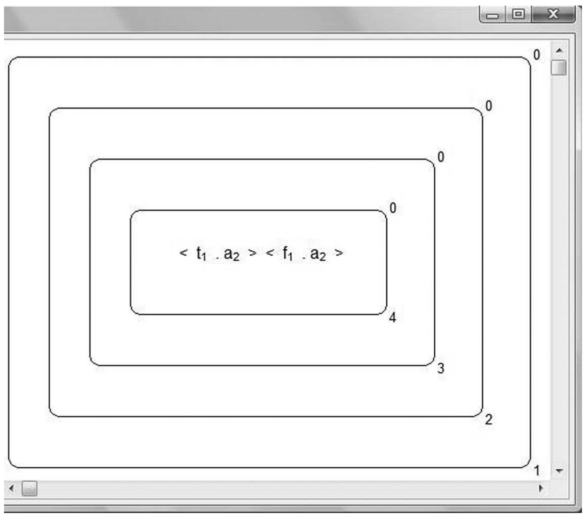


Figure 2. Configuration of the system after Step 1

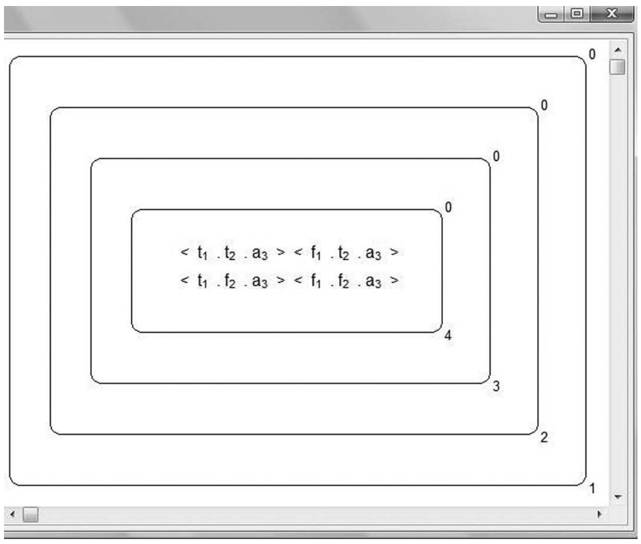
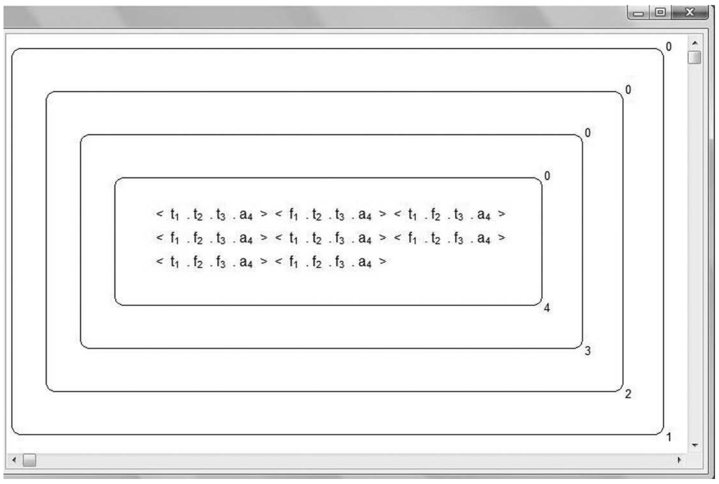
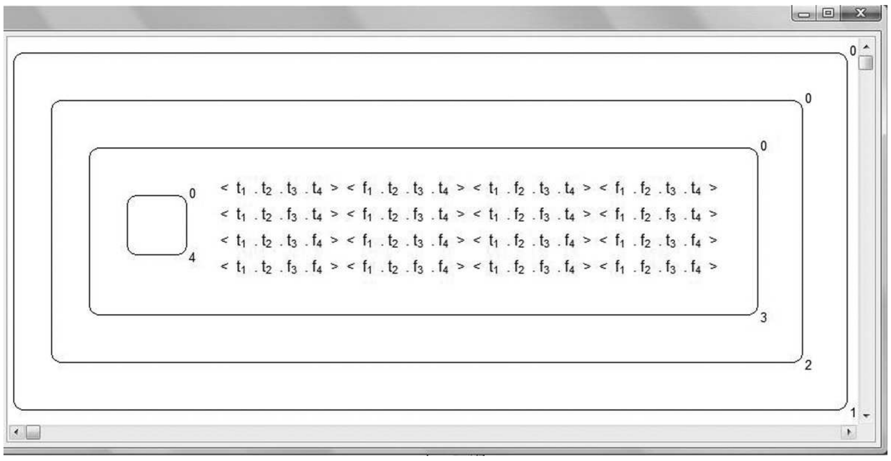

0
0
0
0
< t_1 . t_2 . a_3 > < f_1 . t_2 . a_3 >
< t_1 . f_2 . a_3 > < f_1 . f_2 . a_3 >
4
3
2
1

Figure 3. Configuration of the system after Step 2



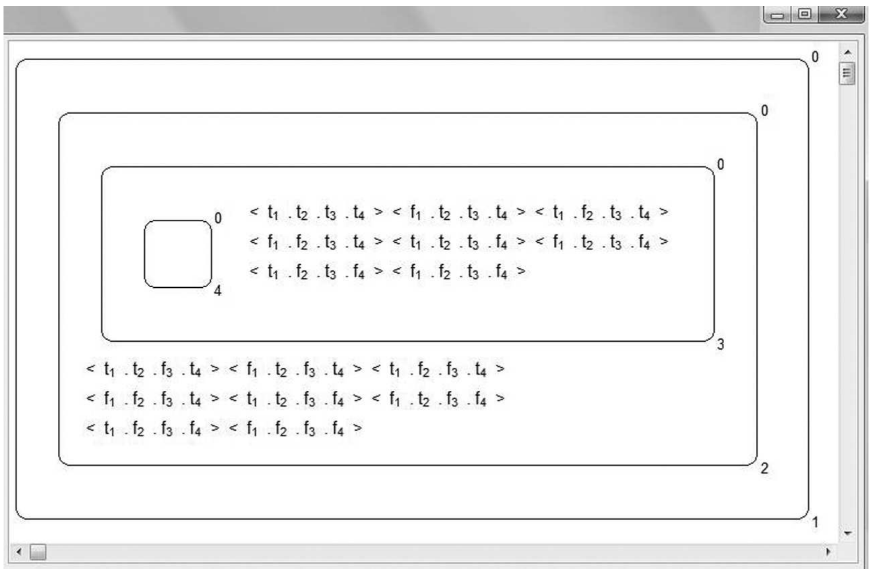
```
< t1 . t2 . t3 . a4 > < f1 . t2 . t3 . a4 > < t1 . f2 . t3 . a4 >
< f1 . f2 . t3 . a4 > < t1 . t2 . f3 . a4 > < f1 . t2 . f3 . a4 >
< t1 . f2 . f3 . a4 > < f1 . f2 . f3 . a4 >
```

Figure 4. Configuration of the system after Step 3



```
< t1 . t2 . t3 . t4 > < f1 . t2 . t3 . t4 > < t1 . f2 . t3 . t4 > < f1 . f2 . t3 . t4 >
< t1 . t2 . f3 . t4 > < f1 . t2 . f3 . t4 > < t1 . f2 . f3 . t4 > < f1 . f2 . f3 . t4 >
< t1 . t2 . t3 . f4 > < f1 . t2 . t3 . f4 > < t1 . f2 . t3 . f4 > < f1 . f2 . t3 . f4 >
< t1 . t2 . f3 . f4 > < f1 . t2 . f3 . f4 > < t1 . f2 . f3 . f4 > < f1 . f2 . f3 . f4 >
```

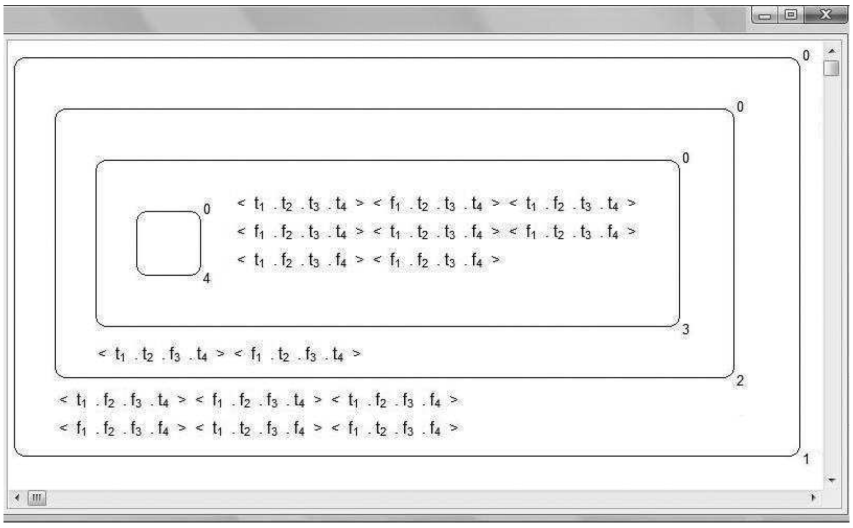
Figure 5. Configuration of the system after Step 4



$< t_1 . t_2 . t_3 . t_4 > < f_1 . t_2 . t_3 . t_4 > < t_1 . f_2 . t_3 . t_4 >$
$< f_1 . f_2 . t_3 . t_4 > < t_1 . t_2 . t_3 . f_4 > < f_1 . t_2 . t_3 . f_4 >$
$< t_1 . f_2 . t_3 . f_4 > < f_1 . f_2 . t_3 . f_4 >$

$< t_1 . t_2 . f_3 . t_4 > < f_1 . t_2 . f_3 . t_4 > < t_1 . f_2 . f_3 . t_4 >$
$< f_1 . f_2 . f_3 . t_4 > < t_1 . t_2 . f_3 . f_4 > < f_1 . t_2 . f_3 . f_4 >$
$< t_1 . f_2 . f_3 . f_4 > < f_1 . f_2 . f_3 . f_4 >$

Figure 6. Configuration of the system after Step 5

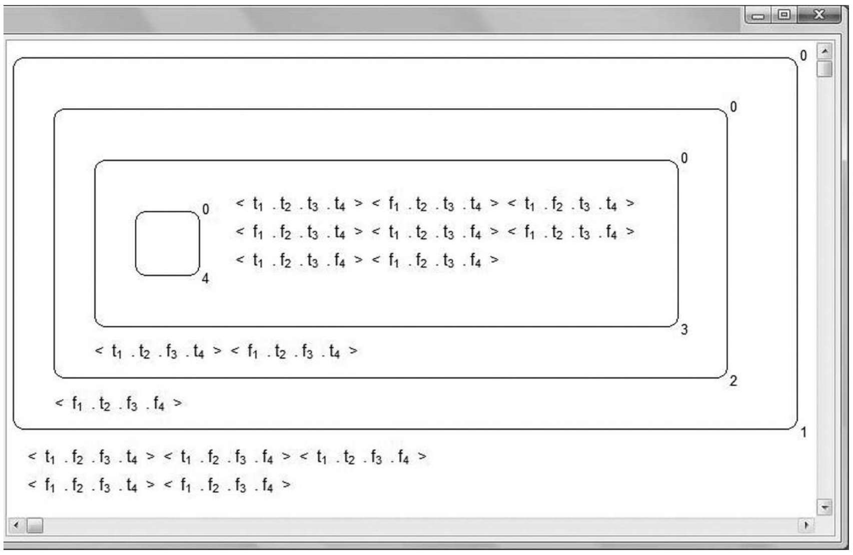


```
< t1 . t2 . t3 . t4 > < f1 . t2 . t3 . t4 > < t1 . f2 . t3 . t4 >
< f1 . f2 . t3 . t4 > < t1 . t2 . t3 . f4 > < f1 . t2 . t3 . f4 >
< t1 . f2 . t3 . f4 > < f1 . f2 . t3 . f4 >

< t1 . t2 . f3 . t4 > < f1 . t2 . f3 . t4 >

< t1 . f2 . f3 . t4 > < f1 . f2 . f3 . t4 > < t1 . f2 . f3 . f4 >
< f1 . f2 . f3 . t4 > < t1 . t2 . f3 . f4 > < f1 . t2 . f3 . f4 >
```

Figure 7. Configuration of the system after Step 6



```
< t1 . t2 . t3 . t4 > < f1 . t2 . t3 . t4 > < t1 . f2 . t3 . t4 >
< f1 . f2 . t3 . t4 > < t1 . t2 . t3 . f4 > < f1 . t2 . t3 . f4 >
< t1 . f2 . t3 . f4 > < f1 . f2 . t3 . f4 >

< t1 . t2 . f3 . t4 > < f1 . t2 . f3 . t4 >

< f1 . t2 . f3 . f4 >

< t1 . f2 . f3 . t4 > < t1 . f2 . f3 . f4 > < t1 . t2 . f3 . f4 >
< f1 . f2 . f3 . t4 > < f1 . f2 . f3 . f4 >
```

Figure 8. Configuration of the system after Step 7

Annex 2

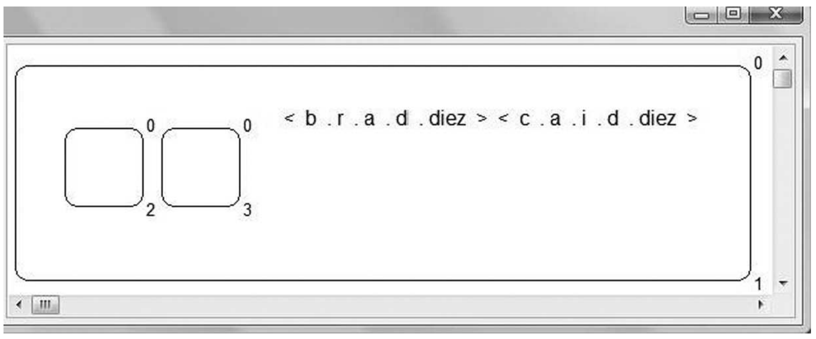


Figure 9. The initial configuration of the P system for inflections generation

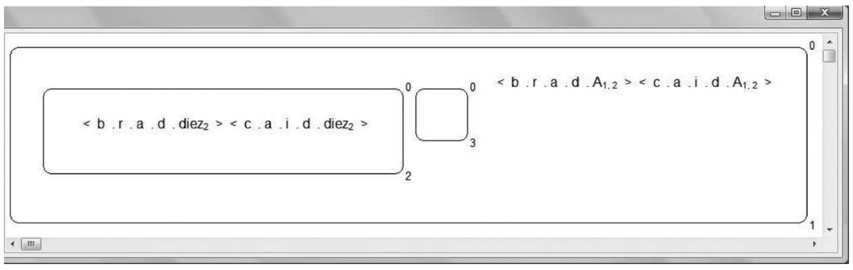
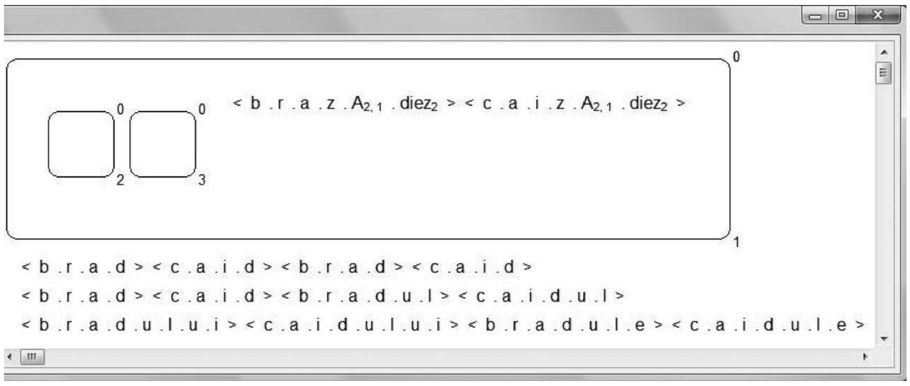

< b . r . a . d . diez$_2$ > < c . a . i . d . diez$_2$ >
< b . r . a . d . A$_{1,2}$ > < c . a . i . d . A$_{1,2}$ >
0
0
0
2
3
1

Figure 10. Configuration of the system after Step 1



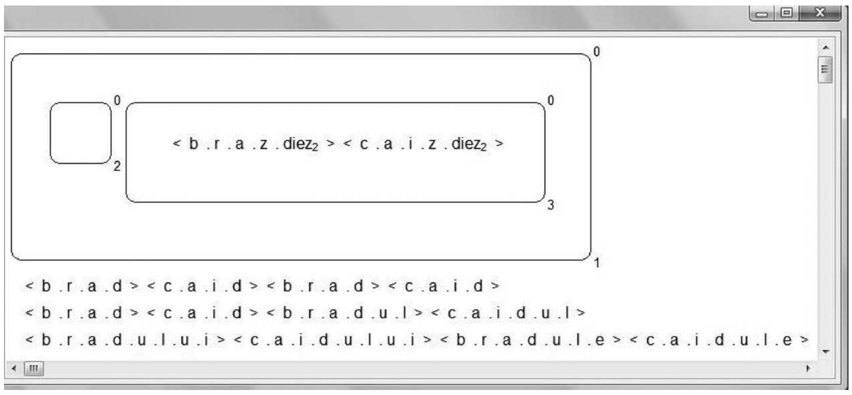
< b .r .a .z .$A_{2,1}$.$diez_2$ > < c .a .i .z .$A_{2,1}$.$diez_2$ >

< b .r .a .d > < c .a .i .d > < b .r .a .d > < c .a .i .d >

< b .r .a .d > < c .a .i .d > < b .r .a .d .u .l > < c .a .i .d .u .l >

< b .r .a .d .u .l .u .i > < c .a .i .d .u .l .u .i > < b .r .a .d .u .l .e > < c .a .i .d .u .l .e >

Figure 11. Configuration of the system after Step 2



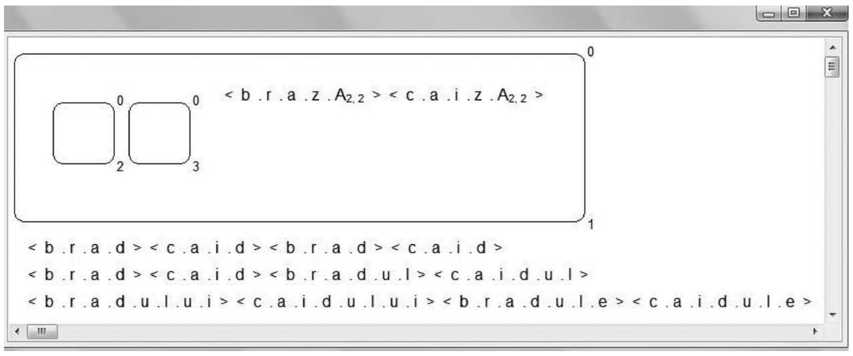
< b .r .a .z .$diez_2$ > < c .a .i .z .$diez_2$ >

< b .r .a .d > < c .a .i .d > < b .r .a .d > < c .a .i .d >

< b .r .a .d > < c .a .i .d > < b .r .a .d .u .l > < c .a .i .d .u .l >

< b .r .a .d .u .l .u .i > < c .a .i .d .u .l .u .i > < b .r .a .d .u .l .e > < c .a .i .d .u .l .e >

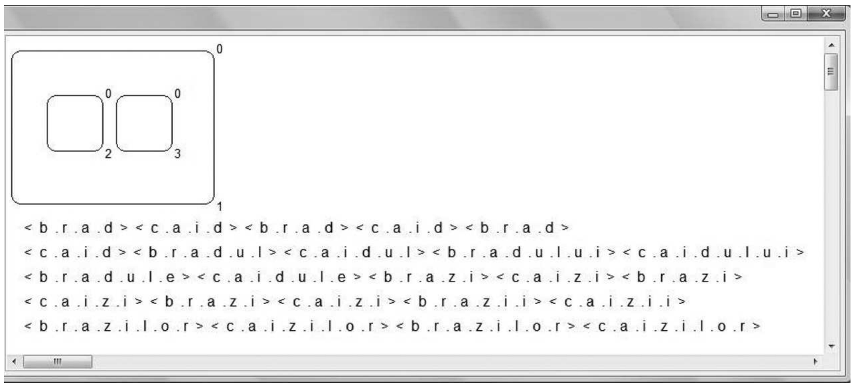
Figure 12. Configuration of the system after Step 3



```
< b .r .a .z .A2,2 > < c .a .i .z .A2,2 >
```

```
< b .r .a .d > < c .a .i .d > < b .r .a .d > < c .a .i .d >
< b .r .a .d > < c .a .i .d > < b .r .a .d .u .l > < c .a .i .d .u .l >
< b .r .a .d .u .l .u .i > < c .a .i .d .u .l .u .i > < b .r .a .d .u .l .e > < c .a .i .d .u .l .e >
```

Figure 13. Configuration of the system after Step 4



```
< b .r .a .d > < c .a .i .d > < b .r .a .d > < c .a .i .d > < b .r .a .d >
< c .a .i .d > < b .r .a .d .u .l > < c .a .i .d .u .l > < b .r .a .d .u .l .u .i > < c .a .i .d .u .l .u .i >
< b .r .a .d .u .l .e > < c .a .i .d .u .l .e > < b .r .a .z .i > < c .a .i .z .i > < b .r .a .z .i >
< c .a .i .z .i > < b .r .a .z .i > < c .a .i .z .i > < b .r .a .z .i .i > < c .a .i .z .i .i >
< b .r .a .z .i .l .o .r > < c .a .i .z .i .l .o .r > < b .r .a .z .i .l .o .r > < c .a .i .z .i .l .o .r >
```

Figure 14. Configuration of the system after Step 5

References

[1] D. Díaz-Pernil, I. Pérez-Hurtado, M.J. Pérez-Jiménez, A. Riscos-Núñez. *A P-lingua programming environment for Membrane Computing*, Proceedings of the 9th Work-shop on Membrane Computing, pp. 155–172 (2008)

[2] M. García-Quismondo, R. Gutiérrez-Escudero, I. Pérez-Hurtado, M.J. Pérez-Jiménez, A. Riscos-Núñez. *An Overview of P-Lingua 2.0*, Tenth Workshop on Membrane Computing (WMC10), Curtea de Arges, Romania, August 24-27, pp. 240–264 (2009)

[3] V. Manca, C. Martín-Vide, Gh. Păun. *On the Power of P Systems with Replicated Rewriting.* Journal of Automata, Languages, and Combinatorics, 6, 3, pp. 359–374 (2001)

[4] Sh.N. Krishna, R. Rama. *P Systems with Replicated Rewriting.* Journal of Automata, Languages and Combinatorics 6(3): pp. 345–350 (2001).

[5] A. Alhazov, E. Boian, S. Cojocaru, Y. Rogozhin. *Modelling Inflections in Romanian Language by P Systems with String Replication.* Computer Science Journal of Moldova, V.17, N.2(50), pp. 160–178 (2009)

[6] Gh. Păun. *Membrane Computing: an Introduction.* Springer (2002).

[7] D. Díaz-Pernil, I. Pérez-Hurtado, M.J. Pérez-Jiménez, A. Riscos-Núñez. *P-Lingua: A Programming Language for Membrane Computing.* In D. Díaz, C. Graciani, M.A. Gutiérrez, Gh. Păun, I. Pérez-Hurtado, A. Riscos (eds.) Proceedings of the Sixth Brainstorming Week on Membrane Computing, Report RGNC 01/08, Fénix Editora, pp. 135–156 (2008)

V. Macari, G. Magariu, T. Verlan,

Received August 22, 2010

Institute of Mathematics and Computer Science,
Academy of Sciences of Moldova,
Academiei 5, Chişinău MD-2028 Moldova
E–mail: *vmacari@gmail.com*, *gmagariu@math.md*, *tverlan@math.md*