Generating all recursively enumerable languages with a time-varying distributed H-systems of degree 2

Maurice Margenstern, Yurii Rogozhin

Publications du GIFM, N° 99-102

Service de reprographie de l'I.U.T. de Metz

ISBN 2-9511539-5-3 **IUT**

Groupe d'Informatique Fondamentale de Metz I.U.T. de Metz
directeur de la publication : Maurice Margenstern

Generating all recursively enumerable languages with a time-varying distributed H-systems of degree 2

Maurice Margenstern [1] Yurii Rogozhin [2]

[1]Institut Universitaire de Technologie, Metz, France
Groupe d'Informatique Fondamentale de Metz
e-mail: margens@antares.iut.univ-metz.fr
[2]Institute of Mathematics and Informatics of the
Academy of Sciences of Moldova
e-mail: rogozhin@math.md

Abstract

A time-varying distributed H system is a splicing system which has the following feature: at different moments one uses different sets of splicing rules. The number of these sets is called the degree of the system. The passing from a set of rules to another one is specified in a cycle. It is known that any formal language can be generated by a time-varying distributed H-system of degree at least 4. We proved that there are universal time-varying distributed H-systems of degree 2. In this paper we strengthen that result by showing, for any recursively enumerable language, how to construct a time-varying distributed H-system of degree 2 that exactly generates that language. We also prove that such a construction is impossible for time-varying distributed H-systems of degree 1 by proving that those systems generate regular languages.

Keywords: DNA-computing, splicing systems, Turing machines, recursively enumerable sets.

1 Introduction

Starting from [15], paper on splicing computations, a lot of studies were devoted to various extensions of H-systems originating from [3], in particular, pointing at their possible universality power. Extending the original definition, paper [15] defined the notion of *test tube* and proved the universality of an extended H-system with a finite number of test tubes. But no indication on the number of test tubes needed in order to obtain universality was given in [15]. That number was first set to 9 [1], then to 6 [16], and finally established to 3 in [21]. The latter result is very near to the real frontier between decidability and undecidability or, in other terms, between the possibility and the impossibility of predicting the eventual behaviour of such systems, depending on the number of test tubes. Indeed, as it is known that for a single test tube, generated languages are regular [19], it remains to examine the case of two test tubes, still open up to now.

Time-varying distributed H-systems were recently introduced in [16], [17, 18] as another theoretical model of biomolecular computing (DNA-computing), based on splicing operations. Instead of considering *test tubes*, these models introduce *components*, later see the formal definition, which cannot all be used at the same time but one after another, periodically.

This aims at giving an account of real biochemical reactions where the work of enzymes essentially depends on the environment conditions. In particular, at any moment, only a subset of all available rules are in action. If the environment is periodically changed, then the active enzymes change also periodically.

In [17], it is proved that 7 different *components*, are enough in order to generate any recursively enumerable language. Recently, see [11, 12], both authors proved that two components are enough for constructing a *universal* time-varying distributed H-system, *i.e.* a time-varying distributed H-system, capable of simulating the computation of any Turing machine. Universality of computation and generating any recursively enumerable language are equivalent properties, but it is *a priori* not necessarily true, that the universality of some time-varying distributed H-systems with two components entails that time-varying distributed H-systems generate all recursively enumerable languages, with only two components. In [14], it was proved that time-varying distributed H-systems with 4 components generate all recursively enumerable languages. Independently of that result, here we prove that 2 different

components are enough in order to generate any recursively enumerable language. It will be enough to show an algorithm which allows to transform any recursively enumerable language $\mathcal{L}$ into a time-varying distributed H-system that generates $\mathcal{L}$. Taking for $\mathcal{L}$ a universal recursively enumerable language, the algorithm transforms $\mathcal{L}$ into a time-varying distributed H-system that generates any recursively enumerable language. The result was presented by both authors at FBDU'98, satellite workshop of MFCS/CSL'98 Federal Conference, August 22 - 30, 1998, Brno, Czech Republic.

We also prove here that time-varying distributed H-systems of degree one generate only regular languages. Accordingly, our paper proves that the frontier between decidability and undecidability for time-varying distributed H-systems depending on the number of their components is established: two is the smallest required number of components in order to generate all recursively enumerable sets.

In our second section, we remind definitions and fix some notations. In the third section, we prove that time-varying distributed H-systems of degree one generate only regular languages. In the fourth section, we prove our claim by following a rather classical line. Instead of recursively enumerable language, we may consider recursively enumerable sets of natural numbers. One way to see that property is to consider that a recursively enumerable set of natural numbers is an encoding of some recursively enumerable language. It is well known that a recursively enumerable set of integers can be enumerated by a total recursive function, possibly computing infinitely many times the same values. The computation of that function can itself be simulated by a Turing machine. Accordingly, if we are able to obtain as a product of some time-varying distributed H-system of degree two all the final configurations computed by that Turing machine starting from initial ones, our claim will be proved. This is what we do in the fourth section by directly simulating a Turing machine. In a previous version of our work, we proved the result by simulating a tag-system, using mainly the technique we used in [12]. Hendrick Jan Hoogeboom kindly draw our attention on the fact that, probably, our construction would be directly suitable for Turing machines, without big changes. It happened to be the case and we take the occasion for thanking him here for that.

2 Definitions, notations and results

2.1 Informal indications

We recall some notions. An *alphabet* is a finite, non-empty set whose elements are called *letters*. A *word* (over some alphabet Σ) is a finite (possibly empty) concatenation of letters (from Σ). The set of all words over Σ is denoted by Σ^*. A *language* (over Σ) is a set of words (over Σ).

A language $\mathcal{L}$ is called *recursively enumerable*, or simply *r.e.*, if there is some Turing machine generating $\mathcal{L}$. This means that there is a Turing machine M which computes values $M(0)$, $M(1)$, ..., $M(n)$, ... in such a way that $\mathcal{L} = \{M(0),\ M(1),\ \ldots\ , M(n), \ldots\}$.

An *(abstract) molecule* is simply a word over some alphabet. An *abstract enzyme*, or *splicing rule*, is a quadruple (u_1, u_2, u_1', u_2') of words, which is often written in a two dimensional way as follows:

$$\begin{array}{c|c} u_1 & u_2 \\ \hline u_1' & u_2' \end{array}.$$

A splicing rule $r = (u_1, u_2, u_1', u_2')$ is applicable to two molecules m_1, m_2 if there are words w_1, w_2, w_1', w_2' with $m_1 = w_1 u_1 u_2 w_2$ and $m_2 = w_1' u_1' u_2' w_2'$, and produces two new molecules $m_1' = w_1 u_1 u_2' w_2'$ and $m_2' = w_1' u_1' u_2 w_2$. In this case, we also write $(m_1, m_2) \vdash_r (m_1', m_2')$. We also write $w \vdash_r w'$ if there are w_1 and w_1' such that $(w, w_1) \vdash_r (w', w_1')$.

A *Head-splicing-system* [3, 4], or *H-system*, is a triple $H = (V, A, R)$ of an alphabet V, a set $A \subseteq V^*$ of initial molecules over V, and a set $R \subseteq V^* \times V^* \times V^* \times V^*$ of splicing rules. H is called finite if A and R are finite sets. For any set $\mathcal{L} \subseteq V^*$ of molecules we denote by $\sigma_H(\mathcal{L}) := \{w \in V^*;\ \exists w_1, w_2 \in \mathcal{L} : \exists w' \in V^* : \exists r \in R : (w_1, w_2) \vdash_r (w, w')\}$ the set of all molecules derivable from $\mathcal{L}$ by one application of a splicing rule. Further, $\sigma_H^0(\mathcal{L}) := \mathcal{L}$, $\sigma_H^{i+1}(\mathcal{L}) := \sigma_H^i(\mathcal{L}) \cup \sigma_H(\sigma_H^i(\mathcal{L}))$, $\sigma_H^*(\mathcal{L}) := \bigcup_{i \geq 0} \sigma_H^i(\mathcal{L})$, $\sigma(H) := \sigma_H^*(A)$.

Thus, $\sigma(H)$ is the set of all molecules that can be generated in H starting with V as initial molecules by iteratively applying splicing rules to copies of the molecules already generated.

A *time-varying distributed H-system* (of degree n, $n \geq 1$), short $TVDH$-system, is a construct

$$\Gamma = (V, T, A, R_1, R_2, \ldots, R_n),$$

where V is an alphabet, $T \subseteq V$ is called *terminal alphabet*, A is a finite subset of V^* called *set of axioms*, and R_i are finite sets of splicing rules

over V, $1 \leq i \leq n$. Sets R_i, $1 \leq i \leq n$, are called the *components* of the system.

At each moment $k = n \cdot j + i$, for $j \geq 0$, $1 \leq i \leq n$, only component R_i is used for splicing the currently available strings. Specifically, we define

$$\mathcal{L}_1 = A,$$
$$\mathcal{L}_{k+1} = \sigma_i(\mathcal{L}_k), \text{ for } i \equiv k(mod\ n),\ k \geq 1,$$

where $\sigma_i = \sigma_{(V,A,R_i)}$, $1 \leq i \leq n$.

Therefore, from a step k to the next step, $k+1$, one passes only the result of splicing the strings in $\mathcal{L}_k$ according to the rules in R_i for $i \equiv k(mod\ n)$; the strings in $\mathcal{L}_k$ which cannot enter a splicing are removed. That latter clause plays a great rôle in the sequel.

The language generated by Γ is defined by

$$\mathcal{L}(\Gamma) = (\bigcup_{k \geq 1} \mathcal{L}_k) \cap T^*.$$

We denote by RE the family of recursively enumerable languages, by VDH_n, $n \geq 1$, the family of languages generated by time-varying distributed H systems of degree at most n, and VDH_* the family of all languages of this type.

Păun's latest result, [17], shows that $VDH_n = VDH_* = RE$, $n \geq 7$. A.Păun's latest result, [14], shows that $VDH_n = VDH_* = RE$, $n \geq 4$.

In this paper we improve the above result by showing that $VDH_n = RE$ for $n \geq 2$.

2.2 Our results

Our goal is now to prove the following result :

Theorem 1. *Starting from $n \geq 2$, time-varying distributed H systems generate all recursively enumerable sets, or in other words, $VDH_n = RE$ if and only if $n \geq 2$. Moreover, $VDH_1 = REG$, i.e. time varying distributed H-systems with one component generate exactly all regular languages.* This result can be split into two partial theorems:

Theorem 1a. *Time varying distributed H-systems of degree 1 generate exactly regular languages, i.e. $VDH_1 = REG$.*

Theorem 1b. *Time-varying distributed H systems of degree 2 generate all recursively enumerable sets, or in other words, $VDH_2 = RE$.*

Later, we shall refer to those results as, respectively, the decidability result and the undecidability result.

3 The decidability result

Let be $\Gamma = (V, T, A, R_1)$. As there is a single component, we can denote it simply R_1 and, accordingly, σ_1 can also be denoted by σ.

According to the definitions, $\mathcal{L}_1 = A$. We have that $\mathcal{L}_2 = \sigma(A)$.

Let us now split $\mathcal{L}_2$ into two subsets: $\mathcal{L}_2'$ will denote the words which enter the rules of the component and $\mathcal{L}_2''$, will denote those which do not enter the rules. Accordingly, $\mathcal{L}_3 = \sigma(\mathcal{L}_2')$. By construction, we have that $\mathcal{L}_2' \subseteq \mathcal{L}_2$. Split $\mathcal{L}_3$ into two subsets $\mathcal{L}_3'$ and $\mathcal{L}_3''$ defined in the same way as for $\mathcal{L}_2$. We have that $\mathcal{L}_3 = \sigma(\mathcal{L}_2')$ and, as $\mathcal{L}_2' \subseteq \mathcal{L}_2$, that $\mathcal{L}_3' \subseteq \mathcal{L}_2'$. Now, $\mathcal{L}_4 = \sigma(\mathcal{L}_3') \subseteq \sigma(\sigma(\mathcal{L}_2'))$. But, by the definition of σ, it is plain that $\sigma(\sigma(\mathcal{L})) = \mathcal{L}$ for any language $\mathcal{L}$. Accordingly, $\mathcal{L}_4 \subseteq \mathcal{L}_3$. And so, due to the definition of $\mathcal{L}(\Gamma)$, $\mathcal{L}(\Gamma) = (\mathcal{L}_1 \cup \mathcal{L}_2 \cup \mathcal{L}_3) \cap T^*$.

By [19], we know that extended H-systems with a finite set of axioms and a finite set of rules generate regular languages. Accordingly, $\mathcal{L}_2$ is a regular language. For each rule r of R, let us define L_r as the set of words upon V^* which do not enter r. As it is plain that the complement of L_r in V^* is recognizable by a finite automaton, L_r is a regular language and so, as R is finite and as $\mathcal{L}_2'' = \bigcup_{r \in R} (\mathcal{L}_2 \cap L_r)$, $\mathcal{L}_2''$ is also a regular language. As $\mathcal{L}_2'$ is the complement of $\mathcal{L}_2''$ in $\mathcal{L}_2$ which is also a regular language, we obtain that $\mathcal{L}_2'$ is a regular language. By [15], extended H-systems with a regular set of axioms and with a finite number of rules generate regular languages. Accordingly, $\mathcal{L}_3$ is a regular language, and so is $\mathcal{L}(\Gamma)$. This proves that $\mathcal{L}(\Gamma) \subseteq REG$. As it is proved in [17] that $REG \subset VDH_1$, the proof of theorem $1a$ is completed.

4 The undecidability result

It is known that $VDH_n \subseteq RE$, see [17] and [14]. For evident reasons of space, those authors could not give a precise hint of the proof. A detailed proof of the simulation of a time-varying distributed H-system is too long and too tedious, even for a technical report. However, quoting Church thesis is too short, for a technical report has room enough for a sketchy plan of the detailed proof. For the reader willing to skip this part of the report, we display this plan of a detailed proof in small characters.

As classical Turing machines, *i.e.* deterministic ones with a single head and a single bi-infinite tape, may simulate any other kind of Turing machines, see for instance [5, 25], we shall perform the simulation on a multi-tape Turing machine.

Let $\Gamma = (V, T, A, R_1, R_2, \ldots, R_k)$ be a time-varying distributed system of degree k. The simulating Turing machine, say M, has k+9 tapes that are infinite only to right. These tapes are initialized starting from their first cell. Some of them has two heads and they have the following rôles: the first tape, say $\mathcal{I}$, contains input m, a natural number, which indicates that M stops when it has produced the m+1$^{\text{th}}$ element of $\mathcal{L}(\Gamma)$ in the enumeration of that language according to the program of M. For counting reasons, that tape has a second head. Then k tapes contain the encoding in a fixed form of sets R_i, each one encoded on a tape devoted to it. Four tapes are auxiliary tapes devoted to the computation. In the first three ones, say $\mathcal{W}_1$, $\mathcal{W}_2$ and $\mathcal{W}_3$, the machine is supposed to compute all words of length ℓ+1 in a preliminary stage of its computation, where $\ell = m$ at that moment. Another tape, $\mathcal{S}$ is devoted to simulate the *soup* produced by the simulation. Tape $\mathcal{S}$ is empty at the beginning of the computation. Then three tapes come in order to implement several counters: for the length ℓ of the word of V^* on which the computation is performed, the number of iterations of the current rules, the highest index of $\mathcal{L}_i$ for which an element is computed. Call these counters, respectively, $\mathcal{K}_\ell$,$\mathcal{K}_{it}$ and $\mathcal{K}_{ind}$. Their common initial value is m+1 and all of them have two heads in order to make the simulation more effective. A last tape, say $\mathcal{R}$, is devoted to gather the results, that is the word of $\mathcal{L}(\Gamma)$ that are yielded during the simulation. That latter tape is empty at the beginning of the simulation.

The program of the machine works as follows:

In a first stage, the machine computes all the words of length m+1 in all $\mathcal{W}_i$'s at the same time: the corresponding heads of the tapes work in parallel. In order to accelerate the computation, we may assume that each tape $\mathcal{W}_i$ has two heads. Of course, the program makes use of $\mathcal{K}_\ell$, whose content is compared with the content of $\mathcal{I}$, the input tape.

Still in this first stage, the machine computes all possible iterated applications of rules of R_1, at most m+1 times, storing the results in $\mathcal{S}$ if they contain a symbol in $V \backslash T$, in $\mathcal{R}$ if the produced word is in T^*. The machine simulates first one application of the rules, then a second application and so on, updating $\mathcal{K}_{it}$ each time it is needed, until m+1 iterations are performed or m+1 words are already stored on $\mathcal{R}$. The second head of $\mathcal{I}$ advances by one step to right when a new word is stored on $\mathcal{R}$. If m+1 words are stored on $\mathcal{R}$, the machine performs the terminal stage.

If this is not the case, assuming it has simulated m+1 successive applications of rules in R_i, now it simulates m+1 successive applications of rules in R_j with $j \equiv i$+1 mod k, making use of tapes $\mathcal{W}_i$'s and $\mathcal{S}$. Again the results are stored in $\mathcal{R}$ or $\mathcal{S}$ depending on whether they are or not in T^*. If $j = 1$, then $\mathcal{K}_{ind}$ is incremented by one. If m+1 words are present on $\mathcal{R}$, the machine performs the terminal stage. If not, it goes on until all applications up to m+1 iterations are performed, then it performs a new turn of this stage, processing with R_h with $h \equiv j$+1 mod k.

If all previous computations are performed up to m+1 and if there are not m+1 words stored on $\mathcal{R}$, then the maximum of length for words and for iterations are incremented by one. Assume that the current length was ℓ, then it is ℓ+1 and, in preliminary stage, the machine adds to tapes $\mathcal{W}_i$'s all the words in V^* of length ℓ+1. Then it repeats the stages we already indicated for computing all applications of the rules, iterated up to ℓ+1 times, successively taking into account ℓ+1 components, possibly starting periodically from R_1.

The terminal stage is easy: the machine erases the first m words stored on $\mathcal{R}$ and pushes back the last word produced at the beginning of the tape. Then this is performed, it stops.

Notice that machine M may not halt: this is the case if $\mathcal{L}(\Gamma)$ is finite. This property is coherent with the fact that the finiteness or not of $\mathcal{L}(\Gamma)$ cannot be

known by applying some algorithm, fixed in advance.

Consequently, we obtain that $VDH_2 \subseteq RE$. And so, in order to prove theorem $1b$, we have to prove that for any recursively enumerable set $\mathcal{R}$, we are able to devise a time-varying distributed H-system with exactly two components, which generates $\mathcal{R}$. We shall start by informal indications which explain the set of rules that we later introduce.

4.1 Informal indications

In our proof, we shall directly simulate the computation of a Turing machine.

Let us remind the working of Turing machines. Such a machine makes use of a tape, infinite in both directions, splitted into cells each one containing a letter of some fixed alphabet. There is a distinguished letter called the *blank* and it is assumed that at the starting point of the computation, all cells of the tape, except possibly finitely many ones, contain the blank symbol. The head of the machine is able to read each cell, one after another, and when it has read the *current symbol*, it replaces it by a new one, possibly the same. In that case we say that the new symbol has been written by the machine on the tape. Afterwards, the machine goes to scan the content of another cell, the right neighbour or the left neighbour of the cell it scanned, or it may scan again the same cell. When the machine head scans a cell, it is assumed that it reads the scanned symbol under a certain *state* belonging to a fixed finite set. When it has written the new symbol, possibly the same as the one it read, the machine head enters a new state, possibly the same.

At this point, let us sum up the described transformation as follows: at time t, scanning symbol x of cell c under a state $\mathbf{a}$, the machine writes y in place of x, enters state $\mathbf{b}$ and scans the content of cell $c+1$, or cell $c-1$, or cell c again, in order to be ready for the the action to be performed at time $t+1$. We say, respectively, that the machine head goes to right, to left or remains on the same place. And so, the considered action can be encoded by the following quintuple: $\boxed{\mathbf{a}\ x\ M\ y\ \mathbf{b}}$, where M called the *move* of the head takes values R, L or S according to whether the machine head goes to right, to left or remains on the same cell. Such a quintuple is called an *instruction* and the program of a Turing machine is a finite sequence of instructions.

At time 0, by convention the starting time of the computation, the *current* configuration is the smallest part of the tape containing the cell

scanned by the machine head and outside which all cells are blank, together with the indication of the cell scanned by the head and of the state of the head. Later, the new *current* configuration is defined with respect to the previous current configuration by the previous transformation. We can illustrate this change by the following figure:

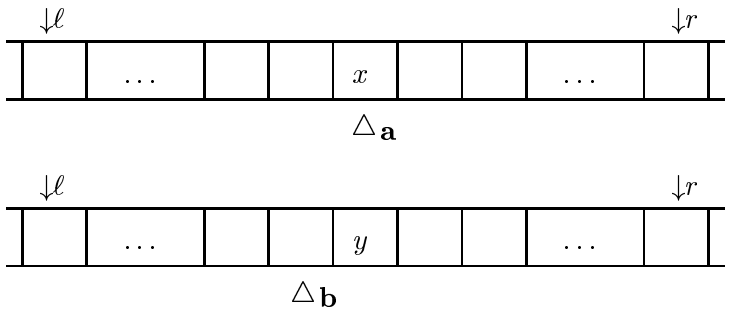


Figure 1. One step in a Turing machine computation

where the head of the machine goes to the left of the previous scanned cell.

On the above figure, the upper part represents the current configuration at time t and the lower part does the same for time $t+1$. The *ends* of the configuration are indicated by $\downarrow \ell$ and $\downarrow r$, respectively left end and right end. On that figure, both configurations are of the same length, as far as the transformation concerned an interior cell. Assume now that at time t, the head scans the left end of the current configuration being in state $\mathbf{a}$ and reading x. Then the position of the left end of the configuration at time $t+1$ will be by one cell to the left of the position of the same end at time t. A symmetrical conclusion holds for the right end if the head moves to the right while scanning the right end cell.

The part of the tape delimited by the both ends of the configuration can be represented by a word on the alphabet of the symbols which can be read or written on the tape, say the *tape alphabet*. A slight modification of that word allows to represent the configuration: let ω_ℓ be the part of that word representing the contents of the cells ranging from the left end up to the left neighbour of the scanned cell. It is clear that ω_ℓ is empty if the scanned cell is the left end of the current configuration. Symmetrically define ω_r to be the part of the word representing the set of cells ranging from the right neighbour of the scanned up to the right end of the configuration. It is clear that ω_r is empty if the scanned

cell is in the right end cell. If x denotes the scanned symbol and if the machine head is in state $\mathbf{a}$, we can denote the current configuration by $\omega_\ell\,\mathbf{a}\,x\,\omega_r$.It will be enough that the states are denoted by letters which do not belong to the tape alphabet.

Assume that ω_ℓ is not empty. It can be rewritten as $\nu_\ell\,z$, where z is a tape symbol. Then the action we described in the above figure can be rewritten in the following way:

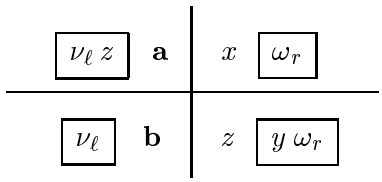


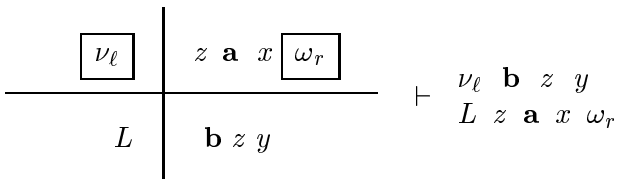
Figure 2. From the current configuration to the next one

This looks rather near to a splicing rule.

Indeed, what we shall do boils down to make the just written scheme enter the frame of splicing rules.

Let us show a bit informally for an instruction with a move to left and another instruction with a move to right.

Let us first consider instruction $\boxed{\mathbf{a}\,x\,L\,y\,\mathbf{b}}$. Using the previous figure, we see that we cannot directly go from the upper line to the lower one by a splicing rule because a first effect of the splicing rule is to split the current *molecule* in two parts. And so, we need a second rule for recombining the right parts after a second splitting. It can be done by just using an intermediary configuration:



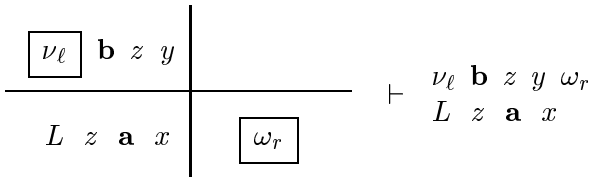


Figure 3. Simulating a move to left

taking into account that $\omega_\ell = \nu_\ell\, z$ is assumed.

Consider now an instruction to right, say $\boxed{\mathbf{a}\; x\; R\; y\; \mathbf{b}}$. We get the following figures:

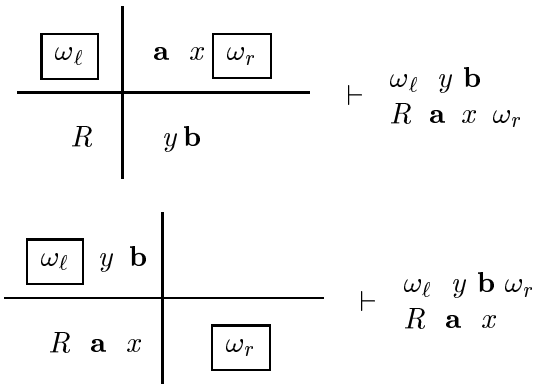


Figure 4. Simulating a move to right

Notice the following particularity of those figures. In both cases, the second rule applies to both words produced by the application of the first rule. This is not the case of the first rules where, in both cases, the lower word of the figure is not the representation of a configuration: $L\,\mathbf{b}\,z\,y$ in the first case, $R\,y\,\mathbf{b}$ in the second one. Those auxiliary words are necessary for applying the rules. They must be brought in some way. This is the reason of the axioms which will be introduced in the formal definitions and of special rules the rôle of which will be to keep needed axioms ready for use.Special rules are also needed for producing auxiliary words without which the computation could not go on.

These figures hold for transformations occurring *inside* the current configurations, not for their ends.

That remark raises the question of the delimitation of the Turing configuration. The simplest way is to introduce new letters with that rôle. And so, let us consider that $\omega_\ell\; \mathbf{a}\; x\; \omega_r$ is encoded by $X\; \omega_\ell\; \mathbf{a}\; x\; \omega_r\; Y$. In some regards, that encoding corresponds to what nature does in DNA

strands, where informating parts are enbraketed within special delimiters, always the same ones, see, for instance, [7].

After these introductory informal remarks, we can now proceed to the formal definition of our representation of a Turing machine computation and to the formal description of our time-varying distributed H-system with two components.

4.2 Formal definitions

As already indicated in our introduction, we shall consider recursively enumerable sets of natural numbers instead of recursively enumerable languages. It will make our proof simpler.

It is a well known fact that from theoretical computer science that for every recursively enumerable set $\mathcal{L}$ of natural numbers, there is a Turing machine $T_{\mathcal{L}}$ which generates $\mathcal{L}$ as follows. Let $f_{\mathcal{L}}$ be a total recursive function enumerating the elements of $\mathcal{L}$, *i.e.* $\mathcal{L} = \{f_{\mathcal{L}}(x)\ ;\ x \in I\!N\}$. It is well known, see [6], chap.XIII, § 68, Theorem XXVIII, how to construct a Turing machine $T_{\mathcal{L}}$ from the definition of $f_{\mathcal{L}}$ which would compute what f does, *i.e.* for every natural number x, machine $T_{\mathcal{L}}$ transforms any initial configuration $q_1 01^x$ into the final configuration $q_0 01^{f_{\mathcal{L}}(x)} 0 \ldots 0$.

> **Remark** The number of 0's occuring after the position of the head in the previous final configuration, can easily be computed: if $s(x)$ is the space needed by the Turing machine during its computation, that number is $\max\{0, s(x) - f_{\mathcal{L}}(x) - 1\}$.

Let us assume for a while that we succeeded to simulate machine $T_{\mathcal{L}}$. It is not yet enough for proving the theorem. Indeed, starting from one word, say $q_1 01^x$ for some x, we would obtain an encoding of $f_{\mathcal{L}}(x)$ as $q_0 01^{f_{\mathcal{L}}(x)}$. But this is not what we need, because this gives us a single word.

It is not difficult to see that starting from machine $T_{\mathcal{L}}$, there is a Turing machine $T'_{\mathcal{L}}$ which computes $01^{x+1} q_0 01^{f_{\mathcal{L}}(x)} 0 \ldots 0$, starting from $q_1 01^x$ As we shall see, it will be possible for us to device a time-varying distributed H-system Γ which, arriving to that stage of the computation, will split the obtained word in two parts: $1^{f_{\mathcal{L}}(x)}$ if $f_{\mathcal{L}}(x) \neq 0$ and 0 if $f_{\mathcal{L}}(x) = 0$ as a *generated* word and $q_1 01^{x+1}$ as a new starting configuration. This will allow us to obtain exactly $\mathcal{L}$ as $\mathcal{L}(\Gamma)$.

In what follows, in order to facilitate the reading of the formulas, we shall write $f(x)$ instead of $f_{\mathcal{L}}(x)$ when it will be clear from the context that that latter function is meant.

Definition of Γ

And so, we assume that $\{0,1\}$ is the tape alphabet of Turing machine $T'_{\mathcal{L}}$ and let us number its states by $q_0, q_1, \ldots, q_n$, where q_1 is the initial state and q_0 the final one. From now on, we shall use latin small letters for denoting tape symbols, greek letters for words on the alphabet of the tape symbols, bold latin small letters for states of the Turing machine.

Define $TVDH$-system Γ as follows:

Its terminal alphabet is $T = \{0,1\}$, the tape alphabet of $T'_{\mathcal{L}}$. Let us introduce variables for $q_0, q_1, \ldots, q_n$, namely $\mathbf{a}$ and $\mathbf{b}$ with that additional assumption that $\mathbf{b}$ ranges over all q_i's but $\mathbf{a}$ ranges over all of them except q_0. Moreover, we shall introduce two copies of the set of q_i's, namely $\{q'_0, q'_1, \ldots, q'_n\}$ and $\{q''_0, q''_1, \ldots, q''_n\}$. Accordingly, if $\mathbf{a}$ denotes one of the q_i's, say q_j, then $\mathbf{a}'$ denotes q'_j and $\mathbf{a}''$ denotes q''_j. We shall use the same convention with $\mathbf{b}$.

Let also $x, y, z \in \{0,1\}$.

Define alphabet V by setting

$V = \{0,1\} \cup \{X, Y, Y', \mathbf{b}, \mathbf{b}', \mathbf{b}'', t_1, t_2, t_3, t_4, t_5, R, L, S, Q, Z, \rightarrow, \leftarrow\}$, with $\mathbf{b}, \mathbf{b}', \mathbf{b}''$ taking all their possible values.

Axioms A are given by

$$A = \{Xq_1Y,\ t_1\mathbf{a}'0Y,\ t_2\mathbf{a}0Y,\ Z\mathbf{b},\ RyZ,\ S\mathbf{b}Z,\ L\mathbf{b}xZ,\ X\mathbf{b}''0yt_3,$$
$$X\mathbf{b}t_4,\ Z\rightarrow,\ RQZ,\ t_31Z,\ t_5\leftarrow,\ Zx,\ t_4\leftarrow Z,\ Y',\ \ \leftarrow Y,$$
$$Xq_1Z,\ Xq_10Z,\ Q\leftarrow Z\},$$

with $\mathbf{a}, \mathbf{a}', \mathbf{b}, \mathbf{b}'', \mathbf{x}, \mathbf{y}$ taking all their possible values.

Notice that for any pair $\mathbf{a}$, x, there is at most one instruction in the program of $T'_{\mathcal{L}}$ of the form $\boxed{\mathbf{a}\,x\,M\,y\,\mathbf{b}}$, M, y, $\mathbf{b}$ being uniquely determined. Denoting by $\mathcal{I}$ the set of instructions of $T'_{\mathcal{L}}$, the expressions $\mathbf{a}$, $x \in \mathcal{I}$ as well as $\mathbf{a}$, x, $\mathbf{b}$, $y \in \mathcal{I}$ will be used for indicating an instruction by its constituting elements. In particular, we will use quantifiers with such expressions. Moreover, we shall specialise $\mathcal{I}$ by denoting by $\mathcal{I}_R$ the set of all instructions with a move to right, by $\mathcal{I}_L$ the set of all instructions with a move to left and by $\mathcal{I}_S$, the set of all stationary instructions, *i.e.* with $M = S$.

The components of rules, are given below by:

component R_1

1.1 : $\dfrac{s \mid \mathbf{a}Y}{t_1 \mid \mathbf{a}'0Y}$
$\forall s \in \{0,1,X\},\ \forall \mathbf{a},$

1.2 : $\dfrac{s \mid \mathbf{a}\,x}{R \mid y\,\mathbf{b}}$
$\forall\, s \in \{0,1,X\},$
$\forall\, \mathbf{a}, x, y, \mathbf{b} \in \mathcal{I}_R,$

1.3 : $\dfrac{s \mid \mathbf{a}\,x}{S \mid \mathbf{b}\,y}$
$\forall\, s \in \{0,1,X\},$
$\forall\, \mathbf{a}, x, y, \mathbf{b} \in \mathcal{I}_S,$

1.4 : $\dfrac{s \mid z\,\mathbf{a}\,x}{L \mid \mathbf{b}\,z\,y}$
$\forall\, s \in \{0,1,X\},\ \forall\, z,$
$\forall\, \mathbf{a}, x, y, \mathbf{b} \in \mathcal{I}_L,$

1.5 : $\dfrac{X\,\mathbf{a}\,x \mid s}{X\,\mathbf{b}''\,0\,y \mid t_3}$
$\forall\, s \in \{0,1,Y\},$
$\forall\, \mathbf{a}, x, y, \mathbf{b} \in \mathcal{I}_L,$

1.6 : $\dfrac{1 \mid q_0\,0\,1}{R \mid Q \rightarrow}$

1.7 : $\dfrac{1 \mid q_0\,0\,0}{\mid Q \leftarrow 0}$

1.8 : $\dfrac{s \mid \rightarrow 1}{t_3 \mid 1 \rightarrow}$
$\forall s \in \{1, Q\},$

1.9 : $\dfrac{\mid \rightarrow 0}{t_5 \mid \leftarrow}$

1.10 : $\dfrac{\leftarrow 1 \mid}{t_4 1 \leftarrow \mid}$

1.11 : $\dfrac{Z \mid 1}{t_4 \leftarrow \mid Z}$

component R_2

2.1 : $\dfrac{s \mid \mathbf{a}'\,0Y}{t_2 \mid \mathbf{a}\,0Y}$
$\forall s \in \{0,1,X\},\ \forall\, \mathbf{a},$

2.2 : $\dfrac{y\,\mathbf{b} \mid}{R\,\mathbf{a}\,x \mid}$
$\forall\, \mathbf{a}, x, y, \mathbf{b} \in \mathcal{I}_R,$

2.3 : $\dfrac{Z \mid \mathbf{b}}{R\,y \mid Z}$
$\forall\, \mathbf{b}, y,$

2.4 : $\dfrac{s\,\mathbf{b}\,y \mid}{S\,\mathbf{a}\,x \mid}$
$\forall\, s \in \{0,1,X\},$
$\forall\, \mathbf{a}, x, y, \mathbf{b} \in \mathcal{I}_S,$

2.5 : $\dfrac{Z \mid y}{S\,\mathbf{b} \mid Z}$
$\forall\, \mathbf{b}, y$

2.6 : $\dfrac{s\,\mathbf{b}\,z\,y \mid}{L\,z\,\mathbf{a}\,x \mid}$
$\forall\, s \in \{0,1,X\},\ \forall\, z,$
$\forall\, \mathbf{a}, x, y, \mathbf{b} \in \mathcal{I}_S,$

2.7 : $\dfrac{Z \mid y}{L\,\mathbf{b}\,z \mid Z}$
$\forall \mathbf{b},\ \forall y, z,$

2.8 : $\dfrac{X\,\mathbf{b}'' \mid 0}{X\,\mathbf{b} \mid t_4}$
$\forall \mathbf{b},$

2.9 : $\dfrac{Q \rightarrow \mid}{R\,q_0\,0 \mid}$

2.10 : $\dfrac{Z \mid \rightarrow}{R\,Q \mid Z}$

2.11 : $\dfrac{Z \mid 0}{Q \leftarrow \mid Z}$

2.12 : $\dfrac{1 \rightarrow \mid}{t_3 \rightarrow 1 \mid}$

$$1.12: \quad \begin{array}{c|l} & Q \leftarrow Y' \\ \hline & \leftarrow Y \end{array} \qquad\qquad 2.13: \quad \begin{array}{c|l} Z & \rightarrow \\ \hline t_3\, 1 & Z \end{array}$$

$$2.14: \quad \begin{array}{c|l} s & 1 \leftarrow \\ \hline t_4 & \leftarrow 1 \end{array}$$

$\forall s \in \{0, 1, Q\}$,

$$2.15: \quad \begin{array}{c|l} Q \leftarrow & x \\ \hline & Y' \end{array}$$

$\forall x$,

$$2.16: \quad \begin{array}{c|l} X\, 0 \leftarrow & 1 \\ \hline X\, q_1\, 0 & Z \end{array}$$

Both components R_1 and R_2 also contain the following rules:

$$\begin{array}{c|} \alpha & \\ \hline \alpha & \end{array}$$

for each axiom $\alpha \in A$, except Xq_1Y. As we shall see in the computation, axioms must always be present for making it possible to apply most of the rules.

4.3 Checking the simulation

4.3.1 Checking the simulation of one step in a Turing computation

The first thing we have now to check is that starting from a word representing the current configuration of machine $T'_{\mathcal{L}}$, the application of the rules of Γ leads to a word representing the next configuration. In other words, given any instruction of the program of $T'_{\mathcal{L}}$, we have to show that there are rules in R_1 and R_2 and axioms allowing to mimic the application of that instruction to a suitable configuration.

There are several cases according to the move of the instruction and of the position of the scanned cell in the configuration: according to whether the move is to right, to left or stationary; according to whether the scanned cell is an interior one or a cell being at an end of the configuration.

Case of an interior cell

We already seen, in the informal part how to simulate an instruction with a right move or a left move to an interior cell. Let us again have a look at it.

Let ω_ℓ, ω_r be finite words of alphabet $\{0,1\}$ and $\omega_\ell\,\mathbf{a}\,x\,\omega_r$ be the current configuration of machine $T'_{\mathcal{L}}$ with $\omega_\ell = \nu_\ell\,z$. Assume that Γ has obtained word $X\,\nu_\ell\,z\,\mathbf{a}\,x\,\omega_r\,Y$ at a time, say t, when the rule to be applied must be taken from R_1. Assume that instruction $\boxed{\mathbf{a}\,x\,L\,y\,\mathbf{b}}$ applies to the current configuration. Possibly after several steps, Γ should produce word $X\,\nu_\ell\,\mathbf{b}\,z\,y\,\omega_r\,Y$. Looking above at figure 3, the reader can get convinced that in two steps Γ produces the expected word and no undesirable one: indeed, rule 1.4 applies as far as word $L\,\mathbf{b}\,z\,y$ is present, (it is obtained by rule 2.7), leading to words $X\,\nu_\ell\,\mathbf{b}\,z\,y$ and $L\,z\,\mathbf{a}\,x\,w_r\,Y$ that can enter in a rule of R_2 and, therefore, are kept for the further processing. Now, rule 2.6 straightforward applies to these words and provides $X\,\nu_\ell\,\mathbf{b}\,z\,y\,\omega_r\,Y$ and $L\,z\,\mathbf{a}\,x$. Observe now that $L\,z\,\mathbf{a}\,x$ cannot enter a rule of R_1 and so, it is ruled out. On another hand, word $X\,\nu_\ell\,\mathbf{b}\,z\,y\,\omega_r\,Y$ is the representation of the next configuration and, as it is now the turn to apply a rule namely from R_1, we see that the previous conditions are again observed, which proves the simulation of one step of Turing computation by Γ, at least when a instruction with a move to left is applied.

With the help of figure 4, the reader can now check that in the case of an instruction $\mathbf{a}\,x\,R\,y\,\mathbf{b}$ with a move to right, rules 1.2 and 2.2 successively apply, providing the expected next configuration, namely $X\,\omega_\ell\,y\,\mathbf{b}\,\omega_r\,Y$. In the same way, the reader can check the simulation of applying a stationary instruction, say $\mathbf{a}\,x\,S\,y\,\mathbf{b}$, via rules 1.3, 2.4 and 2.5.

Case of end cells

We have now to check what happens when the head has to go to the right of the right end of the configuration or to the left of the left end. As in our representation of a configuration the ends are realised by special markers, X for the left end, Y for the right one, such cases are easily characterised : $X\,\mathbf{a}\,x\,\boxed{\nu}\,Y$ for the left end and $X\,\boxed{\omega}\,\mathbf{a}\,Y$ for the right one.

In the case of the left end, the case is dealt with by the already seen rules 1.2, 2.2 and 2.3 if the instruction determined by $\mathbf{a}\,x$ has a move to right, and by 1.3, 2.4 and 2.5 if it is a stationary instruction. When $\mathbf{a}\,x$ determines an instruction with a move to left, it can then be checked that first rule 1.5 applies, yielding words $X\,\mathbf{a}\,x\,t_3$ and $X\,\mathbf{b}''\,0\,y\,\boxed{\nu}\,Y$. Both words can enter a rule of R_2, namely 2.8 which now yields $X\,\mathbf{b}\,0\,y\,\boxed{\nu}\,Y$ and $X\,\mathbf{b}''\,t_4$. As that latter word cannot enter any rule of R_1, it

is removed and so, in that case we checked that Γ also simulates the behaviour of Turing machine $T'_{\mathcal{L}}$.

The situation of a right end is not dealt with symmetrically. Indeed, there is a uniform way for solving that case: it is enough to transform representation $X\,\boxed{\omega}\,\mathbf{a}\,Y$ into the following one: $X\,\boxed{\omega}\,\mathbf{a}\,0\,Y$, which can be dealt with as the case of an interior cell. It is easy to check that rule 1.1 and then rule 2.1 allow to obtain the just described transformation.

4.3.2 Checking how to resume the computation on 01^{x+1}

We now arrive at an important point of our simulation. From what we already seen, we now prove that Γ transforms $X\,q_1\,0\,1^x\,Y$ into $X\,0\,1^{x+1}\,q_0\,0\,1^{f(x)}\,0\,\ldots\,0\,Y$.

Let us write $\omega_m = \begin{cases} 1^{f(m)} & \text{if } f(m) \neq 0 \\ 0 & \text{if } f(m) = 0 \end{cases}$ where f enumerates $\mathcal{L}$, and inductively define $\mathcal{L}_m$ by

$$\mathcal{L}_0 = \{\omega_0\}$$
$$\mathcal{L}_{k+1} = \mathcal{L}_k \cup \{\omega_{k+1}\}.$$

It will be enough to check that if at some time t_k we have obtained $\mathcal{L}_k$ and $X\,q_1\,0\,1^{k+1}\,Y$, then at a later time t_{k+1} we shall obtain $\mathcal{L}_{k+1}$ and $X\,q_1\,0\,1^{k+2}\,Y$. And so, starting from $X\,0\,1^{k+2}\,q_0\,0\,1^{f(k+1)}\,0\,\ldots\,0\,Y$, we leave a marker, Q at the place of $q_0 0$ and, at the same time, send a signal searching the first 0 following the 0 scanned by the head at that time. When the wanted 0 is found, it cuts off the superfluous 0's and Y and the signal goes back to the marker. When it reaches it, it splits the word in two parts: the right one is the expected ω_{k+1}; in the left one, the marker is changed into Y and the signal goes on to left until it meets 0. At that moment it transforms what is on the left of that $X0$ into Xq_10, and the new expected starting configuration is reached. This is illustrated by Figure 5, below.

Starting from word $\boxed{X\,0\,1^{k+2}\,q_0\,0\,1^{f(k+1)}\,0\,\ldots\,0\,Y}$, the application of rules 1.6, 2.9 and 2.10 allows to put marker Q on the place of q_0, to erase $q_0 0$ and to insert in front of $1^{f(k+1)}$ signal $\rightarrow$, in the case when $f(k{+}1) \neq 0$: indeed, the application of rule 1.6 yields $X\,0\,1^{k+2}\,Q\,\rightarrow$ and $R q_0\,0\,1^{f(k+1)}\,0\,\ldots\,0\,Y$. Both words are kept as far as they enter rule 2.9 which in its turn yields word $\boxed{X\,0\,1^{k+2}\,Q\rightarrow 1^{f(k+1)}\,0\,\ldots\,0\,Y}$, as well as $R\,q_0\,0$ which enter no rule of R_1 and, consequently, is ruled out. Then, rules 1.8, 2.12 and 2.13 allow to cross over the block of 1's constituting the expected result ω_{k+1}. Indeed, assuming that the

computation starts from word $X\,0\,1^{k+2}\,Q\,1^u \rightarrow 11^v\,0\,\ldots\,0\,Y$, rule 1.8 yields $X\,0\,1^{k+2}\,Q\,1^u 1 \rightarrow$ and $t_3 \rightarrow 1\,1^v\,0\,\ldots\,0\,Y$ which enter rule 2.12 yielding $X\,0\,1^{k+2}\,Q\,1^u 1 \rightarrow\ 1^v\,0\,\ldots\,0\,Y$ and $t_3 \rightarrow 1$ which is ruled out, as far as it enters no rule of R_1. And so, by induction on the number of 1's to be crossed over, we see that the appropriate iteration of rules 1.8, 2.12 and 2.13 lead to word $\boxed{X\,0\,1^{k+2}\,Q\,1^{f(k+1)} \rightarrow\ 0\,\ldots\,0\,Y}$. But now, rule 1.9 allows to throw away the superfluous substring $0\,\ldots\,0\,Y$ and to reverse the signal, leading to word $\boxed{X\,0\,1^{k+2}\,Q\,1^{f(k+1)} \leftarrow}$. Notice that now, the starting set of rules becomes R_2. Indeed, repeated applications of rule 2.14, 1.10 and 1.11 allow to arrive to word $\boxed{X\,0\,1^{k+2}\,Q \leftarrow\ 1^{f(k+1)}}$. Indeed, starting from $X\,0\,1^{k+2}\,Q\,1^u\,1 \leftarrow\ 1^v$, rule 2.14 yields $X\,0\,1^{k+2}\,Q\,1^u \leftarrow 1$ and $t_4\,1 \leftarrow 1^v$, both word entering rule 1.10, yielding $X\,0\,1^{k+2}\,Q\,1^u \leftarrow 1\,1^v$ and $t_4 1 \leftarrow$, the latter being ruled out as far as it enters no rule of R_2.

$$\boxed{X\,0\,1^{k+2}\,q_0\,0\,1^{f(k+1)}\,0\,\ldots\,0\,Y}$$

$$\boxed{X\,0\,1^{k+2}\,Q\ \rightarrow\ 1^{f(k+1)}\,0\,\ldots\,0\,Y}$$

$$\boxed{X\,0\,1^{k+2}\,Q\,1^{f(k+1)}\ \rightarrow\ 0\,\ldots\,0\,Y}$$

$$\boxed{X\,0\,1^{k+2}\,Q\,1^{f(k+1)}\ \leftarrow}$$

$$\boxed{X\,0\,1^{k+2}\,Q\ \leftarrow\ 1^{f(k+1)}}$$

$$\boxed{X\,0\,1^{k+2}\ \leftarrow\ Y} \qquad \boxed{1^{f(k+1)}}$$

$$\boxed{X\,0\ \leftarrow\ 1^{k+2}\,Y}$$

$$\boxed{X\,q_1\,0\,1^{k+2}\,Y}$$

Figure 5. Adding a new element of $\mathcal{L}$ (right) and turning to the computation of the next one (left)

The case when $f(k{+}1) = 0$ is dealt with by rules 1.7 and 2.11. It arrives immediately at word $\boxed{X\,0\,1^{k+2}\,Q \leftarrow 0}$, also yielding $q_0\,0\,\ldots\,0Y$ which is ruled out as far as it enters no rule of R_2.

Now, whatever is the value of $f(k{+}1)$, rule 2.15 and 1.12 apply and allow to separate the result as a word in the terminal alphabet, namely ω_{k+1} from the other part, transformed into word $\boxed{X\,0\,1^{k+2} \leftarrow Y}$. Indeed, applied to word $X\,0\,1^{k+2}\,Q \leftarrow 1^{f(k+1)}$, with the convention $1^0 = 0$, rule 2.15 yields $X\,0\,1^{k+2}\,Q \leftarrow Y'$ and $\boxed{1^{f(k+1)}}$. The latter word is the expected ω_{k+1} and the computation goes on with the former word, and it enters rule 1.12 allowing to obtain $\boxed{X\,0\,1^{k+2} \leftarrow Y}$ and also $Q \leftarrow Y'$ which is ruled out, as entering no rule of R_2.

We are now reaching the last stage of the computation leading to $\boxed{X\,q_1\,0\,1^{k+2}\,Y}$. Starting from word $X\,0\,1^{k+2} \leftarrow Y$, rules 2.14, 1.10 and 1.11 again apply, allowing after an appropriate iteration to reach word $X\,0 \leftarrow 1^{k+2}\,Y$, to which rule 2.16 applies, yielding the needed configuration $\boxed{X\,q_1\,0\,1^{k+2}\,Y}$ together with $X\,0 \leftarrow Z$ which is ruled out as entering no rule of R_1.

And so we proved our claim about turning from $\mathcal{L}_k$ to $\mathcal{L}_{k+1}$.

5 Conclusion

It could be indicated that a small improvement can be brought to our proof in order to a bit minimise the number of *rules* in each component.

Indeed, let us go back to the considerations of Turing machines simulating the computation of partial recursive functions. As shown in [6], we may assume that during its computation, $T_{\mathcal{L}}$ never goes to the left of the 0 occurring in the initial configuration $q_1 01^x$ and that machine $T_{\mathcal{L}}$ has $\{0, 1\}$ as a tape alphabet, see [8] for that latter property. It is not difficult to device machine $T'_{\mathcal{L}}$ in order it also satisfies both these properties. This will allow us not to consider the case when the head of the Turing machine has to go to the left of the left end of the current configuration. This allows to get rid of rules 1.5 and 2.8.

Another possible way to explore lies in the following remark. It is not difficult to notice that our proof does not at all make use of the parallelism which is contained in the definition of time-varying distributed H-systems as well as in all kind of H-systems. In some sense it could be said that the proof needs to avoid parallelism. We can illustrate it as

follows. As the working of a deterministic Turing machine is essentially a sequential one, it is *a priori* natural that our simulation of that part of the computation is sequential and, in some extent, it seems natural to consider it as non-parallelizable in the general case. But when we have to go from word $\boxed{X\,0\,1^{k+2}\,q_0\,0\,1^{f(k+1)}\,0\,\ldots\,0\,Y}$ to both word ω_{k+1} and the starting point for the next computation, namely word $\boxed{X\,q_1\,0\,1^{k+2}\,Y}$, it could be suggested to do that in parallel according to the following scheme:

$$\frac{\boxed{X\,0\,1^{k+2}\,q_0\,0\,1^{f(k+1)}\,0\,\ldots\,0\,Y}}{\boxed{X\,0\,1^{k+2}\,\leftarrow\,Y}\qquad\qquad\boxed{\rightarrow 1^{f(k+1)}\,0\,\ldots\,0\,Y}}$$

Figure 6. A tentative scheme for parallelization

At first glance, it seems possible that $\rightarrow$ goes forward to the 0's in order to get ω_{k+1} at the same time as $\leftarrow$ goes backward to $X\,0$ in the other word. But this does not work so easily: it may turn out that ω_{k+1} is so long, that during the time needed by $\rightarrow$ to reach the first 0 occurring after the block of 1's, the system has enough time for going to $\boxed{X\,0\,1^{k+3}\,q_0\,0\,1^{f(k+2)}\,0\,\ldots\,0\,Y}$ and that the same process as indicated in figure 6 starts for obtaining ω_{k+2}. Now, with the rules we gave, both processes for obtaining ω_{k+1} and ω_{k+2} will interfere, producing the results ω_{k+1} and ω_{k+2} but also undesirable values which may not belong to $\mathcal{L}$.

But do such long ω_{k+1} occur? The answer is yes. Assume it would be possible to compute $\mathcal{L}$ by an *increasing* total recursive function. That would allow us to perform the scheme indicated by figure 6, because when the almost ω_{k+2} would be separated, ω_{k+1} would already be yielded and no interference could occur. But it is not possible to compute any recursively enumerable infinite set by an increasing total function, worse there are examples that cannot be computed that way. The reason is simple: if it were always possible to compute a recursively enumerable infinite set by an increasing total recursive function, the considered set would always be recursive. Indeed, here is a decision procedure for $n \in \mathcal{L}$: for any n, compute $f_{\mathcal{L}}(0)$, ..., $f_{\mathcal{L}}(n)$; if $n \in \mathcal{L}$, it necessarily occurs among those computed values; if it does not occur, then $n \notin \mathcal{L}$. It is not difficult to adapt that argument if we assume that $|f_{\mathcal{L}}(n) - f_{\mathcal{L}}(n+1)|$

cannot reach arbitrarily large values.

And so, it seems to us that the following questions make sense: how many components are needed in order to obtain a *parallel* universal computation? How many of them are needed in order to generate any recursively enumerable set with a parallel computation?

A partial answer is given in [14] where it is proved that four components are sufficient to generate all recursively enumerable sets. Indeed, the proof given in [14] preserves the parallelism of the original definition of time-varying distributed systems. Accordingly, we are also rather near to the frontier between decidability and undecidability under the complementary assumption that parallelism should be preserved. It is clear that it is important to study the status of $TVDH_3$ under this assumption.

Acknowledgement

The author acknowledge the very helpful contribution of *INTAS project* 97-1259 for enhancing their cooperation, giving the best conditions for producing the present result.

References

[1] Ferretti C., Mauri G., Zandron C., Nine test tubes generate any RE language, TCS special issue on *MCU'*98, to appear.

[2] Gruska J., Foundations of computing. International Thomson Computer Press, (1997).

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

[4] Head T., Păun Gh., Pixton D., Language theory and molecular genetics. Generative mechanisms suggested by DNA recombination, chapter 7 in vol.2 of G.Rozenberg, A.Salomaa, eds., *Handbook of Formal Languages*, 3 volumes, Springer-Verlag, Heidelberg, 1997.

[5] J.E. Hopcroft, J.D. Ullman, Introduction to automata theory, languages and computation. Addison-Wesley Publishing Company, (1979).

[6] Kleene S.C., Introduction to Metamathematics. Van Nostrand Comp. Inc., New-York, 1952.

[7] Landweber L., Kari L., The evolution of cellular computing: Nature's solution to a computational problem. In Preliminary proceedings, Fourth International Meeting on DNA Based Computers, June 15 - 19, 1998, University of Pennsylvania, 3–13 (1998).

[8] Malcev A.I., Teorija rekursivnykh funkcij. Moskva, Nauka, 1965 (in Russian).

[9] Margenstern M., Non-erasing Turing machines: a new frontier between a decidable halting problem and universality. Lecture Notes in Computer Science, **911**, 386-397, in *Proceedings of LATIN'95* (1995).

[10] Margenstern M., Frontier between decidability and undecidability : a survey. to appear in *Theoretical Computer Science*.

[11] Margenstern M., Rogozhin Yu., A universal time-varying distributed H-system of degree 2. In Preliminary proceedings, Fourth International Meeting on DNA Based Computers, June 15 - 19, 1998, University of Pennsylvania, 83–84 (1998).

[12] Margenstern M., Rogozhin Yu., A universal time-varying distributed H-system of degree 2, to appear in *Biosystems*.

[13] Minsky M.L. Computation: Finite and Infinite Machines. Prentice Hall, Englewood Cliffs, N.J., 1967.

[14] Păun A., On Time-Varying H Systems, *Bulletin of EATCS*, N°67, 157–164, February 1999.

[15] Păun G., Rozenberg G., Salomaa A., Computing by splicing, Theoretical Computer Science. **168-2**, special issue on Universal Machines and Computations, 321–336 (1996).

[16] Păun G., DNA computing: distributed splicing systems. in *Structures in Logic and Computer Science. A Selection of Essays in honor of A. Ehrenfeucht*, Lect. Notes in Computer Sci. 1261, Springer-Verlag, 353–370 (1997).

[17] Păun G., DNA Computing Based on Splicing: Universality Results. Proceedings of MCU'98, vol.I, ISBN 2-9511539-2-9, 67-91 (1998).

[18] Păun G., Rozenberg G., Salomaa A., DNA Computing: New Computing Paradigms. Springer, 1998.

[19] Pixton D., Regularity of splicing languages. Descrete Applied Mathematics, 69 (1-2), 99–122 (1996).

[20] Priese L. Towards a precise characterization of the complexity of universal and nonuniversal Turing machines. SIAM Journal of Computation, **8**, 4, 508-523 (1979).

[21] Priese L., Rogozhin Yu., Margenstern M., Finite H-Systems with 3 Test Tubes are not Predictable. In Proceedings of Pacific Symposium on Biocomputing, Kapalua, Maui, January 1998 (R.B.Altman, A.K.Dunker, L.Hunter, T.E.Klein, eds), World Sci. Publ., Singapure, 545–556 (1998).

[22] Rogozhin Yu.V., Sem' universal'nykh mashin T'juringa. Matematicheskie Issledovanija, **69**, 76-90 (1982) (Seven universal Turing machines) (in Russian).

[23] Rogozhin Yu.V., Small universal Turing machines, Theoretical Computer Science, **168-2**, special issue on Universal Machines and Computations, 215-240 (1996).

[24] Rogozhin Yu.V., A Universal Turing Machine with 22 States and 2 Symbols. Romanian Journal of Information Science and Technology, vol.1, no. 3, 259–265 (1998).

[25] K. Wagner, G. Wechsung, Computational Complexity, VEB Deutscher Verlag der Wissenschaften, Berlin, 1986.

Imprimé en France Dépot légal octobre 1999 ISBN 2-9511539-5-3