Splicing Systems for Universal Turing Machines

Tero Harju[1] and Maurice Margenstern[2]

[1] University of Turku, Department of Mathematics,
20014 Turku, Finland
harju@utu.fi
[2] LITA, EA3097, UFR MIM, Université de Metz,
Île du Saulcy, 57045 Metz, France
margens@sciences.univ-metz.fr

Abstract. In this paper, we look at extended splicing systems (i.e., H systems) in order to find how small such a system can be in order to generate a recursively enumerable language.

It turns out that starting from a Turing machine M with alphabet A and finite set of states Q which generates a given recursively enumerable language L, we need around $2\times|I|+2$ rules in order to define an extended H system $\mathcal{H}$ which generates L, where I is the set of instructions of Turing machine M. Next, coding the states of Q and the non-terminal symbols of $\mathcal{L}$, we obtain an extended H system $\mathcal{H}_1$ which generates L using $|A|+2$ symbols. At last, by encoding the alphabet, we obtain a splicing system $\mathcal{U}$ which generates a universal recursively enumerable set using only two letters.

Keywords: DNA computing, splicing systems.

1 Introduction

Splicing systems are one of the broadest concepts of DNA computing, and so many papers deal with various aspects of what is possible to do with splicing systems that we cannot quote all of them, see [11].

Let us say simply that in most papers, the construction of the language which is computed by the splicing system is the same as considered in [11]. For this approach, let L be a language and denote by $\widetilde{\sigma}(L)$ the result of the application of the rules of a splicing system $\mathcal{S}$ to L. The language which is produced by $\mathcal{S}$ is given by $\bigcup_{i\in\mathbb{N}} L_i$, where $L_0 = A$, the set of axioms, and language L_{i+1} is defined by: $(*)$ $L_{i+1} = \widetilde{\sigma}^{i+1}(L) = \widetilde{\sigma}^i(L)\cup\widetilde{\sigma}\widetilde{\sigma}^i(L)$. This is the case, for instance, in $[3, 4, 2]$, where the question is also approached through multiplicities. In such splicing systems, each generation contains all the information of the previous generations.

In [5], we defined another approach: we *do not* assume that the elements of a generation survive to the next generation. Our approach can be formalised by the following scheme:
$(**)$ $\sigma^{i+1}(L) = \sigma\sigma^i(L)$.

C. Ferretti, G. Mauri and C. Zandron (Eds.): DNA10, LNCS 3384, pp. 149–158, 2005.

We consider finite sets of axioms and finite sets of rules. For the processing operation $(*)$, it was proved by Culik and Harju, [1] (see also Pixton, [12]) that the generated splicing system is regular. For the nonpreserving operation $(**)$, we show that all recursively enumerable languages can be generated.

This result means that the nonpreserving operation introduce some control on the process which explains the possibility of universal computations. This results is to be compared with other results on extensions of splicing systems where various means of control are introduced, in particular, by elimination of molecules which cannot enter a rule, see for instance [8, 9, 15] with time-varying distributed systems and especially for [9] which is the closest to our definition but not exactly the same.

In the first section, we remind the definitions about splicing systems and we give the definition of our approach.

In the second section, we remind our universality results of [5] and we give a method in order to obtain a rather **small** splicing systems which can generate any recursively enumerable language.

2 Splicing Systems and Turing Machine Simulations

A splicing system $\mathcal{S}$ is a triple (Σ, A, R), where Σ is a finite alphabet, A is a finite set of words called **axioms** and R is a finite set of **rules** which we presently define.

A rule of $\mathcal{S}$ is given by four words in Σ^*, say (u_1, u_2u_3, u_4) which we shall display as follows:

$$\begin{array}{c|c} u_1 & u_2 \\ \hline u_3 & u_4 \end{array}$$

In the literature, the same rule is also often displayed as $u_1\#u_2\$u_3\#u_4$.

The application of a rule to a pair of words (w_1, w_2) can be defined as follows:

- if w_1 contains an occurrence of u_1u_2, say $w_1 = x_1u_1u_2y_1$, and w_2 contains an occurrence of u_3u_4, say $w_2 = x_2u_3u_4y_2$, then rule $u_1\#u_2\$u_3\#u_4$ applies to (w_1, w_2) and the result of the application is $(x_1u_1u_4y_2, x_2u_3u_2y_1)$.

This application can be denoted by:

$$\begin{array}{c|c} x_1u_1 & u_2y_1 \\ \hline x_2u_3 & u_4y_2 \end{array} \vdash_r \left\{ \begin{array}{l} x_1u_1u_4y_2 \\ x_2u_3u_2y_1 \end{array} \right.$$

- if w_1 does not contain u_1u_2 or if w_2 does not contain u_3u_4, then we say that rule $u_1\#u_2\$u_3\#u_4$ does not apply to (w_1, w_2).

When rule r applies to (w_1, w_2) with (y, z) as a result, we denote this by $(w_1, w_2) \vdash_r (y, z)$.

The **language generated** by $\mathcal{S}$ which we denote by $\mathcal{L}(\mathcal{S})$ is defined as follows:

$$\mathcal{L}(\mathcal{S}) = \bigcup_{n \in I\!N} \sigma^n(A)$$

where $\sigma^0(A) = A$ and
$\sigma(\mathcal{M}) = \{w\ ;\ \exists w_1, w_2, z \in \mathcal{M}, \exists r \in R(w_1, w_2) \vdash_r (w, z) \ \text{ or } \ (w_1, w_2) \vdash_r (z, w)\ \}$,
with $\mathcal{M}$ running over $\sigma^n(A)$, compare with $(**)$.

We also consider **extended** splicing systems.

They are obtained by changing the alphabet and the definition of the generated language in the following way. Now, we consider that $\Sigma = T \cup N$, where T is called **terminal alphabet** and N is called the **set of non-terminal symbols** and the system itself, say $\mathcal{E}$, is denoted by (T, N, A, R). Also, the language generated by $\mathcal{E}$ is defined by $\mathcal{L}(\mathcal{E}) = \mathcal{L}(\mathcal{S}) \cap T^*$, where $\mathcal{S} = (T \cup N, A, R)$.

As indicated with full details in [5], extended splicing systems can simulate deterministic Turing machines with a single head and a single bi-infinite tape.

The idea of the simulation is that going from a current configuration to the next one in the computation of Turing machine is a **local** transformation of the current configuration. As splicing is also local to the sites where the rule operates, we may expect to represent one step of the computation of a Turing machine by the application of a splicing rule.

This is the case and our report [5] gives an explicit set of rules for that purpose.

We have just to mention a point which is connected with the simulation within **finite** strings of a Turing configuration which is a finite part of the **infinite** tape.

This means that the treatment of the ends of a word is different for Turing machines and splicing systems. We solved this problem by introducing a special marker which indicates the ends of the Turing configuration. When the signal which simulates the Turing machine head meats the marker, it removes it by one square further, putting in its place a blank. This is not difficult to implement in splicing rules, see [5].

In [5], we proved the following result:

Theorem 1. *For any RE language $\mathcal{M}$, there is an extended splicing system $\mathcal{E}$ such that $\mathcal{L}(\mathcal{E}) = \mathcal{M}$.*

Using the schemes for splicing rules introduced in [5], we could prove that:

Corollary 1. *Let $\mathcal{M}$ be an RE language on $\{a, b\}$ which is simulated by a Turing machine M with k instructions. There is a splicing system having $2k + 32$ rules which generates $\mathcal{M}$.*

Also, using a coding of the alphabet of the recursively enumerable set by only two letters, we obtained in [5]:

Corollary 2. *For each RE language $\mathcal{M} \subseteq \Gamma^*$, there is a non-extended splicing system $\mathcal{S}$ and a coding $c : \Gamma^* \mapsto \{0, 1\}^*$ such that*

$$\mathcal{M} = c^{-1}\Big(L(\mathcal{S})\Big)$$

3 Universality Results

From theorem 1 we know that for each RE language, we can construct an extended splicing system which generates it.

In [5], we proved that there is a uniform way to do this by using universal Turing machines:

Theorem 2. *There is an extended splicing system $\mathcal{E} = (T, N, A, R)$ and an encoding c over T^* such that for any RE $\mathcal{M}$, there is a word $w_{\mathcal{M}}$ such that the new system $\mathcal{E}' = (T, N, A \cup \{w_{\mathcal{M}}\}, R)$ generates $c^{-1}(\mathcal{M})$.*

The idea of the proof is to simulate a universal Turing machine U and $w_{\mathcal{M}}$ is a suitable encoding of a Turing machine M which generates $\mathcal{M}$.

We shall follow the same idea in a somehow more sophisticated pattern in order to find an extended system which generates recursively enumerable languages with a **small** number of rules.

In the late fifties and early sixties of the previous century, there was a race to find the smallest universal Turing machines. A long pause was put on this race by the results of Yurii Rogozhin who, in 1982, devised seven very small universal Turing machines, see [13, 14]. From these machines, we take the one which has seven states and four symbols. It is usually denoted by $UMT(7, 4)$.

These machines simulate tag systems which are proved to be universal, see [10]. Tag-systems are defined as follows. We have an alphabet A, a positive number p and a mapping $a_i \mapsto P_i$ from A into A^*, P_i's being called the **productions**.

One step of computation is defined by the following process where w is the **current word**:

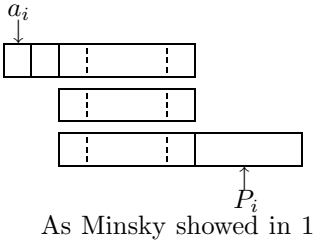
let a_i be the first letter of w;
erase first p letters of w, and let w' be what remains;
append P_i to w'.

The computation starts again with $w'P_i$ as the new current word. It halts by meeting a **halting** letter in first position. We may assume that there is a single halting letter and we denote it by **!**.

As an example, consider tag-system P on {a,b,c} with current word bbb:

P:	applied to bbb:
a ⟶ b	bb|b
b ⟶ bc	|bb|c
c ⟶ !	| |c !

The general scheme for $p = 2$ can be simulated as follows:



As Minsky showed in 1962 that tag-systems can simulate Turing machines, and as tag-systems can be easily simulated by Turing machines, this opened

the way to very small universal Turing machines. Below, the first seven lines of Table 1, except the second instruction, are taken from Rogozhin's $UTM(7,4)$.

Table 1. Program of machine U'

	0	**1**	b	c
1	L	$\mathbf{0}\,L$	$c\,R\,2$	$b\,L$
2	$\mathbf{1}\,R$	$\mathbf{0}\,L\,1$	$c\,R$	$\mathbf{1}\,R\,5$
3	$\mathbf{1}\,L\,4$	R	$c\,R$	$b\,R$
4	$\mathbf{1}L\,7$	L	$c\,L$	$b\,L$
5	$c\,L\,4$	R	$c\,R$	$b\,R$
6	$R\,5$	$\mathbf{0}\,R$	R	$\mathbf{0}\,R\,1$
7	$R\,3$	$R\,8$	$L\,6$	_
8		$\mathbf{0}\,R$	R	$\mathbf{0}\,R\,9$
9	$c\,R\,\mathbf{!}$	R		R

Without entering in the technique of the simulation, in a first stage, the machine locates the production to be appended to the current word. To do so, the encoding of the letters in the current word contain as many symbols as there are markers in the encoding of the productions between the letter and its production. To give a better idea to the reader, the tape of $UTM(7,4)$ looks like this:

$$\mathbf{10}P_n \ldots P_i \ldots P_1 P_0 L_1 c L_2 c \ldots c L_k$$

where $L_i = \mathbf{1}^{N_i}$ with N_i being the number of b's to be marked between L_i and its corresponding production, and where P_i is a concatenation, in reverse order, of codes of the letters in the form $b\mathbf{00}^{N_i}$, and P_i itself starts with an additional b.

When production P_i, corresponding to L_1 is located, the tape looks like this:

$$\mathbf{10}P_n \ldots P_i \boxed{\ldots P_1 P_0 L_1} \underset{\triangle}{c} L_2 c \ldots c L_k$$

where, inside the frame, b's are replaced by c's and 0's by 1's.

We are interested in the aspect of the tape when the halting letter is the first one. In that case, the tape looks like this:

$$\mathbf{10}\boxed{P_n \ldots P_i \ldots P_1 P_0 L_1} \underset{\triangle}{c} L_2 c \ldots c L_k$$

with the same transformation of the tape as previously in the part of the tape within the frame. The triangle indicate the position of the head which scans a c under state 2. Next, the head goes on to the right under state 5 until it meets

the leftmost **0** which it replaces by a c. Then, the head goes back to the left under state 4, leaving **1**'s unchanged and transforming b's into c's and then c's into b's, until the head meets the rightmost **0**. It replaces **0** by **1** and goes to the left under state 7. There it meets a **1**, which means that the computation of the tag system is completed. Table 1 appends instructions to Rogozhin's $UTM(7,4)$, giving us a new machine U', in order to restore the encoding of the tag system and to prepare the collection of a word which belongs to the resulting language. This is why crossing back the configuration, the head arrives to its right-hand end where it halts on the leftmost **0**.

In terms of extended splicing systems, this means that we arrive to a word whose right-hand end is of the form $q_f\#$. Rules (B_1), (B_2), (s_1) and (s_2) allows us to remove # from the right-hand end of the word and to go leftwards until the rightmost **0** is met: what is on the right hand of this letter is the word to append to the language. This is performed by rules similar to rule (C_1) with $*$ replaced by **0**. We may notice here that as the result of the tag-system has always at least three symbols, we do not need rule (C_2).

At this point, the rôle of rule (I_1) is replaced by a much more complicate process. First, we have to destroy all **0**'s which we meet, still going to the left, until the rightmost occurrence of b is reached. We append a new symbol, d, to the right hand of b and we start the program of a new Turing machine V. Let us call $\mathcal{T}$ the encoding of the tag-system on the tape: its right-hand end is d and its left-hand end is **1** as far as in between, there are only **0**'s and b's. The rôle of V is to put the encoding of the next initial current word on the right hand of $\mathcal{T}$.

Recall that the initial current word corresponds to the encoding of configuration (3). Recall that the tag system to which we apply U' does not directly simulate Turing machine K in the proof of theorem 1. It simulates it through a register machine R_1 with two registers which simulates a register machine R_0 with three registers simulating K. The simulation of R_0 by R_1 entails an exponential slowdown: if the registers of R_0 contain non-negative integers x, y and z, the registers of R_1 contain $2^x3^y5^z$ and 0 at an appropriate step. As the initial configuration of R_0 is $x, 0, 0$, the initial configuration of R_1 is $2^x, 0$. Now, if we encode configuration (3) which is essentially n in unary, we get that $x = 2^n$. In order to avoid a double exponential, we encode n in binary. Accordingly, machine K must be replaced by a machine K' with a two letter alphabet which does the same as K and in which n is encoded in binary: **0**, **1** and $*$ of the tape of the new machine K are respectively encoded as **10**, **11** and **01** and we reserve **00** to encode the blank of K. As **10** cannot be confused with the blank of the tape of machine K, we use $*$ as a separator. Call $*$ the blank cell which is the left neighbour of the leftmost **1** on the tape of K'. We also can assume that K' does not go to the left of *ast*. It is enough to guarantee this for the successor in the proof of Kleene's theorem. As there is no difficulty, we leave the easy details to the reader.

Accordingly, we may assume that $x = n$. The next value of x will be $n+1$. Let $u = 2^x$. This means that the new value of u is $2^{n+1} = 2.2^n$.

Turning now to the tag-system simulation, u, v, the contents of the registers of R_1 are encoded as $Aa(aa)^uBb(bb)^v$. We may assume that the current word which corresponds to configuration (3) is $Aa(aa)^uBbbb$ with $u = 2^n$, where A, a, B and b are fixed letters of the alphabet of the tag system. We may assume that a, A, b and B are respectively encoded by $\mathbf{1}c$, $\mathbf{111}c$, $\mathbf{11}c$ and $\mathbf{1111}c$.

Summarising all this information, when machine U' halts, the initial current word of the tag-system goes from

$$\mathbf{111}c\mathbf{1}c(\mathbf{1}c\mathbf{1}c)^u\mathbf{1111}c\ \mathbf{11}c\mathbf{11}c\mathbf{11}c$$
$$\mathbf{111}c\mathbf{1}c(\mathbf{1}c\mathbf{1}c)^{2u}\mathbf{1111}c\ \mathbf{11}c\mathbf{11}c\mathbf{11}c.$$

For that purpose, we keep a copy of $(\mathbf{1}c)^u(d\mathbf{1})^{10}$ which we put on the left hand of the encoding $\mathcal{T}$ of the tag-system on the tape. We introduce an additional letter, d, which we use in such a way that during the program of multiplication by 2, we use motions between two $\mathbf{0}$'s. Letter d is also used to encode a fixed part of the initial current word which corresponds to the encoding of A and of $Bbbb$.

The installation of the next initial current word is performed by machine V whose table is displayed by Table 2.

Machine V marks with d the rightmost $\mathbf{0}$ of $\mathcal{T}$ which also indicates its right-hand end: state 1. Then, it goes to the left-hand end of the configuration, marking by $\mathbf{1}$ the $\mathbf{0}$'s of T: state 2; the motion goes on with state 3 until the leftmost $\mathbf{0}$ is reached. State 4 is a test for loop (L_1) which erases $(\mathbf{1}c)^u$ while copying it onto $(\mathbf{1}c)^{2u}$. On state 4, the head of V erases a $\mathbf{1}$ to meet a c or a d. If it meets a d, it is the end of (L_1). If it is a c, a new round of the copying action is to be performed. State 5 puts the head to the other end of the configuration, on the leftmost $\mathbf{0}$. There, it writes down $(\mathbf{1}c)^2$, using the instructions of states 21 up to 23 corresponding to $\mathbf{0}$. State 23 sends the head to the left. The motion is controlled by state 6 which halts it on the rightmost $\mathbf{0}$ which marked a c. When $\mathbf{0}$ is reached, the head moves by one step to the right and a new test occurs.

When loop (L_1) is completed, a new loop, (L_2) occurs: it copies $(d\mathbf{1})^{10}$ on the left hand of $\mathcal{T}$ to $(\mathbf{1}c)^{10}$ at the right-hand end of the configuration. The test of the loop is performed by state 9, the motion to the right is controlled by state 7, the motion to the left by state 8. The writing on $\mathbf{0}$ is made by states 7 and 28. The latter state calls state 8 which transfers the control to the test of state 9 when the head has performed its motion to the left. The test looks whether it reads d, in which case a new cycle of copying. If it reads b, the head knows that this second round of copying is completed. State 11 puts the head on the right-hand end of the tape and from there, the instructions of states 21 up to 27 scan $\mathbf{1}$'s and c's of this end of the tape and they change it such that $Bbbb$ appears at the end of the encoding. When this is done, state 27 marks with d the first letter of the encoding of $Bbbb$ which is a $\mathbf{1}$. Next, state 28 brings back the head on mark d of the right-hand end of $\mathcal{T}$: the head corrects the tape in such a way that its beginning encodes A, which is performed by state 29 and with the help of states 31 and 32, the head marks by d the last letter of the encoding of Aa which is a c. Then, the head goes on to right in a cycle of copying $\mathbf{1}c$ on the left hand of $\mathcal{T}$ for the computation of the following initial current word. This is

performed by states 41 up to 44 where state 41 realises the test which controls the loop. The loop is completed when it meets d under state 41. It then restores the **1** which was marked by d and the machine goes to the left to the other d which marks the rightmost c of Aai: state 52. Then, it restores the **0**'s which are with b in $\mathcal{T}$.

The occurrence of d in state 53 indicates that the head is on the left of $\mathcal{T}$ and that it has wrongly changed a **1** into **0**. This is corrected by state 54 which also changes into **1** the leftmost letter of $\mathcal{T}$. The, state 55 brings the head to d where it restores **0** which should be there and it stops. We can now give the control to machine U'.

Table 2 could be made more compact but it would be much more difficult to understand it.

Table 2. Program of machine V

	0	**1**	b	c	d
1	$d\,L\,2$				
2	**1** L	L	L		$L\,3$
3	$R\,4$	L		L	L
4		**0** R		**0** $R\,5$	**0** $R\,7$
5	**1** $R\,21$	R	R	R	R
6	$R\,4$	L	L	L	L
7	**1** $R\,28$	R	R	R	R
8	$d\,R\,9$	L	L	L	L
9		R	$R\,11$		**0** $R\,7$
11	$L\,21$	R	R	R	R
21	$c\,R\,22$	$L\,22$		L	
22	1 $R\,23$	$c\,L\,23$		**1** L	
23	$c\,L\,6$	$L\,24$		**1** L	
24		$L\,25$		L	
25		$c\,L\,26$		**1** L	

	0	**1**	b	c	d
26		$L\,27$		**1** L	
27		$d\,L\,28$		**1** L	
28	$c\,L\,8$	L		L	$R\,29$
29		R		**1** $L\,10$	
31		R		$R\,32$	
32		R		$dR\,41$	
41		R		**0** $R\,42$	**1** $L\,51$
42	$c\,L\,43$	L	L	L	L
43	**1** $R\,44$				
44	$c\,R\,41$	R	R	R	R
51		L		L	$c\,L\,52$
52		L		L	$c\,L\,53$
53		**0** L	L		$R\,54$
54	**1** R		R		**0** R **!**

Now, let us count he number of rules which are needed.
Turing machine U' has 33 instructions, the halting one included. Machine V has 93 instructions, halting also included. This makes 126 instructions, hence, 252 rules. To detach the word which is computed by the tag system, we need rules

(B_1), (B_2), (s_1), (s_2) and an adaptation of rule (C_1) to this setting. As (B_2), (s_1), (s_2) and (C_1) are schemes of rules and as the alphabet has 5 letters, this gives us 21 rules.

To removing **0**'s after detaching the word which is produced, we need rules similar to (B_1) and $B(2)$ which means again 6 rules. We need an additional rule similar to (I_1) to transfer the control to V.

Accordingly, we need 28 rules for tasks on the simulated Turing tape which are typical for splicing systems and which cannot be performed by Turing machines. In total, we need 280 rules.

This gives us the following result:

Corollary 3. *There is an extended splicing system* $\mathcal{E} = (T, N, A, R)$ *such that* $T = \{\mathbf{0},\mathbf{1}\}$*, such that* R *contains* 280 *rules an such that there is an encoding* c *over* T^* *such that for any RE* $\mathcal{M}$*, there is a word* $w_{\mathcal{M}}$ *such that the new system* $\mathcal{E}' = (T, N, A \cup \{w_{\mathcal{M}}\}, R)$ *generates* $c^{-1}(\mathcal{M})$*.*

4 Conclusion

The last result of the paper points at how low the descriptional complexity of a universal splicing system can be. Using the well known Turing simulation of tag systems which allowed to obtain very small universal Turing machines, here, we obtained a rather small splicing system which is able to generate any recursively enumerable set. We did not completely investigate this aspect. It could also be of interest to find out how complex are the rules which are involved in corollary 3.

Also, notice that our result is highly *sequential* in its spirit, despite the highly parallel potentiality of the model. We think that this is in connection with our introductory remark about the sensibility of the computational of splicing systems with respect to the definition of the generation of the language. The well known regularity result of extended splicing systems with finitely many axioms and finitely many rules is also probably connected with the fact that the lack of control prevents the realisation of highly sequential processes. This relation between sequentialisation and universality was already noticed in [8, 9, 15]. However, these papers dealt with several splicing systems and the discussion was more on the communications between the systems. With the result on universality which holds also for non-extended splicing systems, we think that the present paper throws a new light on this connection.

Acknowledgement

Both authors acknowledge the help of IST-2001-32008 **MolCoNet** *project* for giving the best conditions to their cooperation.

References

1. Culik, K., II and Harju, T., Splicing semigroups of dominoes and DNA, *Discrete Appl. Math.*, **31**, (1991), 261–277.
2. Ferretti C. and Frisco P., Mauri G., Simulating Turing machines through extended *mH* systems, *Computing with Bio-Molecules. Theory and Experiments*, (G. Păun, ed.), Springer Verlag, (1998), 221–238.
3. Freund R., Kari L., and Păun G., DNA computation based on splicing: the existence of a universal computer, *Theory of Computing Systems*, **32**, (1999), 69–112.
4. Ferretti, C., Mauri, G., Kobayashi, S., and Yokomori, T. , On the universality of Post and splicing systems. Universal machines and computations (Metz, 1998). *Theoret. Comput. Sci.*, **231**, (2000), 157–170.
5. Harju, T. and Margenstern, M. Remarks on the universality of splicing systems, TUCS Report (2004).
6. Kleene S.C., A note on recursive functions, *Bulletin of the American Mathematical Society*, **42**, 544bis–546, (1936).
7. Margenstern M., Turing machines: universality and limits of computational power, in *Formal Languages and Applications*, C. Martin-Vide, V. Mitrana, Gh. Păun, ed., Springer-Verlag, to appear.
8. Margenstern M., Rogozhin Yu. About Time-Varying Distributed *H*-systems, *Lecture Notes in Computer Science*, **2054**, (2000), 53–62.
9. Margenstern M., Rogozhin Yu., Verlan S., Time-varying distributed H sytems with parallel computations: the problem is solved, *Lecture Notes of Computer Sciences*, **2943**, (2004), 48–53.
10. Minsky M.L. , Size and structure of universal Turing machines using Tag systems, *Proc. Sympos. Pure Math.*, **5**, (1962), 229–238.
11. Păun, Gh., Rozenberg, G., Salomaa, A., *DNA Computing*, Springer-Verlag, Berlin Heidelberg, 1998.
12. Pixton, D., Regularity of splicing languages, *Discrete Appl. Math.* **69**, (1996), 101–124.
13. Rogozhin Yu., Seven universal Turing machines, *Mat. Issled.*, No. 69, (1982), 76–90.
14. Rogozhin Yu., Small universal Turing machines, *Theoret. Comput. Sci.*, **168**, (1996), 215–240.
15. Verlan S., 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), 221–232.