U.P.B. Sci. Bull., Series C, Vol. 71, Iss. 4, 2009 ISSN 1454-234x

DNA COMPUTING – MODELLING AND SIMULATING A MOLECULAR TURING MACHINE

Mihnea MURARU[1], Matei-Dan POPOVICI[2]

Folosind paradigma "DNA Computing", explicată în detaliu în cadrul articolului, construim un model de calculator molecular – o Maşina Turing moleculară. Detaliem operațiile moleculare întrebuințate şi cadrul de aplicabilitate al acestora. Apoi, pentru modelul astfel construit, realizăm un simulator comportamental al acestuia.

Evidențiem necesitatea dezvoltării unei codificări (a unui limbaj) pentru modele computaționale şi prezentăm în detaliu problema generării limbajului în contextul nostru precum şi soluția găsită.

Using "DNA Computing", a form of computation detailed further in the article, we design a model for a molecular computer – a Molecular Turing Machine. We explain the applicable molecular operations and their extensibility on the model. Finally we develop a Simulator aimed at studying the behavior of the Molecular Turing Machine.

As part of the Simulator development, we emphasize the necessity for finding a codification or language for computation models and go into detail on the language generation problem for our Molecular Turing Machine and describe our solution.

Keywords: DNA Computing, Turing Machine, NP-complete problems, language generation problem for DNA-based computational models

1. Introduction

"DNA Computing" is a form of computation which, as opposed to traditional silicon-based technologies, uses DNA and biochemical processes.

The basis of this concept was set in 1994 by *Leonard Adleman* [1] from University of Southern California, which proved the efficiency of using DNA for computation.

After a week-long experiment, Adleman succeeded in solving a hamiltonian-path NP-complete problem, using strictly biochemical processes and DNA.

[1] PhD student, Faculty of Automatic Control and Computers, University POLITEHNICA of Bucharest, Romania, mmihnea@gmail.com `

[2] PhD student, Faculty of Automatic Control and Computers, University POLITEHNICA of Bucharest, Romania

The solution was found in less than a second, as opposed to 54 seconds necessary to solve the same problem on a computer. The laborious operations for isolating the molecule representing the problem solution proved to be time-consuming. The computation nevertheless was done almost instantaneously.

In his paper, “Computing with DNA”[1], Adleman describes in detail his experiment. Following his work scientists turned their attention to DNA Computing; seeking to solve other NP-complete problems (such as Travelling Salesman Problem, or the k-clique covering problem) different DNA-based models were designed.

Researchers went further and tried to develop general computation models using DNA. The first Turing-Machine models appeared. The question if P=NP came into attention in correlation with the massive parallelism of DNA. DNA Computing seemed to be a basis for developing new ultra-performance hardware and software technologies. Studies show that on a traditional computer, for running 10^9 operations, 1 Joule of energy is required. By using DNA and molecular operations and using the same amount of energy, $2 \cdot 10^9$ operations can be performed.

In 2002, researchers at Weizmann Institute of Sciences in Rehovot, Israel, built the first programmable molecular computer; in this computer enzymes and DNA molecules replace the silicon microchips. The computer is designed for analyzing cancer activity in a cell, and producing anti-cancer treatment.

Starting from all available theoretical and practical results related to DNA Computing, and using discoveries in molecular engineering and genetics, we attempt to develop an efficient Molecular Turing Machine model using molecular operations.

Our model will contain a Turing Machine structure (a specific DNA encoding for all elements that constitute the Turing Machine) and a sequence of operations that performed one after the other, will produce a transition of the machine.

Using this model we will create a simulation program, designed as a distributed system running in a typical computer network environment. We will use the Simulator to run experiments and we will monitor the evolution of our model, transition by transition, towards the final state or solution.

2. DNA Computing and the Turing Machine

The Turing Machine concept is based on running a well-defined procedure that produces changes and head movement on an infinite symbol-tape. The symbols set is finite and defines the Turing Machine alphabet. The head position defines the current symbol to be read and possibly overwritten. The head can move one position at a time either to the left or to the right. A Turing Machine transition changes the current state of the machine, and also makes the head write a symbol at the current position or move to the left or right.

If from a given state and current symbol on tape there is more than one transition, then the Turing Machine is nondeterministic. For that state-symbol combination the Turing Machine can perform different transitions simultaneously. Based on this, we can picture a tree structure, where the root is the Turing Machine in its initial state, and the leaves are Turing Machines in their current (possibly final) states. If a Turing Machine can perform nondeterministic transitions to different states, it will “instantiate” itself several times and each instance will perform one of the nondeterministic transitions, thus completing the “tree” with all possible threads of execution. Each “instantiated” Machine will run in parallel with all the other Machines and in its turn, may “instantiate” other machines. The execution will come to a stop when the final state is reached, or the entire input has been consumed.

We can use the nondeterministic Turing Machine described briefly above to solve exponential problems (problems with an exponential complexity) such as the Travelling Salesman Problem. There problems are classified as NP-complete problems (NP stands for Nondeterminist Polynomial). As the name suggests, there are problems that can be solved in polynomial time on a nondeterministic machine such as the Turing Machine.

Unfortunately for us, using current technologies, we cannot build nondeterministic machines. Having in mind the current hardware concepts, we could compare a nondeterministic machine (a Turing Machine for example) with a system that has an infinite number of processors available, that can run in parallel at any given time. Such a system is impossible to build nowadays.

Nevertheless, using DNA Computing we could efficiently mimic such a massive parallelism required for a Turing Machine to run.

The DNA molecule is comprised by a double-helix sequence of nitrogen bases: Guanine, Citosine, Tymine and Adenine. These bases can be associated with base-4 numbers in the same way 0 and 1 represent the digits of a base-2 number.

A single chromosome from the human body contains a sequence of 220 million bases, and a single bacterium contains a hundred thousand DNA molecules of different lengths.

This huge storing capacity and the great number of DNA molecules that exist in different life forms give us a clue about the great potential of using DNA for our parallel computations.

Going further, we notice that nature works with DNA in a very similar way computers work with data.

The DNA molecule, in the presence of specific enzymes, can create copies of itself, can be cut at precise intervals or can join another DNA molecule. Using lab procedures, we can classify and isolate DNA molecules by length (number of contained bases).

Looking at the big picture, we have a very big number of molecules, pseudoinfinite in comparison to the size of our problems, and we have chemical reactions that involve simultaneously all molecules from a substance. In other words we have the premises for designing a Molecular Turing Machine.

3. The Molecular Turing Machine Model

Our Molecular Turing Machine Model is an adaptation of the Turing Machine Model described in: „Molecular Computing Machines” by Yaakov Benenson and Ehud Shapiro.

The Molecular Turing Machine will be represented by a circular DNA molecule. Using the four nitrogen bases described above we will perform an encoding of the symbols on the tape, for the machine head and for the states.

The head will be encoded by a combination of two recognition sites. We will call them *Inv* and *St*. (The concept of recognition site will be explained further in the article). This head structure is related to the way a transition in the Turing Machine occurs. To execute a transition, the Turing Machine head will be split from the molecule; the splitting will be done by enzymes that recognize *Inv* and *St* patterns (more details about this procedure in the following section).

The states of the Turing Machine will have a particular encoding in such a way that each state has a different length (number of nitrogen bases) from all the others. The encoding for the current Turing Machine state will be placed immediately after the head.

The head and the current state are followed by a list of encoded symbols, starting with the current symbol. Each symbol is flanked by *L*(eft) and *R*(ight) patterns.

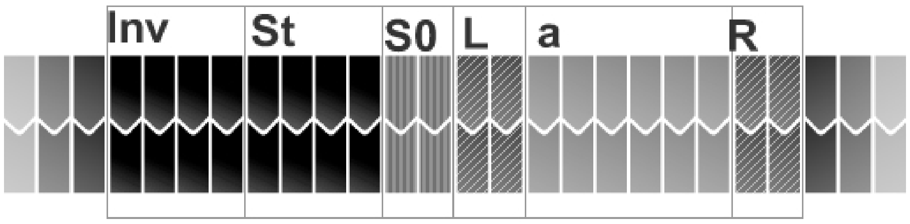

Fig. 1. Molecular Turing Machine structure

The key of the Turing Machine Model is the way transitions are performed. In essence, to execute a transition, the current head and state are split, then a special Transition Molecule will replace them. The Transition Molecule contains a new head, a new state (the final state of the transition) and the new symbol to be written on the tape. In the final step, the molecule will attempt to close and become circular.

All residual elements such as old heads and states are eliminated from the substance using procedures described above.

In the following section, we will go into detail about molecular operations and the necessity of such a structure for both the Turing Machine Molecule and the Transition Molecule.

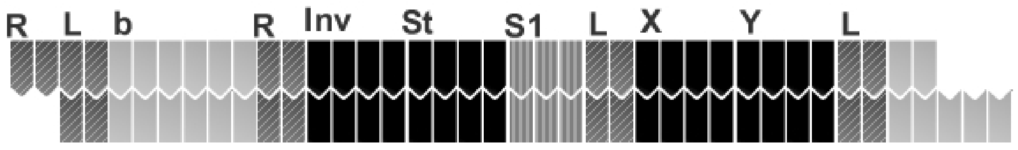

Fig. 2. The Transition Molecule structure

4. Machine Transitions

The execution of transitions is based on the following molecular operations: split of a molecule, join of two different molecules or of the two ends of the same molecule (in order to become circular). We also use the following higher-level operations: electrophoresis – elimination of molecules upon length criteria, replication – generation of molecule copies, and ATP-dependent remove – elimination of noncircular molecules. We shall detail these operations below.

Molecule splitting is done by restriction enzymes. They make two incisions, one in each of the two DNA strands in the molecule. The incisions separate the DNA bases in fixed points. It is believed that this splitting mechanism was developed by bacteria, in order to eliminate viral DNA sequences.

The way a restriction enzyme splits is defined by the following three parameters:

- Recognition site – the DNA sequence a certain enzyme recognizes
- Offset – the displacement where the splitting begins relative to the position where the recognition site was identified
- Cut length – the cut span, measured as the number of DNA bases.

The cutting process is presented in the following pictures:

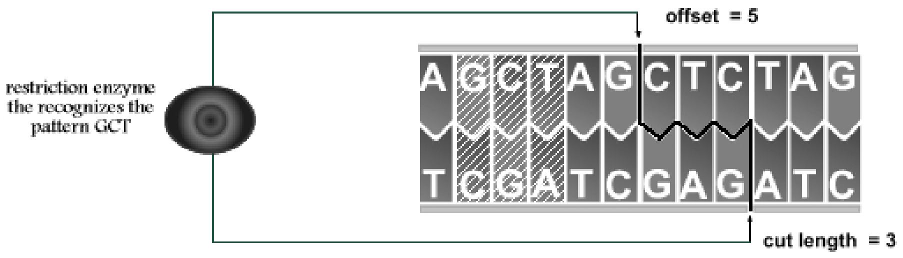

Fig. 3. Recognition site identification and the actual split

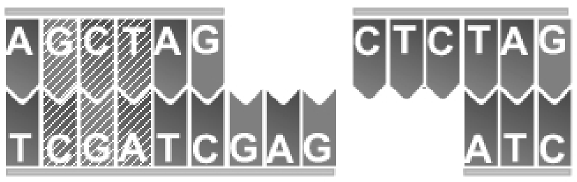

Fig. 4. Two molecules resulted from the split

The two incisions in the DNA strands do not usually overlap (they aren’t made in the same pair of complementary bases). This behavior is important because it enables molecules to join in the future. The two resulted free ends, which do not have a correspondent in the complementary strand, are called *sticky-ends*. They will cause the molecule to join another molecule with a complementary sticky-end.

The coupling of molecules is possible in the presence of a certain enzyme, called *ligase*. Joining is the reverse process of splitting. Two nearby molecules having complementary sticky-ends will attract to each other, forming weak bonds between complementary DNA bases. Ligase will glue the strands together, giving birth to a new molecule. Ligase can also make the two ends of the same molecule join, causing it to become circular.

Electrophoresis is a laboratory procedure for classifying the molecules in the solution based on their lengths. It shall be used to eliminate the molecules having their length outside a certain interval. The procedure is based on the movement tendency of molecules in the presence of an electric field. The forces acting upon a molecule are: the electrostatic force, the friction force, and the electrophoretic delay force (particular for that solution). The larger a molecule, the smaller the resulting force, producing a slower movement. The process comprises several steps:

- First, an electrophoretic, electrically conductive gel is added to the solution, at one end of the container with the DNA molecules;
- Two electrodes are added, generating an electric field;

- The smaller molecules will traverse the gel faster, while the larger ones slower. Eventually, it will be possible to classify the molecules having a certain length based on the distance they travelled throughout the gel.

We will use these operations in order to execute machine transitions. The following steps are present during each transition:

- Restriction enzymes that identify the *St* and *Inv* recognition sites are added to the solution. This will cause the circular molecules encoding Turing Machines to break. The length of the current state and the encoding of the current symbol guarantee that the resulted sticky-end uniquely encodes the state-symbol combination.
- The restriction enzymes are removed and the transition molecules and ligases are added to the environment. The transition molecules that encode a certain state-symbol combination will join Turing molecules exposing the complement of the same combination, in the presence of ligases.
- The residual elements (machine heads, uncoupled transition molecules etc.) are removed by electrophoresis.
- As a result of the concatenation between transition and Turing molecules, the old tape symbol is rebuilt. This is why we must now remove it, using the *X* and *Y* restriction enzymes.
- Residual components are removed and the Turing molecules are stimulated to close (we remind that the consistent configuration of Turing molecules is circular). At this point the transition is concluded.

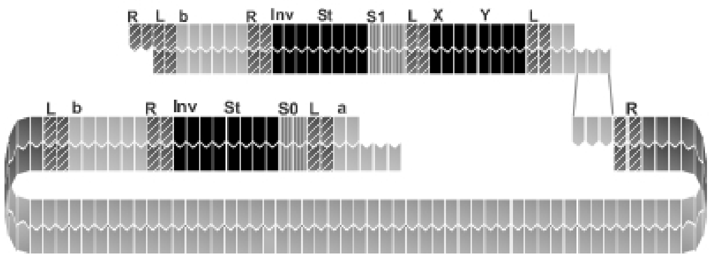

Fig. 5. Coupling between a Turing molecule (down) and a transition molecule (up)

5. Language generation for the Molecular Turing Machine

Given a Turing Machine with a fixed number of states and symbols, our target is to generate a valid language for it using the four DNA bases (A, C, G, T).

The encoding cannot be chosen randomly. As stated in the previous section, a machine transition is made possible through a series of splits and joins of a circular molecule. Splits are done by restriction enzymes, which spot certain recognition sites, and cut the molecule at a fixed offset from the beginning of the site. It is impossible to predict or control the exact moment of splitting, or the place in case the molecule contains the same recognition site in several spots. This is why a recognition site must not appear in combinations of states, symbols etc.

The previous section described the fusion between a split Turing molecule and a transition molecule. This process raises two issues. First, the *St* enzyme must cut somewhere inside the span of the current symbol, and not outside it. Secondly, the resulted sticky-end must uniquely encode the current state-symbol combination. These remarks are formalized by the following invariants:

I1 : The four recognition sites are distinct DNA sequences (will not appear in other combinations in the molecule)

I2 : A restriction enzyme cuts either a complete L or R sequence, or inside the span of the current symbol (never somewhere else)

I3 : The stick-ends resulted from cutting inside the span of a symbol and the L and R sticky-ends are all distinct

We add one more invariant:

I4: The encodings of different states have different lengths

The need for *I4* is explained further.

It shows up that it is difficult and time-consuming to generate a language that obeys only the constraints formulated above. The exponential nature of the generation problem is very pregnant. This is why, besides the Verifications that must be passed in order to satisfy the invariants, we introduce additional Rules (constraints) which lower the complexity of the generation process.

The generation process comprises five steps. In each step, distinct elements are randomly generated (distinct with respect to one another and to elements generated in previous steps); the specific Rules for the current step are checked; as long as a generated element creates conflicts, it is regenerated. A step is said to be complete when all the corresponding elements are successfully added to the language.

The five steps are ordered in a specific manner. The first step generates the base sequences that impose restrictions for all the following steps. The second step finds the sequences *L* and *R*. The third step computes the parameters of the restriction enzymes. The fourth step generates the sequences for the machine states, each having a different length. Finally, the fifth step yields the encodings of the machine symbols. It is obvious that each step places additional restrictions over the steps to follow.

Remark 1: DNA strands are very similar with strings of characters. We will use the term "concatenation" to designate the string operation extended to DNA strands. Also, the expression:

$$a = bc$$

denotes a strand a, *resulted from concatenation of strands* b *and* c.

Remark 2: This section introduces several terms with special meaning:

Invariant – assertion about a certain feature of the molecular model. The invariants completely describe the model from the language generation perspective.

Verification – expression that must be true in order to satisfy one or more invariants.

Rule – additional constraint which simplifies the generation of a valid solution.

We go into detail about each step of the generation process.

The base sequences are the enzyme *recongnition sites*. Most of the restrictions formulated below are related to them. Let

$$REC = \{Inv, St, X, Y\} \tag{1}$$

be the set of base sequences.

We write *rsl* to denote the length of a recognition site:

$$rsl = \#(rec_i),\ rec_i \in REC \tag{2}$$

rsl is constant and it is an input parameter for the generation process. In other words, all the recognition sites have the same length, which can be set by the user. $\#(s)$ stands for the length of sequence *s*.

In this first step, the following verifications are to be made:

V1 : All the sequences in REC are distinct

V2: Combinations InvSt and YX (resulted from concatenation) do not contain another sequence from REC

Verification *V2* isn't very intuitive, but, analyzing the structure of the machine head and transition molecules (presented in the previous sections), we can see that, by bringing together two sequences (*Inv-St*, *Y-X*), a possible unwanted recognition site may form inside the concatenated sequences. Thus, we need *V2* in order to satisfy *I1*.

The length of these base sequences is of no relevance. In contrast, the lengths of *L* and *R* deserve some special attention. If *L* and *R* have a small length, we must test the existence of a sequence in *REC* in a multitude of combinations of the form:

$$S_k L a_i\ ,\ S_k \in \text{states set},\ a_i \in \text{symbols set} \tag{3}$$

The number we combinations we get is:

$$no.comb = |states\ set| * |symbols\ set| \tag{4}$$

where $|M|$ represents the number of elements of set *M*.

It is possible to reduce the number of combinations by adding the constraint that the length of *L* should be greater or equal to the length of the base sequences:

$$\#(L) \geq rsl \tag{5}$$

This constraint guarantees that it is impossible for a base sequence to span over all three symbols $S_k L a_i$, but over at most two: $S_k L$ or $L a_i$.

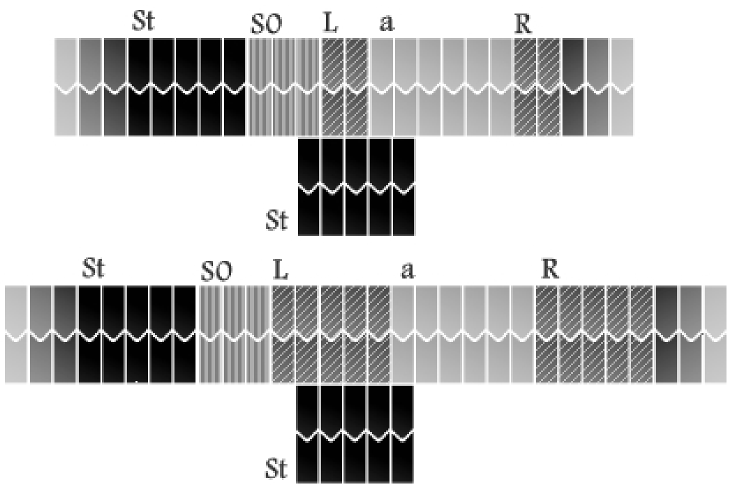

Fig. 6. The length constraint for *L* and *R*

The same applies for *R*.

For simplicity, we will choose the lengths of *L* and *R* to be equal to those of the base sequences. We get the following rules:

$$R1\text{: } \#L = \#R \tag{6}$$

$$R2\text{: } \#R = rsl \tag{7}$$

With these modifications, the number of combinations to be checked becomes:

$$no.comb = |states\ set| + |symbols\ set| \tag{8}$$

which is significantly smaller than the previous result. We also get another important advantage: now we need only to check the two-symbol combinations $S_k L$ and $L a_i$. This advantage will become useful when generating the other elements.

The following verifications guarantee that the invariant *I1* is satisfied:

$$V3\text{: } RInv \not\supseteq rec_i,\ \ rec_i \in REC \tag{9}$$

$$V4\text{: } LY \not\supseteq rec_i,\ \ rec_i \in REC \tag{10}$$

$$V5\text{: } XL \not\supseteq rec_i,\ \ rec_i \in REC \tag{11}$$

$$V6\text{: } RL \not\supseteq rec_i,\ \ rec_i \in REC \tag{12}$$

By $a \not\supseteq b$ we denote that the string *a* doesn't contain the string *b* (as a substring).

In other words, the combinations *RInv, LY, XL* and *RL* must not contain any other recognition sites. Analyzing the structure of the molecules we can see that these combinations appear in different contexts.

One last verification needed to satisfy the invariant *I3* is:

$$V7:\ R \neq L \tag{13}$$

If the sticky-ends resulted from splitting *L* and *R* were identical, we would get unwanted fusions of molecules.

In order to compute the enzyme parameters, we define the following:

S_{min} – minimum state length (14)

$S_{max} = S_{min} + no.states$ – maximum state length (15)

S_{min} has a standard value of 3, obtained experimentally (it speeds up the states generation).

The enzyme parameters to be established are: the offset where the splitting begins relatively to the recognition site location and the cut length. Their computation has to consider the invariants *I2* and *I3*. Choosing a valid offset guarantees that *I2* is satisfied. We get the verification:

$$V8:\ offset = rsl + \#(L) + S_{max} \tag{16}$$

This verification ensures the fact that the splitting will always start inside the span of the symbol, and not before it. When determining the symbol length we will make sure that the splitting doesn't end beyond the symbol).

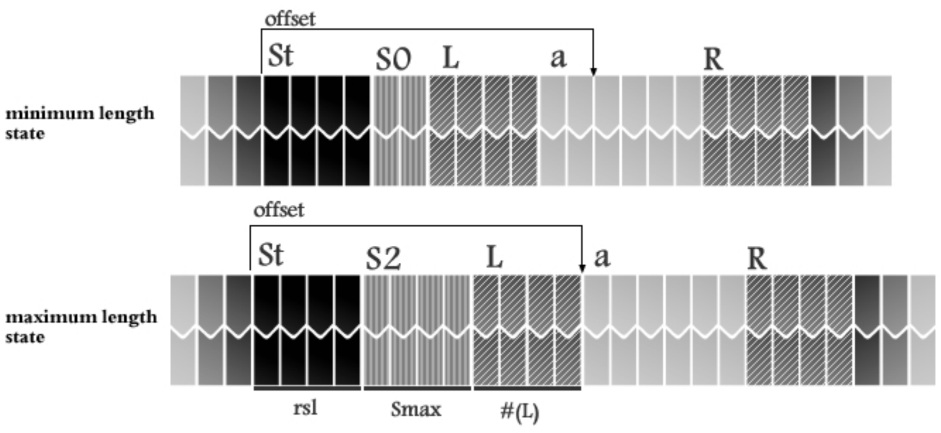

Fig. 7. Choosing the offset in order to guarantee that the splitting takes place inside the symbol

The cut length is important for ensuring that *I3* is satisfied. For example, if the cut length equals 2, and the Turing Machine has 20 states and 2 symbols, then, using 2 “characters” (DNA bases), we are able to encode only $2^4 = 16$ state-symbol combinations (each combination corresponding to a sticky-end). In fact, we need 20 * 2 = 40. We get the verification:

$$V9\text{: } cutlen = \lceil log_4(no.states * no.symbols) \rceil * speedfactor \tag{17}$$

The product inside the logarithm represents the number of all state-symbol combinations. The logarithm will give the number of “characters” needed to encode all these combinations. Base 4 comes from the 4 possible values of a “character” (there are 4 DNA bases). *speedfactor* is used to control the generation speed of the sticky-ends encoding state-symbol combinations. When its value is 1, *cutlen* represents the smallest necessary number of characters for encoding the needed combinations. Ensuring the uniqueness of the sticky-ends for this smallest number proves to be difficult. Increasing the *speedfactor* (in fact the cut length) makes it easier to obtain unique sticky-ends. A good value for *speedfactor* is 2:

$$cutlen^1 = 2 * cutlen \tag{18}$$

Before presenting the states generation method, we remind that distinct states must be encoded with different lengths (*I4*).

$$V10\text{: } \#(S_i) \neq \#(S_j) \ (\forall)\ S_i, S_j \in states\ set \tag{19}$$

The solution at hand is to generate states each with a unit longer than the previously generated state, as suggested by the above definition of S_{max}. The verifications each state must satisfy are:

$$V10\text{: } S_k L \not\supseteq rec_i,\ \ rec_i \in REC,\ \ \ S_k \in states\ set \tag{20}$$
$$V11\text{: } St S_k \not\supseteq rec_i,\ \ rec_i \in REC,\ \ S_k \in states\ set \tag{21}$$

These verifications aim at satisfying *I1*; the combinations above are tied to the molecule structure. Sequences *L* and *St* are found in certain cases next to states.

The states set is built iteratively; the currently generated state must meet *V10* and *V11* in order to be added to the set.

Symbol generation is the most complex part of the process. The first issue is establishing the length of a certain symbol. For simplicity, we assume that all symbols have the same length.

$$R3\text{: } \#(a_i) = \; const. \; (\forall) \, a_i \in symbols \; set \tag{22}$$

I2 states that the restriction enzymes should cut inside that span of the symbol. Thus, we get:

$$V12\text{: } \#(a_i) = S_{max} - S_{min} + cutlen + 1 \tag{23}$$

In the case of the shortest state, the enzyme will cut most deeply inside the span of the symbol, whereas, for the longest state, it will cut exactly at the beginning of the symbol.

Knowing that, for the longest state the cutting begins right after the first character (DNA basis), and that the offset at which the cutting begins is the same for any state, we conclude that for the shortest state the cut will be done at the character $S_{max} - S_{min} + 1$ (the length difference between the longest and the shortest states). Thus, a symbol must be encoded using at least $S_{max} - S_{min} + 1$ characters (DNA bases) to ensure that, in case of the shortest state, the cutting will begin still inside the span of the symbol. Since the cut spans *cutlen* characters, it means that the symbol needs at least $S_{max} - S_{min} + cutlen + 1$ characters.

Once we have computed the length of the symbols, we must satisfy the verifications:

$$V13\text{: } a_m \neq \; a_n, \;\; (\forall) a_m, a_n \in symbols \; set \tag{24}$$
$$V14\text{: } La_kR \nsupseteq rec_i \, , \;\; rec_i \in REC, \;\; a_k \in symbols \; set \tag{25}$$

V14 guarantees that a symbol flanked by *L* and *R* will not contain any recognition sites, as stated by *I1*.

The next verification ensures the uniqueness of the sticky-ends resulted from symbol splitting. We define:

$$se_{im} = the \; sticky - end \; resulted \; from \; splitting \; the \; combination \; s_i a_m, \\ s_i \in states \; set, \;\; a_m \in symbols \; set \tag{26}$$

We now state:

V15:

$$se_{im} \neq \; se_{jm} \, , \\ (\forall) s_i, s_j \in states \;\; set, (\forall) a_m, a_n \in symbols \; set, i \neq j, m \neq n \tag{27}$$

This verification is implemented by first generating a random encoding for the current symbol and ensuring its uniqueness. Then, the generated sequence is successively concatenated with every state and then a cut is made. Each cut produces a sticky-end. If a sticky-end is identical to one generated before (either for a previous symbol, or for the current symbol and a previous state), then the symbol encoding is discarded and a new one is generated. When all the conditions are fulfilled, the encoding is added to the language and the sticky-ends resulted from symbol splitting are stored in a collection, for future checks.

6. Conclusions

By implementing our solution for language generation (as presented in the previous section), we have built a Simulator for the Molecular Turing Machine[2]. It enables us to run virtual experiments and observe their evolution. The simulator models an “experimental jar, that contains DNA molecules encoding Turing Machines and their transitions, restriction enzymes, ligases etc. We have also implemented specific operations, as described in section 4: join, split, electrophoresis, replication, and ATP-dependent remove.

Architecturally, the Simulator has a multilevel structure and runs in a distributed environment, where several worker computers run experiments locally, and a server gathers experiment related data for statistical processing.

Experimentally, we concluded that the probability of reaching a final configuration (even if it exists theoretically) is low. The probability is closely tied to the number of experiment steps (maximum number of transitions the Turing Machine can make): the greater the running time, the greater the probability. We can also raise the possibility of reaching a solution by increasing various experiment parameters (number of DNA molecules, enzymes concentration) and by running in parallel several independent experiments.

R E F E R E N C E S

[1] *Leonard Adelman*, “Computing with DNA”, Scientific American, Aug 1998
[2] *Mihnea Muraru, Matei-Dan Popovici*, “DNA Computing – Modelling and Simulating a Molecular Turing Machine” Graduation Paper, June 2008
[3] *Cristian A. Giumale*, “Introducere in Analiza Algoritmilor”, Ed. Polirom Bucuresti, 2004
[4] *Yaakov Benenson, Ehud Shapiro*, “Molecular Computing Machines”, Dekker Encyclopedia of Nanoscience and Nanotechnology, 2004