*Natural Computing* **1:** 333–390, 2002.
© 2002 *Kluwer Academic Publishers. Printed in the Netherlands.*

# Molecular computing paradigm – toward freedom from Turing's charm

TAKASHI YOKOMORI
*Department of Mathematics, School of Education, Waseda University, Tokyo 169-8050, Japan (E-mail: yokomori@mn.waseda.ac.jp)*
*and CREST, JST, 4-1-8 Honmachi, Kawaguchi-chi, Saitama, 332-0012, Japan*

**Abstract.** This article gives a concise but intensive survey on one of the subfields of natural computing – *Molecular Computing*. Starting with making a brief revisit to Adleman's pioneering work, this paper will give an overview of selected topics of the field from theory to experiments, while the stress is primarily put on research of theoretical achievements of molecular computing models.

**Key words:** molecular computing, molecular machines, self-assembly computation, splicing systems

## 1. Introduction

### 1.1. *Molecular computing as natural computing*

Molecular computing, a main theme in this paper, is a novel computing paradigm inspired by biochemical nature of biomolecules, and in the early days it has been understood that molecular computing purposes to replace silicon-based hardware with biological one by developing molecular-based computers. Unlike this intention of the early schema, however, by now it is mostly recognized that molecular computers should be the ones not to substitute for existing computers but to complement them.

It is not so recent that the term "molecular computers" has appeared in literature. In fact, one can go back to an article of *Biofizika* in 1973 where an idea of cell molecular computers is discussed (Vaintsvaig and Liberman, 1973). Since 1974, Conrad, one of the pioneering researchers in this area, has been conducting consistent research on the information processing capability using macromolecules (such as proteins) (Conrad, 1985), and has edited a special issue entitled "Molecular Computing Paradigms" in (Conrad, 1992). These initial efforts in the short history of molecular computing have been succeeded by Head's original work on splicing systems and languages in (Head, 1987) whose theoretical results on splicing phenomena should be

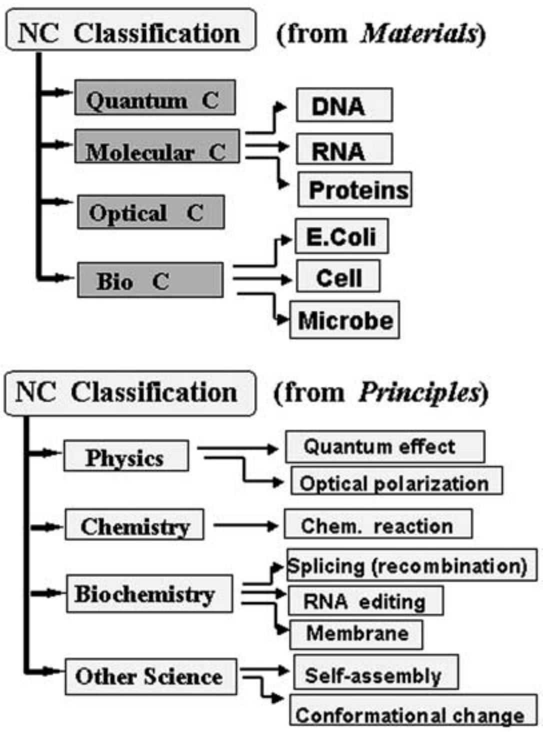


*Figure 1.* A variety of natural computing paradigms.

highly appreciated, in that it was the first achievement of mathematical analysis on biochemical operations of DNA recombination.

It is possible to discuss in general about new computing paradigms in the schema of "Computing with *X*", where *X* can range from various sorts of materials to those of principles. Figure 1 summarizes a variety of natural computing paradigms proposed so far within the "computing with *X*" schema.

## 1.2. *About this article*

This paper is prepared so as to put more stress on theoretical results of molecular computing models, from the formal language theoretic point of view, than any other issues in this new emerging area. The table of content of this article is given here.

Selection of topics included in the article simply reflects on the author's taste, and this paper does not intend to give an extensive survey on the area of molecular computing. Instead, the author preferred to give a rather concise but intensive survey for each chosen topic.

As for some of the prior extensive surveys, one can find Reif (1998, 2002), and other surveys include Kari (1996), Kari and Landweber (1999), Rozenberg and Salomaa (1999), Hagiya (1999). In particular, Păun et al. (1998) is the first textbook emphasizing the formal language theoretic aspect of the discipline. Further, for the reader who may have interests in something more broader subjects in the area, a comprehensive list of reference papers is provided to partially make up for the author's bias.

## 1.3. *Basic notions and notations*

We summarize the basic notions and notations required for understanding this article.

Due to the Watson-Crick complementarity, completely hybridized double stranded molecules can be regarded as strings over an alphabet {A, C, G, T}. Therefore, it is natural for studying and formulating notions in molecular computing to employ the formal framework of the classical formal language theory.

For a set $X$, $|X|$ denotes the cardinality of $X$. The power set of $X$ (that is, the set of all subsets of $X$) is denoted by $\mathcal{P}(X)$.

An *alphabet* is a nonempty finite set of symbols. For an alphabet $V$, $V^*$ denotes the set of all strings (of finite length) over $V$, where the empty string is denoted by $\lambda$. By $V^+$ we denote the set $V^* - \{\lambda\}$. A *language* over $V$ is a subset of $V^*$.

A collection of languages over an alphabet $V$ is called a *family of languages* (or *language family*) over $V$. A language family $\mathcal{FL}$ is closed under an $n$-ary operation $g$ if for any $L_1, \ldots, L_n$ in $\mathcal{FL}$, $g(L_1, \ldots, L_n)$ is also in $\mathcal{FL}$ over $V$, where each $L_i$ is over $V$.

A *phrase-structure grammar* (or *Chomsky grammar*) is a quadruple $G = (N, T, S, P)$, where $N$ and $T$ are disjoint alphabets of nonterminals and terminals, respectively, $S (\in N)$ is the initial symbol, $P$ is a finite subset of $(N \cup T)^* N (N \cup T)^* \times (N \cup T)^*$. An element $(u, v)$ of $P$, called rule, is denoted by $u \to v$.

For strings $x, y \in (N \cup T)^*$, let

$$x \Longrightarrow_G y \overset{\text{def}}{\Longleftrightarrow} \exists x_1, x_2 \in (N \cup T)^* \text{ and } u \to v \in P \text{ such that}$$
$$x = x_1 u x_2, \;\; y = x_1 v x_2.$$

A language generated by $G$ is defined as $L(G) = \{x \in T^* \mid S \Longrightarrow_G^* x\}$.

A phrase-structure grammar $G = (N, T, S, P)$ is

(1) *context-sensitive* if for $u \to v \in P$ there exist $u_1, u_2 \in (N \cup T)^*$, $A \in N$, $x \in (N \cup T)^+$ such that $u = u_1 A u_2$, $v = u_1 x u_2$.

(2) *context-free* if for $u \to v \in P$, $u \in N$.

(3) *regular* if for $u \to v \in P$, $u \in N$ and $v \in T \cup TN \cup \{\lambda\}$.

By $\mathcal{RE}$, $\mathcal{CS}$, $\mathcal{CF}$ and $\mathcal{REG}$, we denote the families of languages generated by (arbitrary) grammars, context-sensitive grammars, context-free grammars, and regular grammars, respectively. $\mathcal{RE}$ comes from the alternative name of *recursively enumerable* language. Further, $\mathcal{FIN}$ denotes the family of all finite languages.

The following inclusions (called *Chomsky hierarchy*) are proved.

$$\mathcal{FIN} \subset \mathcal{REG} \subset \mathcal{CF} \subset \mathcal{CS} \subset \mathcal{RE}.$$

A (nondeterministic) *Turing machine* is a construct $M = (Q, V, T, s_0, F, \delta)$, where $Q$, $V$ are finite sets of states and tape alphabet, respectively. $T(\subseteq V)$ is an input alphabet, $s_0(\in Q)$ is the initial state, $F(\subseteq Q)$ is a set of final states, and $\delta$ is a function from $Q \times V$ to $\mathcal{P}(Q \times V \times \{L, R\})$ acting as follows: for $s, s' \in Q, a, b \in V$, $d \in \{L, R\}$, if $(s', b, d) \in \delta(s, a)$, then $M$ changes it state from $s$ to $s'$, rewrites $a$ as $b$, and moves the read-only head to left (if $d = L$) or right (if $d = R$). $M$ is called *deterministic* if for each $(p, a) \in Q \times V$, $|\delta(p, a)| \leq 1$.

A language $L(M)$ recognized by $M$ is defined as the set of input strings such that, starting with the initial state, they drive the computational process of $M$ to a final state.

It is known that the family of languages recognized by Turing machines is equal to the family $\mathcal{RE}$.

As subfamilies of $\mathcal{RE}$, two families are of special importance. $\mathcal{NP}$ and $\mathcal{P}$ are the families of languages recognized in polynomial-time by Turing machines and deterministic Turing machines, respectively. It remains open whether $\mathcal{P} = \mathcal{NP}$ or not, while the inclusion $\mathcal{P} \subseteq \mathcal{NP}$ is clear from the definition. Further, a subfamily of $\mathcal{NP}$ called *NP-complete* is of special interest, because NP-complete languages are representative of the hardness in computational complexity of $\mathcal{NP}$ in the sense that any language in $\mathcal{NP}$ is no harder than NP-complete languages.

## 2. Adleman's initiative

There would be no doubt that Adleman's experimental work (Adleman, 1994) appeared in the journal *Science* is a landmark in its short history of the area of molecular computing.

Taking up a small instance of a well-known problem "Directed Hamiltonian Path Problem (DHPP)", Adleman has shown how to solve the problem using molecules together with bio-chemical experimental techniques.

DHPP is a graph problem where given a directed graph $G$ with two specific nodes called "Start" and "Goal" one must decide if there exists a *hamiltonian path*, that is, a path from Start to Goal containing exactly one occurrence for other each node. The graph Adleman considered in his paper is given as (a) of Figure 2, where the nodes 0 and 6 are designated as Start and Goal, respectively.

## 2.1. *Molecular algorithm*

In this section, the outline of Adleman's molecular algorithm to solve DHPP is given as follows:

**Input**: A directed graph $G = (V, E)$ with Start 0 and Goal 6.

**Output**: *Yes* (if a hamiltonian path exists) or *No* (otherwise).

**Step 1**: Encode $V$ and $E$ into molecules, in the way suggested by an example shown in (b) of Figure 2.

**Step 2**: Generate all possible paths to construct the *random pool* using volumes of DNA encoded molecules.

**Step 3**: Extract from the pool only hybridized molecules starting with Start and ending with Goal, to form the resulting new pot $T$.

**Step 4**: Extract from $T$ only hybridized molecules containing exactly $|V|$ nodes, to form the resulting new pot $T$.

**Step 5**: Extract from $T$ only hybridized molecules containing every node of $|V|$, to form the resulting final pot $T$.

**Step 6**: Answer *Yes* if there remains any molecule in $T$, or *No* otherwise.

The validity of the algorithm given above is roughly verified in the following manner. First, **Step 1** and **Step 2** guarantee that the structural information of a given graph $G$ is fully encoded into the set of molecules, and the generated random pool contains hybridized molecules encoding a hamiltonian path of $G$ if it exists. Then, a series of **Step 3** through **Step 5** separates only molecules with the properties that they start with Start and end with Goal, and that they contain each node of $V$ exactly once. Hence, if *Yes* is output at **Step 6**, then it means that there exists a hamiltonian path, and otherwise there is no such path.

## 2.2. *Lessons from Adleman's work*

We now abstract the crucial notions and operations in Adleman's experiment and formulate various bio-chemical techniques to design a kind of a "molecular programming language" for molecular computing.

Let $\Sigma = \{\mathsf{A}, \mathsf{C}, \mathsf{G}, \mathsf{T}\}$ be an alphabet and consider a string over $\Sigma$ as a single-stranded DNA sequence. Then, a test tube containing single-stranded DNA molecules can be regarded as a multiset $T$ of strings over $\Sigma$. By $\Sigma^*$, we denote the set of all strings of finite length(including 0) over $\Sigma$. Given $x \in \Sigma^*$, $lg(x)$ denotes the length of $x$.

Then, the following basic operations used in Adleman's experiment are defined in terms of set theoretic formulations:

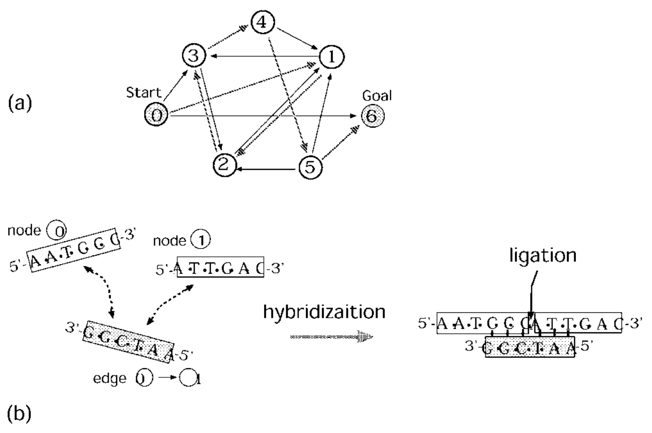


*Figure 2.* (a) Adleman's graph of directed Hamiltonian path problem; (b) encoding nodes and edges into molecules.

1. *amplify*$(T)$: Make two copies of $T$.
2. *merge*$(T_1, T_2)$: Make a union $T = T_1 \cup T_2$.
3. *separate*$(T, w)$:
   Given $T$ and $w \in \Sigma^*$, make $T_{+,w} = \{x \in T \mid x \text{ contains } w\}$
   and $T_{-,w} = T - T_{+,w}$.
4. *L-separate*$(T, \leq n)$:
   Given $T$ and an integer $n \geq 0$, make a set $\{x \in T \mid lg(x) \leq n\}$.
5. *B-separate*$(T, w)$:
   Given $T$ and $w \in \Sigma^*$, make a set $\{x \in T \mid x \text{ starts with } w\}$.
6. *E-separate*$(T, w)$:
   Given $T$ and $w \in \Sigma^*$, make a set $\{x \in T \mid x \text{ ends with } w\}$.
7. *detect*$(T)$: Reply *true* if $T \neq \emptyset$, and *No* otherwise.

We are now in a position to write a *molecular program* for solving an instance $G = (V, E)$ of DHPP. Suppose that $T$ be a random pool that comprises DNA molecules encoding all possible paths. Further, each node $i$ of $V = \{0, 1, \cdots, 6\}$ is encoded as a string $w_i$ of length 6 over $\Sigma$.

```
1. input($T$); (Take $T$ as an input)
2. begin
3.     $T \leftarrow$ B-separate($T, w_0$);
4.     $T \leftarrow$ E-separate($T, w_6$);
5.     $T \leftarrow$ L-separate($T, \leq 42$);
6.     for $i = 1$ to 5 do
7.      begin
8.             $T \leftarrow$ separate($T_{+,w_i}$);
9.      end
10.    detect($T$);
11. end
```

Thus, a molecular algorithmic solution for an instance of DHPP is nicely described using a molecular programming language.

A computing model suggested by Adleman's method can deal with a broader class of problems of $\mathcal{NP}$ including SAT(the satisfiability problem for Boolean formulas), which will be discussed in some details in Section 7.2.2.

## 3. Splicing paradigm

A theory of Splicing systems provides one of the mathematical models for molecular computing, proposed by Tom Head in (Head, 1987). The model of splicing systems[2] was originally introduced to model the biochemical phenomena of DNA recombination and analyze their mathematical (linguistic) properties, then later (after Adleman's groundbreaking work) recognized as a fundamental theory of DNA-based computing.

### 3.1. *Splicing operations*

There are two specific enzymes known to be involving biochemical reactions in vivo. One is a restriction enzyme that recognizes a specific site of double stranded molecules and cuts them at the position, and the other is ligase that pastes two molecular fragments with matching ends to form double stranded molecules.

Using an illustrative example, the notion of *splicing operation* will be defined as a new type of *string rewriting rules* on double stranded DNA sequences with restriction enzymes and ligases, inspired by a phenomenon of DNA recombination in vivo.

For example, a restriction enzyme *Eco*RI can cut a double stranded molecule $x$:

$$\begin{array}{l}5'\text{-GGG}\\3'\text{-CCC}\end{array}\!\boxed{\begin{array}{l}\text{GAATTC}\\\text{CTTAAG}\end{array}}\!\begin{array}{l}\text{GGG}\\\text{CCC}\end{array}\quad\text{into}\quad\begin{array}{l}5'\text{-GGGG}\\3'\text{-CCCCTTAA}\end{array}\quad\text{and}\quad\begin{array}{r}\text{AATTCGGG}\\\text{GCCC.}\end{array}$$

(see Figure 3). Similarly, a double stranded sequence $y$:

$$\begin{array}{l}5'\text{-AAA}\\3'\text{-TTT}\end{array}\!\boxed{\begin{array}{l}\text{GAATTC}\\\text{CTTAAG}\end{array}}\!\begin{array}{l}\text{TTT}\\\text{AAA}\end{array}\quad\text{is cut into}\quad\begin{array}{l}5'\text{-AAAG}\\3'\text{-TTTCTTAA}\end{array}\quad\text{and}\quad\begin{array}{r}\text{AATTCTTT}\\\text{GAAA,}\end{array}$$

where rectangle portion indicates the recognition site of *Eco*RI.

Then, since these two sets of sequences have a common site of sticky end, by means of hybridization of the Watson-Crick complementarity together with ligation, one may newly obtain two sequences $z$ and $w$, that is

$$\begin{array}{l}5'\text{-GGGGAATTCTTT}\\3'\text{-CCCCTTAAGAAA}\end{array}\qquad\text{and}\qquad\begin{array}{l}5'\text{-AAAGAATTCGGG}\\3'\text{-TTTCTTAAGCCC}\end{array}$$

This recombinant behavior of molecules has been formulated in the following way. Note that because of the Watson-Crick complementarity, we

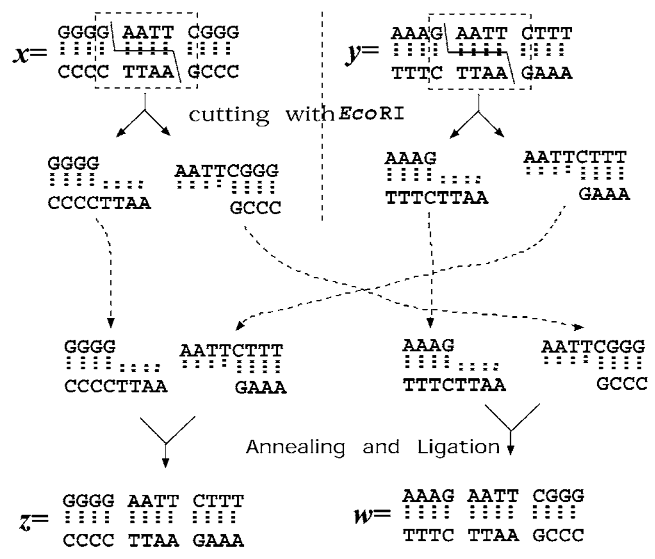


*Figure 3.* Cutting with *Eco*RI and generating new sequences.

have only to consider an operation acting on single stranded sequences, in other words, an operation on strings of a certain alphabet $\Sigma$.[3]

Let $x_1$ and $x_2$ be two strings over $\Sigma$ containing as substrings $\alpha_1\beta_1$ and $\alpha_2\beta_2$, respectively. Then, a *splicing rule* (over $\Sigma$) $r$ is denoted by

$$\alpha_1\#\beta_1\$\alpha_2\#\beta_2,$$

where # and \$ are new symbols not in $\Sigma$ (intuitively, $\alpha_i\beta_i$ $(i = 1, 2)$ corresponds to the recognition site of $x_i$ for each, and meaning that $x_i$ is cut at the position between $\alpha_i$ and $\beta_i$).

Formally, the *splicing* of $x$ and $y$ with $r = \alpha_1\#\beta_1\$\alpha_2\#\beta_2$ is defined as follows:

$$(x, y) \models_r (z, w) \overset{\text{def}}{\Longleftrightarrow} \begin{array}{l} x = x_1\alpha_1\beta_1x_1' \\ y = y_2\alpha_2\beta_2y_2' \end{array} \text{ and } \begin{array}{l} z = x_1\alpha_1\beta_2y_2' \\ w = y_2\alpha_2\beta_1x_1' \end{array}$$

where in the case when $x$ contains more than one occurrence of $\alpha_1\beta_1$, or $y$ contains $\alpha_2\beta_2$ more than once, then the resulting pair $(z, w)$ by the splicing of $x$ and $y$ is nondeterministically produced.

**[Remarks]** The original splicing operation due to Head is formulated as follows:

$$(x, y) \vdash_r z \overset{\text{def}}{\Longleftrightarrow} \begin{array}{l} x = x_1\alpha_1\beta_1x_1' \\ y = y_2\alpha_2\beta_2y_2' \end{array} \text{ and } z = x_1\alpha_1\beta_2y_2'.$$

This is called *1-splicing* and distinguished from the splicing already defined above.

More specifically, a splicing rule was given as a pair of rules $(x_1, u, y_1)$ and $(x_2, u, y_2)$, where $u$ is a common sequence contained in the recognition site of two strings $x$ and $y$, where $x_1$ and $y_1$ ($x_2$ and $y_2$) are two contexts of $u$ in $x$ ($y$). Then, applying these rules to

$$\begin{array}{l} x = x'x_1uy_1y' \\ y = x''x_2uy_2y'' \end{array} \qquad \text{produces} \qquad w = x'x_1uy_2y''.$$

This operation is essentially simulated by $r = x_1\#uy_1\$x_2\#uy_2$, that is, a special case of $\alpha_1\#\beta_1\$\alpha_2\#\beta_2$. In this sense, our definition offers an extended version of the splicing rule introduced by Head.

## 3.2. *Splicing systems and their languages*

Using the 1-splicing operation defined above, in this section, we now introduce a language generative model called *splicing system* (later called H system) (Păun, 1996).

For the first step, an *H scheme* is a pair $\sigma = (V, R)$, where $V$ is a finite alphabet, $R \subseteq V^*\#V^*\$V^*\#V^*$ is a finite set of splicing rules.[4]

Given a $\sigma = (V, R)$ and $L \subseteq V^*$, define

$$\sigma(L) = \{z \in V^* \mid \exists\, x, y \in L, r \in R \text{ such that } (x, y) \vdash_r z\}.$$

Thus, $\sigma(L)$ is the set of strings obtained from $L$ by one application of a 1-splicing operation with $r$ in $R$. When imaging the reaction occurring in a test tube containing double stranded molecules with restriction enzymes and ligases, it is natural to think of the iterative process of splicing reactions. This leads to the following formulation.

Given $\sigma = (V, R)$ and $L \subseteq V^*$, let

$$\sigma^0(L) = L \quad \text{and} \quad \sigma^{i+1}(L) = \sigma^i(L) \cup \sigma(\sigma^i(L)),\ i \geq 0,$$

and define

$$\sigma^*(L) = \bigcup_{i \geq 0} \sigma^i(L).$$

Finally, a triple $\gamma = (V, R, L)$ is called *splicing system* (or *H system*). A language $L'$ is called *splicing language* if $L' = \sigma^*(L)$ for some H system $\gamma = (V, R, L)$.

In general, for two families of languages $\mathcal{FL}_1, \mathcal{FL}_2$, define

$$H(\mathcal{FL}_1, \mathcal{FL}_2) = \{\sigma^*(L) \mid L \in \mathcal{FL}_1,\ \sigma = (V, R),\ R \in \mathcal{FL}_2\}.$$

We will now present some basic results in DNA computing based on splicing operations.

THEOREM 1.
(1) $H(\mathcal{FIN}, \mathcal{FIN}) \subset \mathcal{REG}$.
(2) (Regularity Lemma) $H(\mathcal{REG}, \mathcal{FIN}) = \mathcal{REG}$.

Thus, there is an upper bound for the computing capability of H systems (under 1-splicing operation) which uses a regular set of axioms and a finite set of rules (Culik and Harju, 1991).

Further, later study on this topic shows a stronger result (Pixton, 1995).

THEOREM 2. *Let $\mathcal{FL}$ be a full AFL.*[5] *Then, it holds that $H(\mathcal{FL}, \mathcal{FIN}) \subseteq \mathcal{FL}$.*

## 3.3. *Extended models of splicing systems*

This section discusses several extended models of H systems which have been introduced for the purpose of enhancing the computing capability beyond the *regularity bound*, hopefully of achieving the Turing universal computability, within the framework of allowing finite sets of axioms and of rules.

### 3.3.1. *Extended H systems*

In standard formal language theory, formal grammars deal with two types of symbols, that is, terminals and nonterminals. Similarly, one can consider H systems with this distinction of symbols, to enhance the computing capability of the systems.

An *extended H system* is a quadruple $\gamma = (V, T, A, R)$, where $T$ is a sub-alphabet of $V$ and $A(\subseteq V^*)$ is a finite set of axioms, $\sigma = (V, R)$ is called the *underlying H scheme* of $\gamma$ (Note that when $T = V$, $\gamma$ is nothing but an H system).

A *language generated by an extended H system* $\gamma$ is defined by

$$L(\gamma) = \sigma^*(A) \cap T^*,$$

where $\sigma$ is the underlying H scheme of $\gamma$.

Given two language families $\mathcal{FL}_1$ and $\mathcal{FL}_2$, $EH(\mathcal{FL}_1, \mathcal{FL}_2)$ denotes the family of languages $L(\gamma)$ for $\gamma = (V, T, A, R)$ such that $A \in \mathcal{FL}_1$ and $R \in \mathcal{FL}_2$.

Then, the following results are obtained.

THEOREM 3.
(1). $EH(\mathcal{FIN}, \mathcal{FIN}) = \mathcal{REG}$
(2). $EH(\mathcal{FIN}, \mathcal{REG}) = \mathcal{RE}$

These results are instructive in that as far as extended H systems with finite axioms and finite rules, their language generating power cannot exceed the family of regular languages. On the other hand, in order to characterize the family $\mathcal{RE}$ it suffices to allow extended H systems regular sets of rules, admitted that an infinite set of rules is not realistic from the practical viewpoint.

Now, a question arises here: Is there any way to increase the computing power of extended H systems keeping the condition of finite axioms and rules?

The subsequent section will focus on the efforts to explore the possibility of enhancing extended H systems with *finite* axioms and *finite* rules.

### 3.3.2. *Lesson from Post systems*

A classical rewriting system called *General Regular Post system* (GRP system) is a quadruple $G = (V, \Sigma, P, A)$, where $V$ and $\Sigma(\subseteq V)$ are alphabets, $P$ is a finite set of rules of the form of either $uX \to wX$ or $aX \to Xb$ ($X \notin V$: variable, $u, w \in V^*, a, b \in V$), $A(\subset V^+)$ is a finite set of axioms.

Given strings $\alpha, \beta \in V^*$, define a binary relation $\Longrightarrow$ as follows:

$$\alpha \Longrightarrow \beta \overset{\text{def}}{\Longleftrightarrow} \begin{cases} \exists uX \to wX \in P, \delta \in V^* \; [\alpha = u\delta, \beta = w\delta] \text{ or} \\ \exists aX \to Xb \in P, \delta \in V^* \; [\alpha = a\delta, \beta = \delta b]. \end{cases}$$

Let $\Longrightarrow^*$ be the reflexive and transitive closure of $\Longrightarrow$. Then, a language generated by $G$ is defined as

$$L(G) = \{w \in \Sigma^* \mid \exists u \in A \text{ such that } u \Longrightarrow^* w\}.$$

It is known that the family of languages generated by GRP systems coincides with the family of recursively enumerable languages (Salomaa, 1985).

The secret of the universal computing capability of GRP systems lies in the two types of rules $\alpha X \to \beta X$ (say, type 1 rule) and $aX \to Xb$ (type 2 rule). Interesting is the generative capability between these two types of rules.

THEOREM 4. (Post, 1943) *GRP systems with type 1 rules alone can generate only regular languages, while type 1 rules together with type 2 rules enable GRP systems to generate any recursively enumerable language.*[6]

These results seem to be useful in analysing the computing power of splicing systems. That is, it is rather easy to see that type 1 rules can be simulated by splicing rules. Then, what about simulating type 2 rules by splicing rules? Thus, if this is possible, then one can come to have splicing models within finite axioms and finite rules.

As seen in the subsequent discussion, these observations lead to new types of splicing models with the universal computability.

### 3.3.3. *Circular H systems*

In nature, there exist DNA molecules, such as Bacteria DNA and Plasmid, that form circular structures. This inspires a new type of splicing models of computing where "circular splicing" produces a mixture of linear and circular molecules.

For a (linear) string $x$ over $V$, the circulization of $x$, called *circular string*, is denoted by $\hat{}x$. The set of circular strings over $V$ is denoted by $V\hat{}$.

Given $x, z \in V^*$ and $\hat{}y, \hat{}w \in V\hat{}$, and $r = u_1\#u_2\$u_3\#u_4$, define *circular splicing* by

$$(x, \hat{}y) \models_r (z, \hat{}w) \quad \overset{\text{def}}{\Longleftrightarrow} \quad \begin{cases} \text{for some } x_1, x_2, y_1, y_2 \in V^* \\ x = x_1u_1u_2x_2, \ \hat{}y = \hat{}y_1u_3u_4y_2 \\ z = x_1u_1u_4y_2, \ \hat{}w = \hat{}y_1u_3u_2x_2 \\ \text{where either } y_1 \text{ exclusively or } y_2 \text{ is empty.} \end{cases}$$

Figure 4 illustrates the behavior of circular splicing (circular splicing defined here is called *mixed splicing* in Head et al. (1997)).

*Circular H system* (CH system) is a quadruple $H = (V, \Sigma, A, R)$, where $V$ and $\Sigma(\subseteq V)$ are alphabets, $A(\subseteq V^* \cup V\hat{})$ is a finite set of axioms, and $R(\subseteq V^*\#V^*\$V^*\#V^*)$ is a finite set of rules. Then, we define

$$\delta(A) = \{z \in V^*, \ \hat{}w \in V\hat{} \mid \text{ for some } x, \hat{}y \in A \text{ and } r \in R \\ (x, \hat{}y) \models_r (z, \hat{}w)\}.$$

Further, let

$$\delta^*(A) = \bigcup_{i \geq 0} \delta^i(A) \ \text{ where } \ \begin{array}{l} \delta^0(A) = A \\ \delta^{i+1}(A) = \delta^i(A) \cup \delta(\delta^i(A)) \ (\forall i \geq 0). \end{array}$$

Then, a language $L_\ell(H)$ of linear strings over $\Sigma$ is defined by

$$L_\ell(H) = \delta^*(A) \cap \Sigma^*.$$

We now have the following result. The key idea for the proof is to simulate by using circular operations type 2 rules in Theorem 4.

THEOREM 5. (Yokomori et al., 1997) *For any recursively enumerable language $L$, there effectively exists a circular H system $H$ such that $L = L_\ell(H)$.*

For more discussion on the topics of circular and mixed splicing systems and languages including closure properties of AFLs, refer to Head et al. (1997), Siromoney et al. (1992), Pixton (1995).

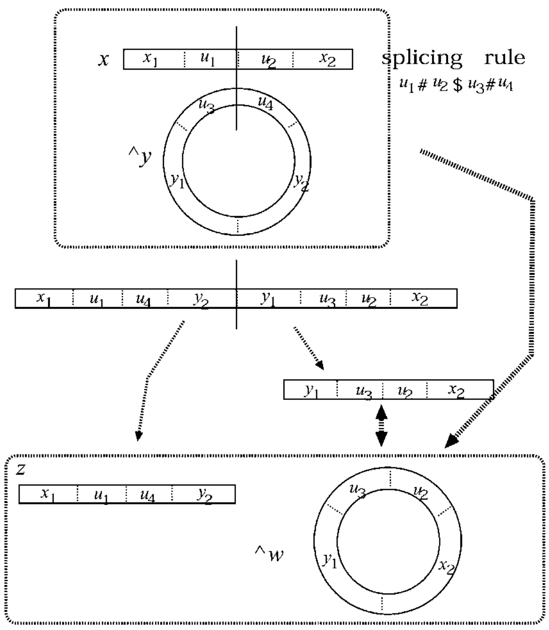


*Figure 4.* Circular splicing.

3.3.4. *Multi-test tube models*

Let us revisit the issue of simulating type 2 rules of GRP systems, where the question was "Is there any other way to carry out a rotation rule $aX \to Xb$ within the splicing scheme?".

One of such ways is possible when one considers the use of more than one test tube as in (Csuhaj-Varju et al., 1996) and (Kobayashi and Sakakibara, 1998) B!

An *elementary formal system* (EFS) is a triple $E = (D, \Sigma, M)$, where $D$ is a finite set of predicate symbols, $\Sigma$ is a finite alphabet, and $M$ is a set of logic formulas. Then, $F$ in $M$ is an *H-form* iff it is of the form:

$$P(x_1 u_1 v_2 y_2) \leftarrow Q(x_1 u_1 u_2 x_2)\ R(y_1 v_1 v_2 y_2) \text{ or } P(w) \leftarrow$$

where $x_1, x_2, y_1, y_2$ are all distinct variables, $u_1, u_2, v_1, v_2$ and $w$ are in $\Sigma^*$. An EFS $E = (D, \Sigma, M)$ is called *H-form EFS* iff every $F$ in $M$ is a H-form. Further, an H-form EFS with a single predicate $P$, that is, an H-form EFS $E = (\{P\}, \Sigma, M)$ is called *simple* H-form EFS.

Given a predicate $P$ and EFS $E = (\{P\}, \Sigma, M)$, define

$$L(E, P) = \{w \in \Sigma^* | P(w) \text{ is provable from } E\}.$$

It is known that a simple H-form EFS is equivalent to an H-system in computing capability.

THEOREM 6. *For any H-system $S = (\Sigma, R, A)$, there effectively exists a simple H-form EFS $E = (\{P\}, \Sigma, M)$ such that $L(E, P) = L(S)$. Conversely, any simple H-form EFS $E = (\{P\}, \Sigma, M)$, one can construct an H-system $S = (\Sigma, R, A)$ such that $L(S) = L(E, P)$.*

Thus, in order to get the computing power beyond H-systems, it is necessary to consider an H-form EFS with more than one predicate symbol, which leads us to an idea of "multi-splicing models" where more than one test tube are available for splicing operations.

Given an $n > 0$, by $\mathcal{HE}_n$ we denote the family of languages defined by H-form EFSs with $n$ predicate symbols. Further, let $\mathcal{HE}_* = \cup_{i\geq 1}\mathcal{HE}_i$.

Then, the followings have been proved:

THEOREM 7.
1. $H(\mathcal{FIN}, \mathcal{FIN}) = \mathcal{HE}_1 \subset \mathcal{HE}_2 \subseteq \mathcal{HE}_3 \subseteq \cdots\cdots \subseteq \mathcal{HE}_{13} = \cdots\cdots = \mathcal{HE}_* = \mathcal{RE}$
2. $\mathcal{HE}_2$ contains non-regular languages.
3. $\mathcal{HE}_3$ contains non-context-free languages.

Thus, 13 test tubes are sufficient to generate any recursively enumerable language in multi-splicing models.

The way how to simulate an H-form EFS $E = (D, R, A)$ by a multi-splicing model is outlined below, where $|D| = 2$, that is, two test tubes $T_1$ and $T_2$ are available (Kobayashi and Sakakibara, 1998). It suffices to show how to simulate $aX \to Xb$ using splicing operations in two test tubes. For a given string $w = ax$, make $w' = BaxE$, where $B$ and $E$ are new symbols of marker. Then, delete a prefix $Ba$ from $w'$ in $T_1$ and transfer $xE$ to $T_2$ that contains $b$. Then, replace $E$ in $xE$ with $b$, resulting in $xb$.

It remains open to solve how many test tubes is minimally necessary to generate any recursively enumerable language. See Csuhaj-Varju et al. (1996) for related discussion.

### 3.3.5. *Other extended models*

As for other extended models based on splicing scheme, there is a nature extension of splicing systems where tree structural molecules are considered to be spliced (Sakakibara and Ferretti, 1997). The language family generated by *tree splicing systems* is shown to be the family of context-free languages.

An *H system with permitting contexts* $(C_1, C_2)$ is an extended model motivated by promotors in vivo where a 2-splicing operation to $(x, y)$ is executed if $x$ and $y$ contain as substrings all elements of $C_1$ and $C_2$, respectively. Conversely, an *H system with forbidding contexts* $(D_1, D_2)$ inspired by inhibitors is considered where a splicing to $(x, y)$ is possible if $x$ and $y$ do *not* contain as substrings any element of $D_1$ and $D_2$, respectively. In both cases two extended H-systems generate any recursively enumerable language (Freund et al., 1995).

More models of extended splicing systems can be found in Păun et al. (1998), while lab experimental work on splicing systems is reported in Laun (1997).

## 4. Self-assembly paradigm

When we look around our daily lives, we sometimes come across various phenomena which are regarded as "self-assembly computing". For example, raindrops closely falling on a leaf autonomously meet, then merge together, gradually growing into a bigger one. This phenomenom, regarded as a naive *addition*, is supported by the property of $H_2O$ and the minimization of potential energy due to surface tension.

### 4.1. *Self-assembly computation principle*

In the theory of molecular computing, a variety of computation models have been proposed and investigated. Among others the models of computation based on *self-assembly* are well recognized to be of great importance in that they can provide one of the most basic frameworks for molecular computing paradigms. In fact, Adleman's groundbreaking experimental work (Adleman, 1994) previously introduced in this paper (solving a small instance of the Hamiltonian Path Problem) is a typical example of molecular computation based on the self-assembly principle. Winfree et al. (Winfree et al., 1999) have proposed their *DNA Block models* and showed with a great success that the self-assembly of a certain type of high dimensional structure of molecules achieves Turing universal computability, that is, generates all recursively enumerable languages. Further investigation along this line by Winfree, Eng and Rozenberg (Winfree et al., 2000) refines on their theory as *DNA tiling models* and shows, e.g., that linear self-assembly models of string tiles can provide alternative characterizations of subfamilies of ET0L languages. These lead to a schema that *computation = self-assembly + structural molecules*.

On the other hand, Yokomori (Yokomori, 1999a) proposes a self-assembly computation model called *YAC* and discusses a new computation schema: *computation = self-assembly + conformational change*, where due to the use of comformational change, the structural complexity of molecules used is much simpler than that of Winfree et al's models. It is shown that *YAC* has the power of Turing universal computability.

Thus, generally one may also claim the following schema that

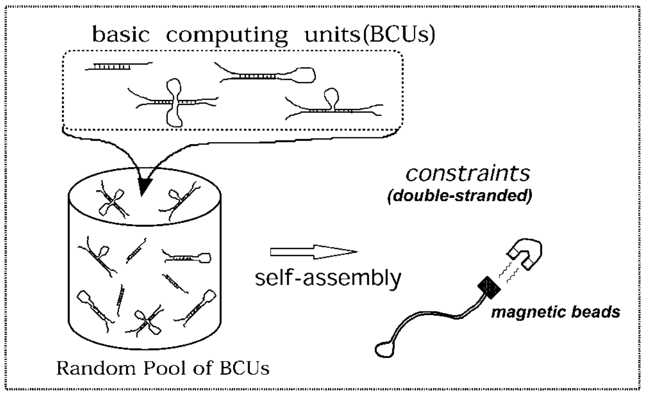


*Figure 5.* Computation schema based on self-assembly.

(Computation) = (Self-Assembly) + (Screening Mechanism)

where "self-assembly" is due to hybridization of either uncoded or coded molecular components, while "screening mechanism" is regulated by either natural or artificial constraint. Related discussion on conformation-driven computation can be found in several references (Conrad, 1985; Conrad, 1992; Saitou and Jakiela, 1996) (note that there are trade-off relations on computational powers between the structural complexity of self-assembly components and the screening mechanism).

When we make a brief revisit to the Adleman's original work in 1994, it turns out that Adleman's algorithm may be formulated into the schema where component structures are sophisticatedly *encoded* into *linear* molecules. From these observations, a general schema of "one pot" self-assembly computation model is outlined as follows:

1. design a finite set of basic units for assembly computation (one may take this finite set as a program),
2. put all those basic units with sufficiently high concentration into one pot, to create a random pool of a single pot (that is, leave the pot for a certain time period, resulting in producing all possible assembly of basic units),
3. (if necessary) perform screening mechanism to extract only necessary (or unnecessary) assembly of basic units,
4. detect whether or not there is an assembly with desired constraints, then answer *yes* if there is, and *no* otherwise.

Figure 5 illustrates the overall outlook of computation schema based on self-assembly.

In the sequel of this section, we will review some of the typical models based on self-assembly computation which include DNA block assembly model, DNA string tile model, YAC model and so forth.

## 4.2. *DNA block models*

Winfree et al. consider the computational powers of a variety of combinatorial self-assembly of the four kinds of DNA complex *units* (see Figure 6):

(a) A duplex molecule: It consists of two strands with contiguous basepairs between the two having optionally a sticky end on either side.
(b) A hairpin molecule: It has a loop on one end and a sticky end on the other.
(c) 3-arm branched molecule: It consists of 3 duplex arms arranged around a center point.
(d) A double crossover (DX) molecule: It consists of two adjacent duplexes with two points of strand exchange.

A question of what can be computed by self-assembly using DNA complex units mentioned above has been considered, where self-assembly starts with a combination of units selected from four kinds of DNA complexes and proceeds at a constant temperature, facilitating only permanent complementary hybridization without any intermolecular reaction and finally produces stable DNA complexes (say, *final* DNA complexes). The computational capability of self-assembly mechanism is analyzed in terms of the complexity of formal languages defined by final DNA complexes that are generated by a combination of DNA complex units selected for self-assembly.

Let $U$ be a subset of four DNA complex units and $L(U)$ be the set of all final DNA complexes obtained by self-assembly using only elements from $U$, where arbitrarily many copies of each unit are allowed to join the assembly in principle. The language defined by $L(U)$ is a set of strings (of linear DNA sequences) over a fixed alphabet $\{\mathsf{A}, \mathsf{C}, \mathsf{G}, \mathsf{T}\}$, obtained from elements of $L(U)$ by denaturation (note that writing and reading DNA sequences involve a certain encoding technique).

The following results are proved.

THEOREM 8. (Winfree et al., 1999)
(i) *The language family of self-assembly obtained by using DNA complex unit of only type* (a) *(i.e., duplexes) are exactly the family of regular languages.*
(ii) *The language family of self-assembly obtained by using DNA complex units of types* (a), (b) *and* (c) *are exactly the family of context-free languages.*
(iii) *The language family of self-assembly obtained by using DNA complex units of types* (a) *and* (d) *are exactly the family of recursively enumerable languages.*

It is easy to see the validity of result (i) above, if one consider the process of linearly growing an assembly of duplexes only. For the result (ii), given a context-free grammar $G$ in Chomsky normal form, an idea is to simulate a derivation tree of $G$ by using DNA complex units of types (a), (b), and (c). Note that a deviation by $A \rightarrow BC$ in $G$ is simulated by a DNA

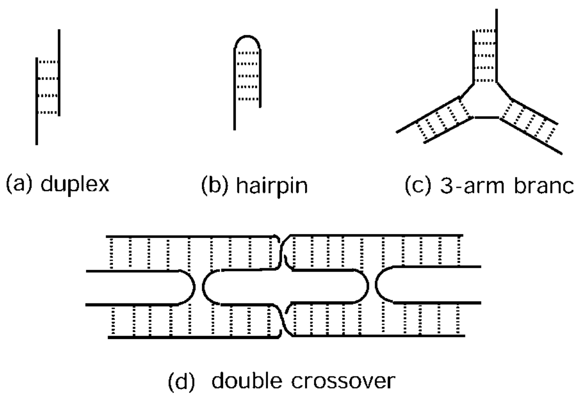


*Figure 6.* DNA complex units for self-assembly: solid lines represent backbone edges; each dotted line represents a pair of reciprocal basepair edges. (a) a duplex with sticky ends. (b) a hairpin with a sticky end. (c) 3-arm branched junction. (d) a double crossover (DX) unit with sticky ends.

complex unit of type (c). The proof for the result (iii) is based on simulating a 1-dimensional cellular automaton (computationally equivalent to a Turing machine) by DNA complex units of type (a) and (d).

Further, it is shown in Eng (1999) that a language family of self-assembly obtained by using DNA complex units of types (a) and (b) are exactly the family of linear (context-free) languages.

## 4.3. *String tile models*

In a previous section, the computational power of self-assembly using DNA complex units can be equal to Turing machines if structurally complicated DNA molecules (such as DX units) are involved. The configuration of computing process in the case, however, requires 2-dimensional space for self-assembly, and therefore, the computing model seems to necessarily be error-prone.

On the other hand, as we have seen above, the computational capability of a linear assembly in terms of duplex units can only generate at most regular languages. This naturally suggests that in order to obtain more capability than regular languages within the framework of a *linear* (or *string*) assembly, one might need more structural molecules than duplex units. In fact, this is formally proved in the following framework.

In order to investigate the computational power of linear self-assembly of DNA tiles, Winfree et al. in Winfree et al. (2000) propose and classify string tiles into five categories whose typical examples are illustrated in Figure 7:

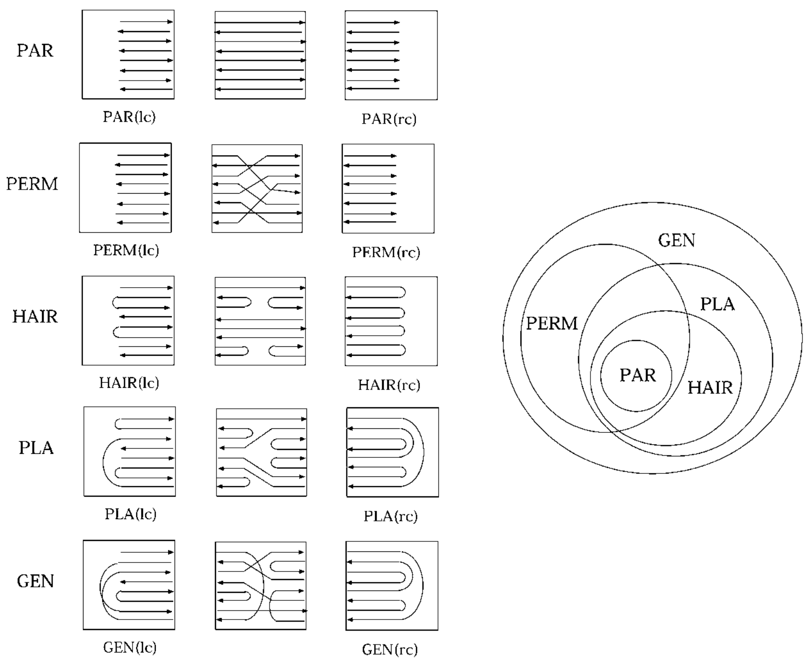


*Figure 7.* (left) Five types of tiles: PAR (parallel tiles), PERM (permutation tiles), HAIR (hairpin tiles), PLA (planar tiles), and GEN (general tiles). For each type, examples of the left-cap tile and the right-cap tile are shown at the left and the right, respectively. (right) Tile class inclusion diagram.

(a) Parallel tiles: each input node connects output node on the opposite side with a *straight arrow*.
(b) Permutation tiles: each input node connects output node on the opposite side with either a straight arrow or *broken arrow*.
(c) Hairpin tiles: each input node either connects output node on the opposite side with a straight arrow or connects an adjacent node on the same side with a *hairpin arrow*.
(d) Planar tiles: any arrow of three types is allowed but all arrows in each tile are depicted as a planar graph.
(e) General tiles: each tile may contain any arrow of three types.

For each type of tiles, three example tiles are depicted in Figure 7: the left one is called the left-cap tile having no sticky ends on its left, while the right one is the right-cap tile having no sticky ends on its right. An arrow inside the tile is a DNA sequence encoding a symbol string.

Each tile has the port on either side, and each port has several nodes for input and output representing 3′-end and 5′-end of DNA molecule, respectively. Thus, each tile has sticky ends and two adjacent tiles can assemble each other. An assembly process of DNA tiles may terminate when the left and right caps are attached to the assembly from the left and right sides, respectively. An assembly $\alpha$ thus finally obtained is called *maximal assembly*. By tracing a cycle-free path $\pi$, created in the maximal assembly $\alpha$, that starts with input node and ends with output node, a sequence (string) of DNAs is obtained which is denoted by *word*$(\pi)$ in $\alpha$. $L(\alpha)$ denotes the set of strings word$(\pi)$ in $\alpha$.

Let $T$ be a finite set of DNA tiles chosen from five types and $\mathcal{A}(T)$ be the set of all maximal assembly obtained from $T$. Then, the set of strings

$$L(T) = \bigcup_{\alpha \in \mathcal{A}(T)} L(\alpha)$$

is defined. Further, the family of languages $L(T)$ generated by a set of DNA tiles $T$ is denoted by $\mathcal{L}(T)$.

It is interesting to observe that every general tile is the composition of two permutation tiles and one hairpin tile, i.e., the following holds:

(general tile) = (permutation tile) + (hairpin tile) + (permutation tile).

As a notation, for a set $T$ of DNA tiles from five types, if $T = T_1 \cup T_2 \cup T_3$, then it means that $T_1$ and $T_3$ are used for left-cap and right-cap tiles, respectively, and $T_2$ for center tiles.

A *scattered n-metalinear grammar* is a scattered context grammar (Rozenberg and Salomaa, 1997) having the axiom strings of the form: $t_1 A_1 t_2 \cdots t_n A_n t_{n+1}$, and production rules of the form

$$\langle A_1, A_2, \cdots, A_n \rangle \rightarrow \langle x_1 B_1 y_1, x_2 B_2 y_2, \cdots, x_n B_n y_n \rangle$$

and

$$\langle A_1, A_2, \cdots, A_n \rangle \rightarrow \langle x_1, x_2, \cdots, x_n \rangle$$

where $A_i$, $B_i$: nonterminals, and $x_i$, $y_i$: terminal strings.

THEOREM 9. (Winfree et al., 2000)
*(i) Let $T = HAIR \cup PAR \cup HAIR$. Then, the family $\mathcal{L}(T)$ is equal to the family of languages generated by scattered metalinear grammars.*
*(ii) The family $\mathcal{L}(HAIR \cup PAR \cup HAIR)$ is equal to the family $\mathcal{L}(GEN \cup PERM \cup GEN)$.*
*(iii) The family $\mathcal{L}(HAIR \cup HAIR \cup HAIR)$ is equal to the family $\mathcal{L}(HAIR \cup PLA \cup HAIR)$.*

Further, a special type of Turing machine $M$ is called *Hennie machine* if $M$ has (1) two-way read/write input tape, and (2) one-way write-only output tape, with the following restriction that (3) for some $k \geq 1$, $M$ visits each cell on the input tape at most $k$ times, and (4) working space on the input tape is bounded by linear of input length.

Let $M$ be a Hennie machine with input alphabet $\Delta$ and output alphabet $\Sigma$. As a transducer, given an input $x$ if $M$ writes $y$ on the output tape in an accepting state, then we denote it by $y = M(x)$. A language generated by $M$ is defined as $L(M) = \{y \in \Sigma^* | y = M(x), x \in \Delta^*\}$.

THEOREM 10. (Winfree et al., 2000) *The family $\mathcal{L}(GEN \cup GEN \cup GEN)$ is equal to the language family generated by Hennie machines. Also, it is equal to the family $\mathcal{L}(HAIR \cup HAIR \cup HAIR)$.*

Finally, it is known that the computational capability of scattered $n$-metalinear grammars is less powerful than that of Hennie machines.

## 4.4 *Other related models*

In this section, we will further review some of the computing models based on self-assembly principle, in particular, focusing on the molecular computing models equivalent to Turing machines.

**YAC models**

Among many characterization results on the family of recursively enumerable languages, the following theorem is a rather unique in that it provides a normal form for phrase-structure grammars that turns out to be very suitable for developing a new molecular computing model. Geffert proved an interesting normal form theorem for phrase-structure grammars which is rephrased here with some modifications:

THEOREM 11. (Geffert, 1991) *Each recursively enumerable language $L$ over $\Sigma$ can be generated by a phrase-structure grammar $G = (\{S, \mathsf{A}, \mathsf{C}, \mathsf{G}, \mathsf{T}\}, \Sigma, \mathsf{R}_1 \cup \mathsf{R}_2 \cup \mathsf{R}_3 \cup \{\mathsf{TA} \to \lambda, \mathsf{CG} \to \lambda\}, \mathsf{S})$, where (1) for each $a \in \Sigma$, $R_1$ contains a rule of the form $S \to z_a Sa$ (for some $z_a \in \{\mathsf{T}, \mathsf{C}\}^*$), (2) $R_2$ comprises rules of the form $S \to uSv$ ($u \in \{\mathsf{T}, \mathsf{C}\}^*$ and $v \in \{\mathsf{A}, \mathsf{G}\}^*$), and (3) $R_3$ consists of a single rule of the form $S \to u$ (for some $u \in \{\mathsf{T}, \mathsf{C}\}^*$).*

A successful derivation process in $G$ can be divided into three phases: At the first phase (of *Generation*), rules in $R_1$ are used. The second phase (of *Extension*) consists of using rules in $R_2$ and ends up with one application of the rule in $R_3$. Finally, the third phase (of *Checking*) uses rules from $\{\mathsf{TA} \to \lambda, \mathsf{CG} \to \lambda\}$, and so here well-formed pairs ($\mathsf{TA}$ or $\mathsf{CG}$) are erased using the two extra cancellation rules only. Further, there is only one occurrence of either $\mathsf{TA}$ or $\mathsf{CG}$ in any sentential form to which cancellation rules can be applied.

Based on this normal form, a computing model YAC has been proposed, in which each rule in $R$ is translated (transformed) into a specific form of a two-dimensional structural molecule. These basic blocks of structural molecules with sticky ends are designed so that they may form a DNA complex linearly growing by self-assembly property, which corresponds to performing Generation and Extension phases. Then, Checking phase is achieved by simply checking if the DNA complex has a potential to form a completely hybridized double-stranded when it was denatured (note that this final phase of the YAC model computation can be regarded as a filtering based on the use of a specific form of languages called hairpin languages which will be discussed in Section 7.3). It is shown in Yokomori (1999a) that any recursively enumerable language can be recognized by the YAC model.

As for molecular computing models using high-dimensional structures, Jonoska et al. propose molecular solutions to two NP-complete problems: the satisfiability (SAT) problem and 3-vertex-colorability (3VC) problem, where $k$-arm branched junction molecules are used to represent the $k$-degree vertices of the graph (Jonoska et al., 1998). For both problems, the proposed algorithms use 3-dimensional graph structures and the algorithm for SAT runs in time proportional to the number variables involved in the given formula, while for 3VC the algorithm requires a constant number of steps regardless of the size of the graph.

Another type of self-assembly models based on DNA complex molecules has been proposed in Sakakibara and Kobayashi (2001), where one may call it "sticker system with DNA complexes" which is a variant of sticker systems originally proposed in Kari el al. (1998) and Păun and Rozenberg (1998). An advantage of the extended sticker systems is that with the help of weak coding, any recursively enumerable language is characterized by the system using only hybridization operations for the complex molecules, which provides a simpler computing mechanism within sticker models of formulation.

## 5. Chemical reaction paradigm

### 5.1 *P-systems – membrane computing*

Păun has initiated the theory of P-systems that are distributed parallel computing models, motivated from the observation that the processes taking place in the complex structure of a living cell can be regarded as computation driven by chemical reaction.

The basic components of P-system are a membrane structure consisting of several membranes embedded in a main (outer most) membrane called *skin* and delimiting *regions* (the space between a membrane and all directly inner membranes (if any inner membrane exists), shown in Figure 8.

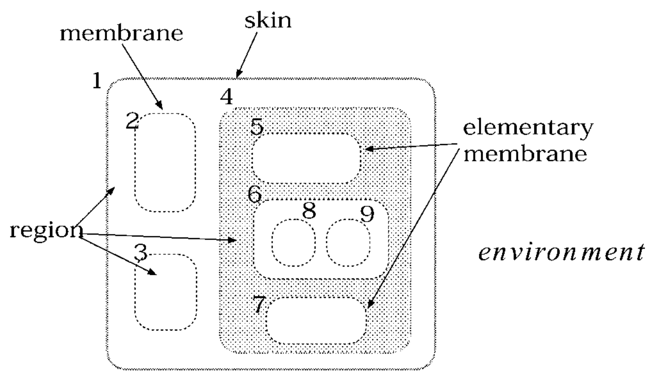


*Figure 8.* A membrane structure: Each membrane is numbered; the membrane 1 is the skin, while membranes 2, 3, 5, 7, 8, 9 are elementary. A region of the membrane 4 is shadowed.

In a region multiset of certain *objects* are placed, and the objects evolve according to given *evolving rules*, which are applied nondeterministically and in a maximally parallel manner. The objects can also be communicated from one region to another. In this manner, the configuration of the system can change from one to another, which is called a *transition* of the system. A sequence of transitions constitutes a *computation* and the result of a computation when the computation halts is defined as the number of objects in a region specified as *output membrane*

Since the P-systems were firstly introduced by Păun, quite a few number of classes of P-systems have been proposed and investigated. Many of them are proved to be computationally complete in that they can compute any Turing computable set of natural numbers. If P-systems have an ability to make membrane division and membrane creation, then NP-complete problems are proved to be solved in linear time. For the details, see Calude and Păun (2001), Păun (1998).

## P-systems with symport and antiport rules

In this section, as one of the most simplest and instructive classes of P-systems recently proposed, P-systems with symport and antiport rules (Păun and Păun, 2002) will be introduced.

Unlike most of the other variants of P-systems, this family of P-systems has a unique feature that *no objects* of a system change during a computation process; they only change their locations (of membrane label numbers) in the system, where objects are various sorts of *chemicals*.

Two mechanisms for objects to communicate each other are considered in this family of P-systems: one is called *symport* which enables objects $a$ and $b$ to pass through a membrane only together in the same direction, represented by a rule of the form $(ab, in)$ or $(ab, out)$ (symport); the other is *antiport* which allows two objects $a$ and $b$ to pass through a membrane in opposite direction, represented by a rule of the form $(a, in; b, out)$ (antiport).[7] Only using these two types of rules, how complicated computation P-system can achieve? This simple question has been settled with a rather surprising answer that P-systems with symport and antiport rules are computationally universal.

Formally, a multiset over a set X is a mapping $M : X \to \mathbf{N} \cup \{\infty\}$ ($\infty$ denotes the infinite multiplicity of an object). A *P-system* (*of degree* $m \geq 1$) *with symport/antiport rules* is a construct

$$\Pi = (V, \mu, M_1, \ldots, M_m, M_e, R_1, \cdots, R_m, i_0), \text{ where}$$

(1) $V$ is the alphabet of *objects*,

(2) $\mu$ is a membrane structure with $m$ membranes (the skip membrane is assumed to be labeled 1),

(3) $M_1, \cdots, M_m$ are the multisets of objects initially present in the regions of the system, and $M_e$ is the multiset of objects present outside the system, *environment* (taking a membrane structure shown in Figure 8, as an example, we have $m = 9$ and the membrane structure is represented by $\mu = [_1\, [_2\, ]_2\, [_3\, ]_3\, [_4\, [_5\, ]_5\, [_6\, [_8\, ]_8\, [_9\, ]_9\, ]_6\, [_7\, ]_7\, ]_4\, ]_1$).

(4) For any $a \in V$, $M_e(a) = 0$ or $M_e(a) = \infty$ and the former means that the object $a$ is absent, the latter means that $a$ appears arbitrarily many times in the environment.

(5) $R_1, \cdots, R_m$ are finite sets of *rules* of the following forms:
- $(a, in)$, $(a, out)$ for $a \in V$ (*uniport rules*)
- $(ab, in)$, $(ab, out)$ for $a, b \in V$ (*symport rules*)
- $(a, out; b, in)$, for $a, b \in V$ (*antiport rules*)

An object $a$ can enter (exit) the region of membrane $i$ if a rule $(a, in)$ ($(a, out)$) is in $R_i$. Objects $a$ and $b$ together can enter (exit) the region of membrane $i$ if a rule $(ab, in)$ ($(ab, out)$) is in $R_i$. Further, an bject $a$ enters and *simultaneously* $b$ exits the region of membrane $i$ if a rule $(a, in; b, out)$ is in $R_i$ (note that if a rule $(ab, in)$ is in $R_1$ and $M_e(a) = M_e(b) = \infty$, then the computation will never halt, because objects $a$ and $b$ are always present at the environment and the rule is active forever, which can feed the system arbitrarily many copies of $a$ and $b$).

(6) $i_0 \in \{1, \cdots, m\}$ is the label number of the output membrane where the computation results are sent at the final stage.

The configuration of the system $\Pi$ is defined by the multisets of objects present in the $m$ regions: By using the rules from $R_1, \cdots, R_m$ *nondeterministically*[8] and in a *maximally parallel manner*,[9] the system changes one configuration to another, which specifies a transition. Thus, a *transition* means a redistribution of objects among regions (and environment), which is maximally executed for the chosen set of rules. A *computation* is a sequence of transitions, and it is successful if it halts, that is, it reaches a configuration where no rule can be applied. The *result* of a successful computation is the number of objects present in the membrane labeled $i_0$ in the halting configuration. A computation which never halts contributes no result.

Finally, the set of numbers computed by $\Pi$ is denoted by $N(\Pi)$, and the family of all sets $N(\Pi)$, $\Pi$ with degree at most $m \geq 1$, is denoted by $NPP_m$. Further, by $\mathcal{NRE}$ we denote the family of recursively enumerable sets of natural numbers (note that if objects present in the output membrane are distinguished for each terminal symbol, then the system can compute a set of vectors of natural numbers, as many other P-systems do).

The main result is now ready to present.

THEOREM 12. *For each $m \geq 5$, we have that $NPP_m = \mathcal{NRE}$.*

The proof constructs a P-system $\Pi$ with five membranes in order to simulate a *matrix grammar $G$ with appearance checking* which is known to be equivalent to a Turing machine. Since in order to compute a set of natural numbers $G$ has the unique terminal symbol $a$, $\Pi$ also has $a$ in the alphabet $V$ and it holds that $N(\Pi) = \{n \in \mathbf{N} | a^n \in L(G)\}$.

The number of membranes could be reduced to two if more complicated rules, such as $(ab, out; cd, in)$, are allowed to use. For a rule of the form $(u, out; v, in)$ with strings $u, v$, a pair of numbers $(lg(u), lg(v))$ is called *radius* of the rule (for example, in the system constructed in the proof for Theorem 12, the radius $(r, s)$ of any rule used satisfies that $r + s$ is bounded by 2). By $NPP_m(r, s)$ we denote the family of sets of natural numbers computed by P-systems with at most $m (\geq 1)$ membranes, using rules of radius componentwise at most $(r, s)$. Then, we have:

THEOREM 13. *For each $m \geq 2$ and $(r, s)$ componentwise larger than or equal (2,2), we have that $NPP_m(r, s) = \mathcal{NRE}$.*

Thus, at the sacrifice of a little increase of radius of rules, two membranes suffice for the P-system to be computationally universal. This kind of tradeoff between the number of membranes and the radius size of symport/antiport rules seems to be worth further investigating.

Finally, an interesting question about the computational power of P-systems with symport/antiport rules is "What about using only symport rules?", which remains open to solve.

### 5.2. *More chemical computing models*

The chemical abstract machine (*CHAM*) was proposed in Berry and Boudol (1992) to formally represent an operational semantics of Process Calculi. A distinguished feature of CHAM uses a membrane to achieve local reactions in a compartment. CHAM also has means for communicating with the outside surrounding the compartment. The reaction (computation) rules are classified into several categories: one of them is the general laws which comprise chemical and membrane laws given in an "inference rule" style, acting on multisets. Using multiset notions, concurrent computation and its semantics are readily formulated, while it has yielded a number of interesting extensions and applications.

As another type of membrane computing models, Suzuki et al. have introduced *Abstract Rewriting Systems on Multisets (ARSM)* that is an abstract chemical reaction system in which floating molecules in a solution can interact each other under the rules prescribed. More specifically, a chemical solution is a finite multiset of objects represented by symbols from a given alphabet, where objects are molecules and reaction rules applied to objects are given by a set of rewriting rules (Suzuki, 2001). Unlike the others in this domain of abstract chemical reaction systems, their works mainly stress on analysis of system behaviors in terms of computer simulation from the artificial life research point of view.

Finally, a reference source (Calude et al., 2001) will help the reader to know the fact that there are quite a number of interesting subjects of multisets processing in mathematics and computer science.

## 6. Aqueous paradigm

Aqueous computing paradigm has several advantages over the others models of molecular computing paradigms. First, aqueous computing needs not constructing an initial random pool. Second, very simple coding design is sufficient to formulate the problem into molecules. Third, unlike conventional computing devices, DNA molecules in aqueous computing need no wiring to connect between circuit devices . . . *floating device*. Further, DNA molecules are suitable for realizing memory device in solution water . . . *molecular memory*. All these aspects of DNA molecules facilitates the implementation of molecular computing devices with DNAs in aqueous computing paradigm (Head, 1999).

Aqueous computing starts with making a solution pot containing many copies of a certain DNA molecule. Each molecule plays a role of DNA memory and the whole pot works as a parallel computer. The advantages of molecular solution in water are the following.

1. Each memory molecule is individually accessible and can take a random location.
2. The solution pot can be duplicated arbitrarily many times to make its homogeneous copy, provided that the pot has a mass of copies for each molecule.
3. Memory molecules in separated pots can be easily unified into one.

A memory molecule consists of many bits and is simple enough in that each address bit only needs the once-writing property.

**Aqueous algorithm**

Taking a small instance of the maximal independent subset problem as an example, we show how to solve a combinatorial hard problem by aqueous algorithm. The problem takes as input an undirected graph $G = (V, E)$ and produces as output a maximal subset $V'(\subseteq V)$ such that for each $x, y \in V'$ $(x, y)$ is *not* in $E$ (note that the maximal independent subset problem is known to be NP-complete in the class $\mathcal{NP}$).

Consider an undirected graph $G = (V, E)$, where $V = \{a, b, c, d, e, f\}$ and $E = \{(a, b), (b, c), (c, d), (d, e)\}$. An answer of this instance is a subset $\{a, c, e, f\}$ which is the unique answer for this specific instance.

Assume an alphabetical order on $V$. A subset of vertices of $V$ is represented by a 0-1 bits of 6 length: $a_1a_2 \cdots a_6$, where $a_i = 1$ (if $a_i$ is in $V$) and $a_i = 0$ (otherwise). For example, a subset $V' = \{a, c, e, f\}$ has a bit representation 101011. In general, for a subset $S \subseteq \{0, 1\}^6$, let $P_S$ be the pot comprising all elements of $S$.

Aqueous algorithm starts with constructing the initial solution pot $P$ consisting of many copies of (molecules representing) 111111, that is, $P = P_{\{111111\}}$ (see Figure 9).

Note that the final pot produced by the algorithm must satisfy the constraints of *no pair of vertices* from the subset $V'$ belongs to $E$. For any edge $(x, y)$ in $E$, let $P_S$ be a pot containing molecules whose representing vertex sets satisfy the constraint of *no edge between $x$ and $y$*. Then, $S$ of $P_S$ must consist of all strings of 6-bits with the bit 0 at the position of either $x$ *exclusively or* $y$.

Thus, the algorithm proceeds as follows. Divide $P_{\{111111\}}$ into two pots and apply them the operations $\mathbf{set}_0(a)$ and $\mathbf{set}_0(b)$ to obtain $P_{\{011111\}}$ and $P_{\{101111\}}$, respectively, where for $x \in V$, an operation $\mathbf{set}_0(x)$ sets "0" to the bit for $x$. Then, merge those two pots into one, resulting in that we now have a pot $P_{\{011111,\ 101111\}}$.

The above procedure is iterated to satisfy other "edge constraints", that is, constraints for edges $(b, c)$, $(c, d)$ and $(d, e)$ of $E$ (see Figure 10). After all these processing, we come to obtain the final pot which satisfies the edge constraints for all edges in $E$.

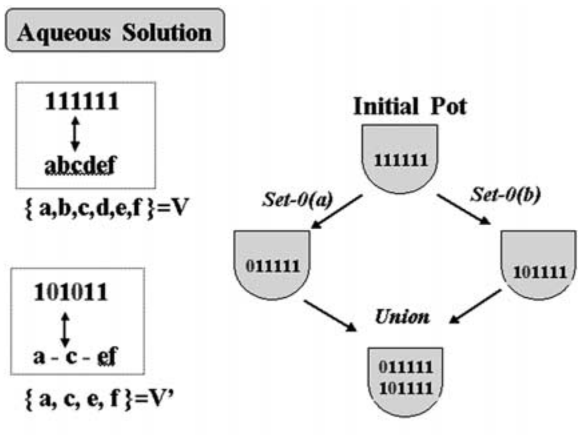


*Figure 9.* Computing process of aqueous algorithm: Part (1).

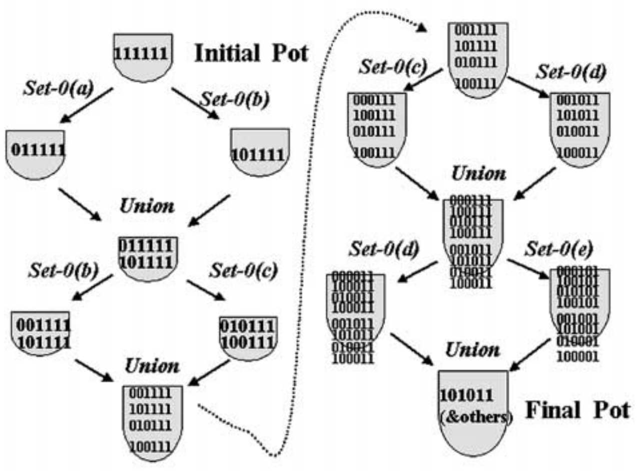


*Figure 10.* Computing process of aqueous algorithm: Part (2).

Now, the rest the algorithm must do is to examine the molecules containing the maximum number of "1" in the bit representation.

There are several implementation methods studied. First, Leiden research group led by Rozenberg together with Head proposes to use a plasmid DNA (a circular double stranded DNA molecule) for memory device and restriction enzymes for writing data in it (Head et al., 2000) B! Second, Binghamton University group led by Head also uses a plasmid where the recognition site of enzymes itself provides a memory bit (Head et al., 1999). Further, a new method by TIT group led by Yamamura proposes to employ Peptide Nucleic Acid(PNA), an artificial analogue of DNA, for writing bits (Yamamura et al., 2001).

## 7. Molecular machines – theory and implementation

### 7.1. *Molecular finite automata*

A finite automaton (FA) is one of the most popular concepts in theoretical computer science, and hence, it is of great use to establish a simple implementation method for realizing an FA in the formulation of molecular computing. In particular, it is strongly encouraged to consider the problem of how simply one can construct a molecular FA using DNAs in the framework of *self-assembly computation*.

Formally, a nondeterministic FA (NFA) is a construct $M = (Q, \Sigma, \delta, p_0, F)$, where $Q$ is a set of states, $\Sigma$ is the input alphabet, $p_0 (\in Q)$ is the initial state, $F (\subseteq Q)$ is a set of final states, and $\delta$ is defined as a function from $Q \times \Sigma$ to $\mathcal{P}(Q)$ (the power set of $Q$) (if $\delta$ is restricted as a function from $Q \times \Sigma$ to $Q$, then $M$ is said to be *deterministic* and is denoted by DFA).

#### 7.1.1. *DNA implementation of NFAs*

The first method introduced here which has been proposed by Gao et al. in Gao et al. (1999) consists of three basic phases: (1) *encoding* the transition rules of a given NFA with an input string onto DNA molecules, (2) *hybridization* and *ligation* which carry out simulating the state transition, and (3) *extracting* the results to decide the acceptance or rejection of the input string.

As an example, consider an NFA $M$ with the initial state 0 and two final states (0:2) and (0:3) given as a transition graph in (a) of Figure 11. For the phase (1), the set of 6 states is encoded by DNA sequences of length 4 as follows:

| | | | |
|---|---|---|---|
| state 0 | atat | state (0:3) | ttat |
| state (1:3) | gctg | state (2:3) | ctca |
| state (0:2) | tatt | state (1:2) | gtcg |

The start molecule is constructed as a double stranded molecule with a sticky end shown in (b) of Figure 11. Each transition rule $p \rightarrow^a q$ (where $p, q$ are states, and $a$ is in $\{0, 1\}$) is encoded as a molecule with a structure so-called "bulge loop". For example, two transition rules: $0 \rightarrow^0 (0:3)$ and $0 \rightarrow^0 (0:2)$ are encoded as (c) of Figure 11, while two transition rules: $0 \rightarrow^1 (1:3)$ and $0 \rightarrow^1 (1:2)$ are encoded as (d) of Figure 11. All of the other transition rules have to be encoded in the same principle on structured molecules with bulge loops (the idea behind the design principle of molecular encoding will be explained below).

A computation in $M$ is simulated in three steps as follows:

(*Step 1*): At the beginning of the reaction, we have the start molecule having the `atatCTTAA` overhang. Suppose that input 0 will be processed. Then, there are two possibilities (tow molecules) that have a `tata` hangover

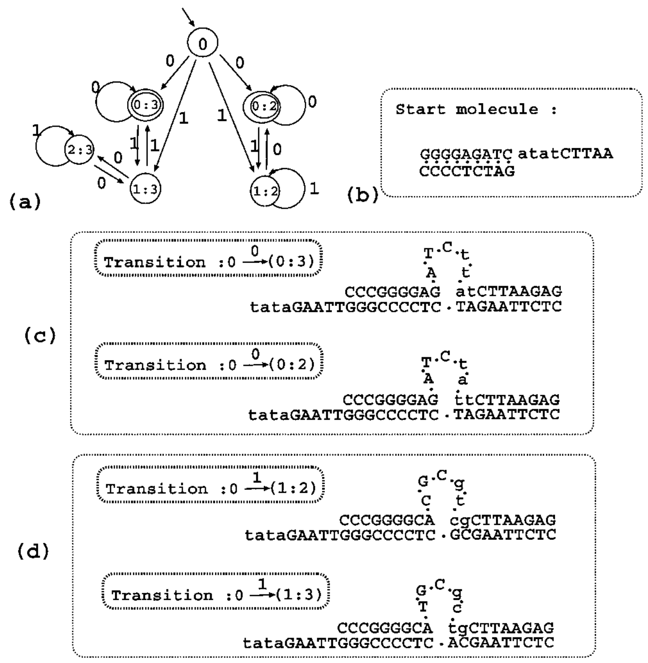


*Figure 11.* NFA for divisibility by 2 or 3: Coding transition rules.

which matches `atat` in the hangover of the current state (of molecule). After hybridization and ligation, the remaining unreacted molecules are all washed away and collected. At this moment, the new possible states (0:3) and (0:2) from 0 with input 0 are covered by the loop part and do not show up yet (the molecule potentially representing the state (0:3) is illustrated as (1) in Figure 12).

(*Step 2*): Add a restriction enzyme *Sma*I which recognizes $\begin{smallmatrix}\texttt{CCCGGG}\\ \texttt{GGGCCC}\end{smallmatrix}$ and cut at the middle betwee `C` and `G`, providing two parts one of which contains bulge loop (see (2) in Figure 12).

(*Step 3*): Add a restriction enzyme *Eco*RI which recognizes $\begin{smallmatrix}\texttt{CTTAAG}\\ \texttt{GAATTC}\end{smallmatrix}$ on the portion with bulge loop and cut it as shown in (3) of Figure 12, providing a molecule representing a new state (0:3) and ready for the next new transition.

The “bulge loop” structure functions to prevent the new state sequence in it from the wrong blunt end reaction during ligation. The *Eco*RI recognition site is used for breaking the bulge loop to expose the next state (i.e., (0:3) or (0:2) in this example) for the subsequent reaction. That is, Steps (1)–(3) are repeated for as many input symbols as necessary. In this manner, it is seen that the nondeterministic transitions in $M$ can be simulated by a sequence of chemical reactions with a certain chemical control.

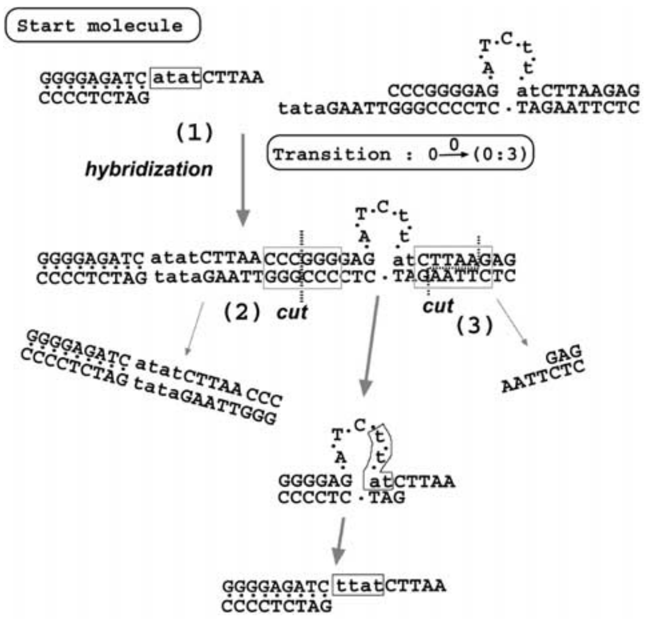


*Figure 12.* Simulating transition with input 0 from the state 0.

This implementation method has several advantages: (1) The size of molecules representing "state" remains unchanged, no matter how long the computation process can be. (2) The reactions proceed in a cyclic way that facilitates its automated implementation. (3) The manner of designing molecules representing states and transitions used in this section can be standardized in that sense that once an input alphabet and the number of states are given, a library of molecules that represent all possible transitions with inputs could be created, independent of a machine specificity.

Finally, it would be interesting and instructive to compare this work with an implementation method more recently proposed in Benenson et al. (2001) where a different type of restriction enzymes and more sophisticated encoding techniques are used, which will be introduced later.

### 7.1.2. *Length-only approach for graph problems*

A new molecular implementation method based on *length-only encoding* is proposed, which leads to a very much simple molecular implementation techniques to solve graph problems. Here, for one example, an effective molecular implementation method for Nondeterministic Finite Automaton, based on *one pot self-assembly* computing, is demonstrated.

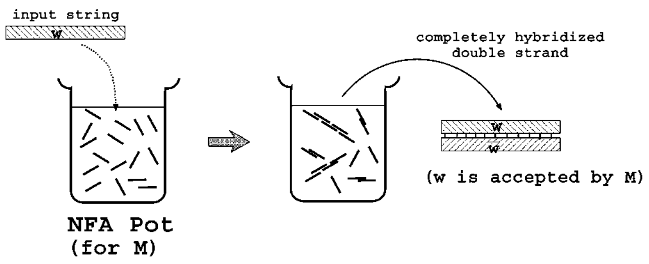


*Figure 13.* Finite automaton pot.

**FA pot – simplest implementation of NFA**

In the sequel, we focus on λ-free regular languages, that is, regular languages not containing the empty string λ. Therefore, in any NFA $M$ we consider here, the initial state $p_0$ is not a final state in $F$ of $M$. The following result is easily shown, but is of crucially importance to attain our ends here. Given any DFA, there effectively exists an equivalent NFA $M$ such that (1) $M$ has the unique final state $q_f$, and (2) there is neither a transition into the initial state $q_0$ nor a transition from the final state $q_f$.

Thus, in what follows *we may assume that a finite automaton M is an NFA satisfying the properties (1) and (2) in the above.*

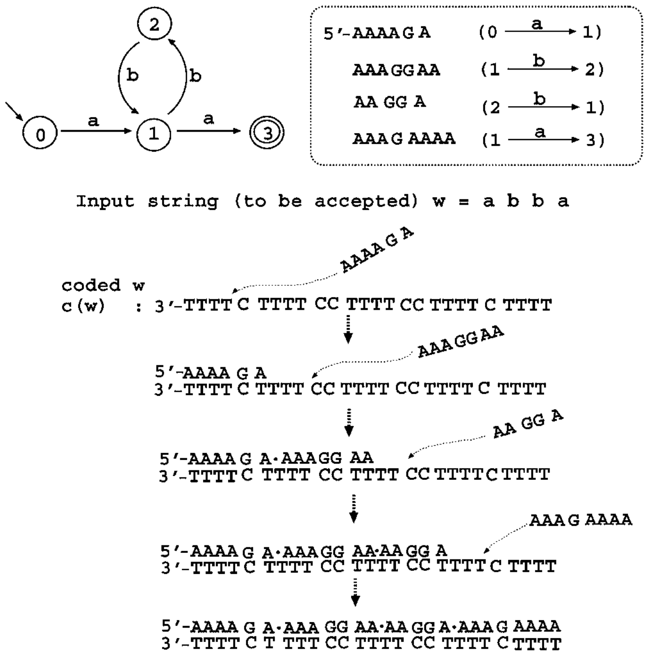
Now, a simple molecular implementation for NFA is presented which one may call *NFA Pot* and its computing schema is illustrated in Figure 13. NFA Pot has the following advantages:

1. In order to encode with DNAs each state transition of $M$, no essential design technique of encoding is required. Only the *length* of each DNA sequence involved is important.
2. Self-assembly due to hybridization of complementary DNAs is the only mechanism of carring out the computation process.
3. An input $w$ is accepted by $M$ if and only if a completely hybridized double strand $[c(w)/\overline{c(w)}]$ is detected from the pot, where $c(w)$ is a DNA sequence representing $w$ and the "overline" version denotes its complementary sequence.

Without loss of generality, we may assume that the state set $Q$ of $M$ is represented as $\{0, 1, 2, \cdots, m\}$ (for some $m \geq 1$) and in particular, "0" and "$m$" denote the initial and the final states, respectively.

Using an example, we show that given an NFA $M$ how one can implement an NFA Pot for $M$.

Let us consider an NFA $M$ (in fact, $M$ is a DFA) in Figure 14, where an input string $w = abba$ is a string to be accepted by $M = (\{0, 1, 2, 3\}, \{a, b\}, \delta, 0, \{3\})$, where $\delta(0, a) = 1$, $\delta(1, b) = 2$, $\delta(2, b) = 1$, $\delta(1, a) = 3$ (note that $M$ satisfies the properties (1) and (2) in the assumption previously mentioned and that the state 3 is the unique final state).



0
1
2
3
a
b
b
a
5'-AAAAG A (0 —a→ 1)
AAAGGAA (1 —b→ 2)
AA GG A (2 —b→ 1)
AAAG AAAA (1 —a→ 3)
Input string (to be accepted) w = a b b a
coded w
c(w) : 3'-TTTTC TTTT CC TTTT CC TTTT C TTTT
AAAAGA
5'-AAAA G A
3'-TTTT C TTTT CC TTTT CC TTTT C TTTT
AAAGGAA
5'-AAAAG A·AAAGG AA
3'-TTTTC TTTT CC TTTT CC TTTTC TTTT
AA GG A
5'-AAAA G A·AAAGG AA·AA GG A
3'-TTTTC TTTT CC TTTT CC TTTTC TTTT
AAAG AAAA
5'-AAAA G A·AAA GG AA·AA GG A·AAAG AAAA
3'-TTTT C T TTT CC TTTT CC TTTT C TTTT

*Figure 14.* FA and its DNA implementation.

In order to encode each symbol $a_k$ from $\Sigma = \{a_1, \cdots, a_n\}$, we associate it with an oligo $\mathsf{GGG}\cdots\mathsf{G}$ of length $k$. Further, each transition $\delta(i, a_k) = j$ is encoded as follows:

$$5'\text{-}\overbrace{\mathsf{AAA}\cdots\mathsf{A}}^{m-i}\overbrace{\mathsf{GGG}\cdots\mathsf{G}}^{k}\overbrace{\mathsf{AAA}\cdots\mathsf{A}}^{j}\text{-}3'.$$

The idea is the following. Two consecutive valid transitions $\delta(i, a_k) = j$ and $\delta(j, a_{k'}) = j'$ are implemented by concatenating two corresponding encoded molecules, that is,

$$5'\text{-}\overbrace{\mathsf{AAA}\cdots\mathsf{A}}^{m-i}\overbrace{\mathsf{GGG}\cdots\mathsf{G}}^{k}\overbrace{\mathsf{AAA}\cdots\mathsf{A}}^{j}\text{-}3',$$

and

$$5'\text{-}\overbrace{\mathsf{AAA}\cdots\mathsf{A}}^{m-j}\overbrace{\mathsf{GGG}\cdots\mathsf{G}}^{k'}\overbrace{\mathsf{AAA}\cdots\mathsf{A}}^{j'}\text{-}3'$$

together make

$$5'\text{-}\overbrace{\mathsf{AAA}\cdots\mathsf{A}}^{m-i}\overbrace{\mathsf{GGG}\cdots\mathsf{G}}^{k}\overbrace{\mathsf{AAA}\cdots\mathsf{A}}^{m}\overbrace{\mathsf{GGG}\cdots\mathsf{G}}^{k'}\overbrace{\mathsf{AAA}\cdots\mathsf{A}}^{j'}\text{-}3'.$$

Thus, an oligo $\overbrace{\mathsf{AAA}\cdots\mathsf{A}}^{m}$ plays a role of "joint" between two transitions and it guarantees for the two to be valid in $M$.

In order to get the result of the recognition task of an input $w$, one application of "**Detect**" operation is used to the pot, that is, the input $w$ is accepted by $M$ iff the operation *detects* a completely hybridized double strand $[c(w)/\overline{c(w)}]$, where $c(w)$ is a DNA sequence encoding $w$ and the "overline" version denotes its complementary sequence. Specifically, for an input string $w = a_{i_1} \cdots a_{i_n}$, $c(w)$ is encoded as:

$$3'\text{-}\overbrace{\mathrm{TTT}\cdots\mathrm{T}}^{m}\overbrace{\mathrm{CCC}\cdots\mathrm{C}}^{i_1}\overbrace{\mathrm{TTT}\cdots\mathrm{T}}^{m}\overbrace{\mathrm{CCC}\cdots\mathrm{C}}^{i_2}\ \cdots\ \overbrace{\mathrm{TTT}\cdots\mathrm{T}}^{m}\overbrace{\mathrm{CCC}\cdots\mathrm{C}}^{i_n}\ \overbrace{\mathrm{TTT}\cdots\mathrm{T}}^{m}\text{-}5'.$$

Figure 14 also illustrates the self-assembly process of computing $w = abba$ in the NFA Pot for $M$ (although $M$ is actually a DFA).

We in turn outline a method for molecular implementation to solve the Directed Hamiltonian Path Problem (DHPP) which is simpler than any others ever proposed. Since this implementation is also based on the "self-assembly in a one-pot computation" paradigm, we may call it *DHPP Pot*.

The DHPP Pot we will introduce has several advantages that are common with and similar to the implementation of NFA Pot. That is,

1. in order to encode each directed edge of a problem instance of DHPP, no essential design technique of encoding is required. Only the *length* of each DNA sequence involved is important.
2. Self-assembly due to hybridization of complementary DNAs is the only mechanism of carring out the computation process.
3. An instance of DHPP has a solution path $p$ if and only if a completely hybridized double strand $[c(p)/\overline{c(p)}]$ of a certain fixed length is detected from the pot, where $c(p)$ is a DNA sequence representing $p$ and the "overline" version denotes its complementary sequence.

As seen above, the implementation methods we have discussed have a common distinguished feature in molecular encoding, that is, *no essential design technique is necessary* in that only the length of DNA sequences involved in the encoding is important.

It should be remarked that the aspect of this non-coding implementation could apply to a large class of problems that are formulated in terms of graph structures (it would be easy to list up a numerous number of problems associated with graphs).

One can also propose a general schema for solving graph problems in the framework of self-assembly molecular computing paradigm where the *length-only-encoding* technique may be readily applied. In fact, some of the graph-related optimization problems such as the Traveling Salesman Problems can be handled in the schema mentioned above.

In the framework of self-assembly molecular computing, it is of great use to explore the computing capability of the self-assembly schema with "double occurrence checking (doc)" as a screening mechanism, because many of the important NP-complete problems can be formulated into this

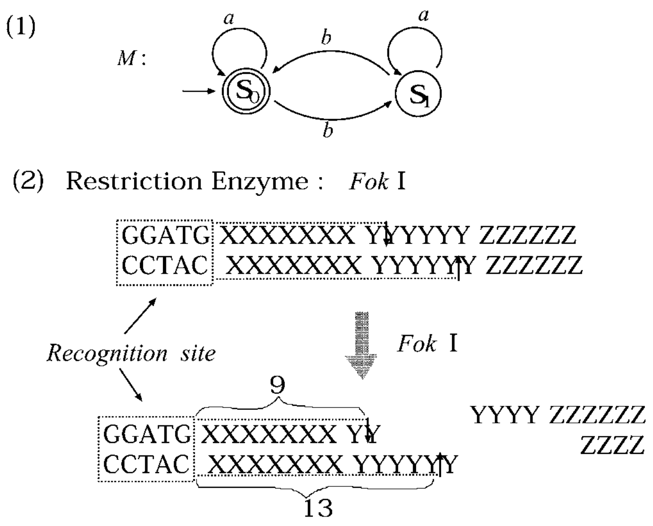


*Figure 15.* (1). DFA $M$ with states $s_0$, $s_1$ and input symbols $a$, $b$. The initial state is $s_0$ which is the unique final state as well. $M$ accepts the set of strings containing even number of $b$'s. (2). The function of a restriction enzyme *Fok*I to work on molecules.

computing schema. We have shown by constructing DHPP Pot that at the sacrifice of *non-linear*(*exponential*) length increase of strands, "doc" function can be replaced with length-only encoding technique. It is, however, strongly encouraged to develop more practical methods for dealing with doc function.

### 7.1.3. *Another DNA implementation – wet FAs*

An interesting wet lab experimental work has been recently reported from Israel's group led by Shapiro, where an autonomous computing machine that comprises DNAs with DNA manipulating enzymes and behaves as a finite automaton was designed and implemented (Benenson et al., 2001). The hardware of the automaton consists of a restriction nuclease and ligase, while the software and input are encoded by double stranded DNAs.

#### Encoding DFA onto DNA

Using an example DFA $M$ given in (1) of Figure 15, we will demonstrate how to encode state transition rules of $M$ onto DNA molecules.

Before going into the details, however, it is important to observe how a restriction enzyme *Fok*I works on double stranded molecules to recognize the site and to cut them into two portions, which is illustrated in (2) of Figure 15. A restriction enzyme *Fok*I has its recognition site $\begin{smallmatrix}\text{GGATG}\\\text{CCTAC}\end{smallmatrix}$ and cut out a double stranded DNA molecule as shown in (2) of Figure 15. That is, splicing by *Fok*I takes place at the location of 9 DNAs right away from the recognition site on

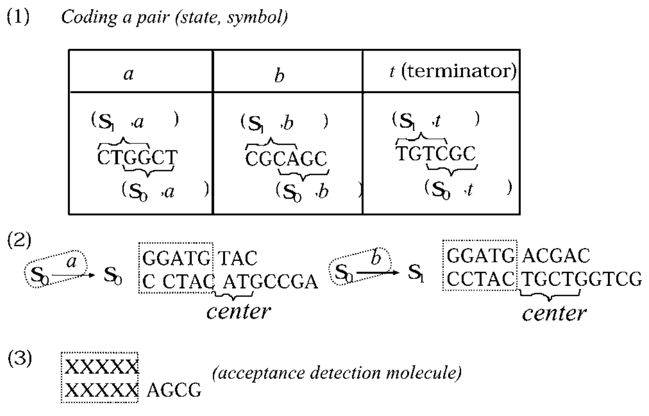


*Figure 16.* States and symbols encoding.

the upper strand and of 13 DNAs right away from the recognition site on the lower strand.

A transition molecule detects the current state and symbol and determines the next state. It consists of (1) a pair (state, symbol)-detector, (2) *Fok*I recognition site and (3) spacer, all these three together determines the location of the *Fok*I cleavage site inside the next symbol encoding, which in turn defines a next state: For example, 3-base pair spacers maintain the current state, and 5-base pair spacers transfer $s_0$ to $s_1$, etc. Also, a special molecule for detecting the "acceptance" of an input is prepared as shown in (3) of Figure 16 (note that *the details of the actual design of molecules for implementing the automaton M is not described here*, and rather only the principle of encoding molecules and computation will be outlined).

Taking this fact into account, encoding *transition molecules* for transition rules of DFA is designed as follows. First, there are two points to be mentioned in designing transition molecules: One is that neither a state nor symbol alone is encoded but a pair (state, symbol) is taken into consideration to be encoded, and the other is that a sticky end of 6-base length encoding (state, symbol) is sophisticatedly designed such that the length 4 prefix of the the sticky end represents $(s_1, x)$, while the length 4 suffix of the same sticky end represents $(s_0, x)$, where $x$ is in $\{a, b, t\}$, and $t$ is the special symbol, not in the input alphabet, for the endmarker (see (1) of Figure 16). Reflecting this design philosophy, structure of transition molecules are created as shown in (2) of Figure 16, where molecules for two transition rules: $s_0 \rightarrow^a s_0$ and $s_0 \rightarrow^b s_1$ are selectively described. The transition molecule $m_{s_0,a}$ for $s_0 \rightarrow^a s_0$ consists of the recognition site of *Fok*I, the center base-pairs of length 3 and the sticky end `CCGA`, because its complementary sequence `GGCT` is designed to represent the pair $(s_0, a)$ as seen from the table. A transition

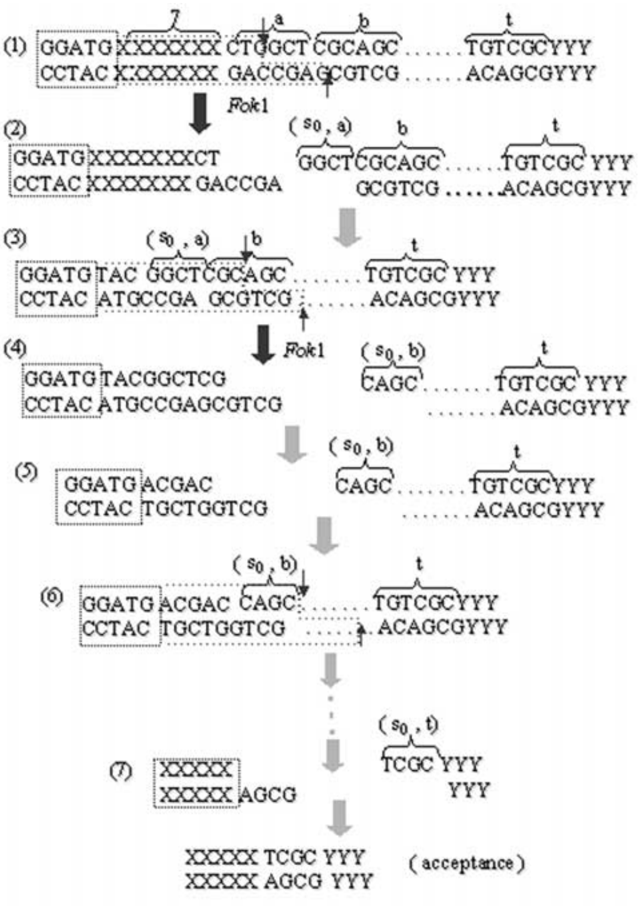


*Figure 17.* Computing process of an input $w = ab \cdots (t)$.

$s_0 \rightarrow^b s_1$ is encoded in a similar principle onto the transition molecule $m_{s_0,b}$ shown in (2) of Figure 16, where a noteworthy difference from $m_{s_0,a}$ is that it has the center base-pairs of length 5. In this principle of designing transition molecules, a transition molecule has the center base-pairs of length 5 if and only if the transition changes its state (from $s_0$ to $s_1$ or vice verse). This feature of distinct length of the center base-pairs plays a crucially important role.

**Computing process**

The molecular automaton processes the input as shown in Figure 17, where input $w$ is assumed to be of the form $ab \cdots (t)$, where $t$ is the endmarker. First, the input $w$ is now encoded into the form of molecule shown in (1) of Figure 17, where seven $X$ base-pairs represent a "spacer" consisting of arbitrary bases.

The input molecule is cleaved by *Fok*I, exposing a 4 base sticky end that encodes the pair of the initial state and the first input symbol, i.e., $(s_0, a)$ in the running example. The computation process is performed in a cyclic manner: In each cycle, the sticky end of an applicable transition molecule ligates to the sticky end of the (remaining) input molecule (obtained immediately after cleaved by the last transition), detecting the current state and the input symbol ((2) of Figure 17) (note that it is assumed at the beginning of computation that all of the possible transition molecules are contained in a solution pot together with the initial molecule and the acceptance detect molecules).

At this moment, since the transition molecule for $(s_0, a)$, shown in (2) of Figure 16, can get access to the remaining input molecule on the right-hand side in (2) of Figure 17, producing a hybridized molecule shown in (3) of Figure 17. Therefore, the resulting molecule is again cleaved by *Fok*I, exposing a new 4 base sticky end ((4) of Figure 17). The design of the transition molecule ensures that the 6 base-pair long encodings of the input symbols $a$ and $b$ are cleaved by *Fok*I at only two different "frames" as we have seen above, the prefix frame of 4 base-pair length for $s_1$ and the suffix frame of 4 base-pair length for $s_0$. Thus, the next restriction site and the next state are exactly determined by the current state and the length of the center base-pairs in an applicable transition molecule. In our example, since the sticky end of the remaining input molecule can match that of the transition molecule $m_{s_0,b}$ for $(s_0, b)$ ((5) of Figure 17) and the molecule $m_{s_0,b}$ exists in the solution pot, so that these two can hybridize to form a double stranded molecule shown in (6) of Figure 17.

The computation process proceeds until no transition molecule matches the exposed sticky end of the rest of the input molecule, or until the special terminator symbol is cleaved, forming an output molecule having the sticky end encoding the final state (see (7) of Figure 17). The latter means that the input string $w$ is accepted by the automaton $M$.

In summary, the implemented automaton consists of a restriction nuclease and ligase for its hardware, and the software and input are encoded by double stranded DNA, and programming amounts to choosing appropriate software molecules. Upon mixing solutions containing these components, the automaton processes the input molecule by a cascade cycle of restriction, hybridization and ligation, producing a detectable output molecule that encodes the final state of the automaton.

In the implementation, a total amount of $10^{12}$ automata sharing the same software ("transition molecules") run independently and parallel on inputs in 120 $\mu\ell$ solution at room temperature at a combined rate of $10^9$ transitions per second with a transition fidelity greater than 99.8%, consuming less than

$10^{10}$W (for the detailed reports of the experimental results, refer to Benenson et al. (2001)).

## 7.2. *Computing with structured molecules*

One of the most popular structural formations a single stranded DNAs take is the *hairpin structure* which is a kind of intramolecular reactions and is caused by autonomously hybridizing the complementary sequences within the single stranded DNAs. It is a natural idea to think of utilizing this self-assembly function for DNA computing, and in fact, there are several attempts in both theory and lab experiment reported.

### 7.2.1. *Whiplash PCR method*

The first attempt of realizing molecular structural computing in terms of hairpin formation is *Whiplash PCR* (WPCR, for short) proposed by Hagiya et al. (1999). This provides us with a simple implementation of a finite state machine.

WPCR utilizes the following facts that a single stranded DNA can anneal to a portion of itself, forming a *hairpin structure* and that polymerase extension can perform a task of *reading out data* written on a DNA sequence (note that forming a hairpin structure is based on the intramolecular reaction).

Suppose that a finite state automaton $M$ having transition rules $\delta(p, a) = q$ and $\delta(q, b) = r$ is to be implemented. Then, the DNA implementation of $M$ is conceptually carried out using WPCR as follows.

(1) *Designing transition rules*: WPCR involves a sophisticated coding design of a single stranded DNA sequence, which is illustrated in (1) of Figure 18, where a rectangle represents a DNA sequence called *stopper*. Specifically, states $p, q$ and an input symbol $a$ are encoded using `C`, `G` and `T`, while the stopper is simply `AAA`. Further, $\overline{p}$ is encoded as the complementary sequence of $p$ (see (1) of Figure 18).

(2) *Annealing molecules to form a hairpin structure*: By annealing, the complementary sequences $p$ and $\neg p$ come to hybridize, then a polymerase extension takes place and the extension will stop when it encounters the stopper sequence, where the polymerization buffer is assumed to miss `T` nucleotide. This enables the stopper `AAA` to stop the polymerase extension, eventually forming a hairpin structure (see (2) of Figure 18).

(3) *Denaturation*: Then, by denaturation, the molecule becomes again a single stranded DNA molecule (see (3) of Figure 18). Note that one of the two ends of the ssDNA[10] retains the current state $q$ by reading it out as its complementary sequence $\neg q$.

(4) *Carrying out the next transition*: The process from (1) through (3) mentioned above can be repeated to perform further simulation of state

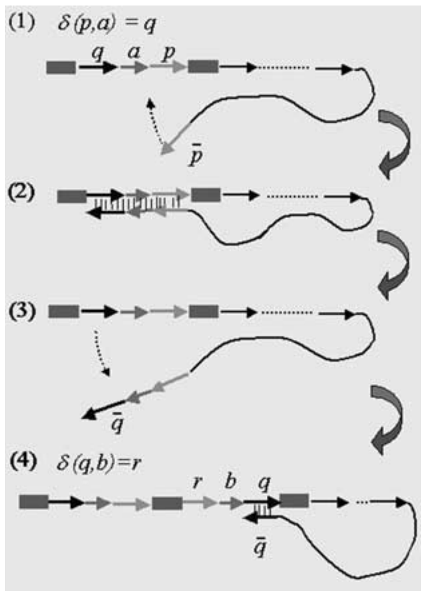


*Figure 18.* Computing process of finite state machine by whiplash PCR.

transition of $M$, because the current state represented by $\neg q$ is now ready for performing another simulation for the next transition $\delta(q, b) = r$.

Recent progress on WPCR techniques can be found in Komiya et al. (2001) where the successful implementation of input/output interface and the successful multiple transition steps on solid phase are reported.

**WPCR with I/O interface**

In the history of DNA computing, in particular at its initial stage, it was common to take DNAs as “memory device” so that DNA computing has been regarded as a computing paradigm of the “single-instruction, multiple-data” (so-called SIMD) model. On the other hand, in order to realize what is called “multiple-instruction, multiple-data (MIMD) model” within DNA molecules, it is necessary that DNAs can implement not only a program but also I/O interface as well.

In the implementation of WPCR with I/O interface, it is of great importance to keep the independence of each molecular machine, but at the same time it is also necessary to prevent an intermolecular reaction. By adopting a solid phase method, WPCR model with I/O interface can successfully embody the MIMD computing paradigm.

Figure 19 illustrates how to realize I/O interface in WPCR model: (1) The input oligomer (indicated by the shaded bar) encoding the state $p$ on the right half anneals to the $3'$-end of the molecular machine. (2) Then, the $3'$-end is

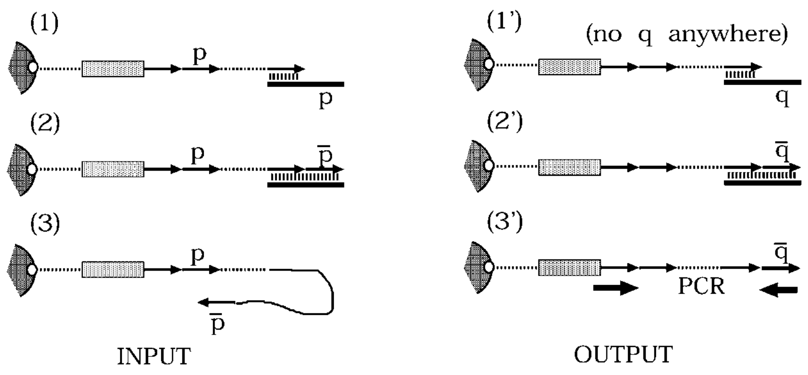


*Figure 19.* The input/output interface.

extended, forming the initial state $\overline{p}$, where $\overline{p}$ is the complementary sequence of $p$. (3) Invoke transition steps by searching and annealing to the sequence $p$. (1′) and (2′) For the output reaction, first the readout sequence $q$ is copied to the 3′-end of the machine by means of polymerase extension. (3′) A molecular machine stops its transition, because there is no sequence $q$ in the transition table. To finalize the process, the machine is amplified by PCR.

Finally, an isothermal technique to perform successive WPCR has been developed and it is reported that successive 4 step transitions were successfully detected.

### 7.2.2. *SAT engines*

As previously discussed, SAT is one of the well-known NP-complete problems where a given Boolean formula one has to decide whether or not the formula is satisfiable (i.e., there exists a truth value assignment under which the truth value of the formula is "true").

As its naming suggests, *SAT Engine* (Sakamoto et al., 2000) has been developed for solving the SAT problem, in which the mechanism of hairpin formation is successfully employed to detect and remove many "inconsistent assignments" to a given Boolean formula.

Using an example, the idea of the SAT algorithm via DNA molecules is outlined. Consider a Boolean formula $F$ with 3 variables and consisting of 5 clauses:

$$
\begin{aligned}
F = {} & C_1 \cdot C_2 \cdots C_5 \\
= {} & (a \vee b \vee c) \wedge (a \vee \neg b \vee c) \wedge (\neg a \vee b \vee c) \wedge (\neg a \vee b \vee \neg c) \wedge (\neg a \vee \neg b \vee \neg c)
\end{aligned}
$$

where a variable $x$ and its negation $\neg x$ are called *literal*. A truth value assignment is a mapping which for each variable assigns 1 or 0 (true or false). A formula $F$ is *satisfiable* if and only if there exists an truth value assignment with which the value of $F$ is true. In our running example, since there is a truth value assignment $(a, b, c) = (1, 1, 0)$ which leads to $F = 1$, $F$ is satisfiable. If each clause $C_i$ in $F$ contains at most $k$ literals, then $F$ is called an instance from $k$-SAT problem.

**DNA SAT algorithm**

First, let us consider a string $u_F = t_1 t_2 \cdots t_5$ over the alphabet $\Gamma = \{a, \neg a, b, \neg b, c, \neg c\}$, where each $t_i$ is an arbitrary literal chosen from $C_i$, for each $i = 1, 2, \cdots, 5$. A string $u_F$ is called *literal string* of $F$. For example, $aab\neg c\neg c$ and $acbb\neg c$ are literal strings of $F$. Each literal $t_i$ chosen from $C_i$ can be regarded as the value with which $C_i = 1$ (in our example, a literal string $aabb\neg c$ is interpreted as $(a, b, c) = (1, 1, 0)$ and it makes $F$ consistently true, while a literal string $acbb\neg c$ is inconsistent with a variable $c$, i.e., it contains both $c$ and $\neg c$, failing to make $F$ true). A literal string of $F$ is satisfiable if it can make $F$ true, and it is unsatisfiable otherwise.

Now, SAT algorithm comprises the following three steps;

(Step 1): For a formula $F$ with $n$ clauses, make the set $LS(F)$ of all literal strings of $F$ whose length are all $n$ (in case of 3-SAT problem, $LS(F)$ consists of at most $3^n$ literal strings).

(Step 2): Remove from $LS(F)$ any literal string that is unsatisfiable.

(Step 3): After removing in Step 2, if there is any literal string remaining, then that is a satisfiable one, so that the formula $F$ turns out to be satisfiable. Otherwise, $F$ is not.

These can be implemented in biomolecular experimental techniques in the following manner:

[Step 1]: By ligating each DNA molecule representing each literal, make all elements of the set $LS(F)$ in parallel.

[Step 2]: A variable $x$ and its negation $\neg x$ are encoded so that these two are just complementary. Therefore, a literal string containing $x$ and $\neg x$ is encoded into a molecule containing two subsequences which are complementary (see (1) of Figure 20). Thus, an encoded molecule for an unsatisfiable literal string can form a *hairpin structure* (see (2) of Figure 20).

[Step 3]: Remove all DNA molecules that form hairpin structures. If there remains any molecule, then detect it and read the sequence for decoding the variable assignment (see Figure 21).

**Implementation and experiments**

The wet lab experiment was carried out by Sakamoto et al. (2000) where the following Boolean formula was considered to solve:

$$F = (a \vee b \vee \neg c) \wedge (a \vee c \vee d) \wedge (a \vee \neg c \vee \neg d)$$

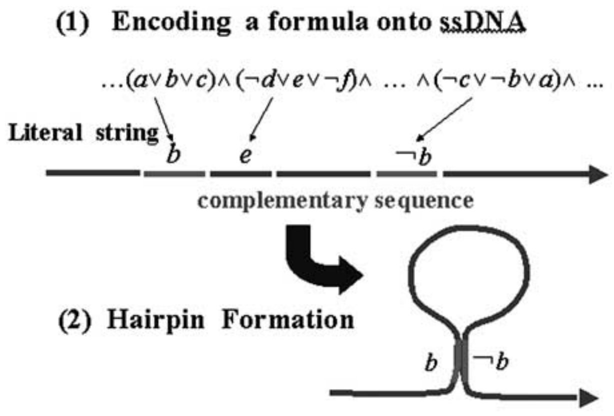

(1) Encoding a formula onto ssDNA
…(a∨b∨c)∧(¬d∨e∨¬f)∧ … ∧(¬c∨¬b∨a)∧ …
Literal string
b
e
¬b
complementary sequence
(2) Hairpin Formation
b
¬b

*Figure 20.* Encoding a formula onto ssDNA and hairpin formation.

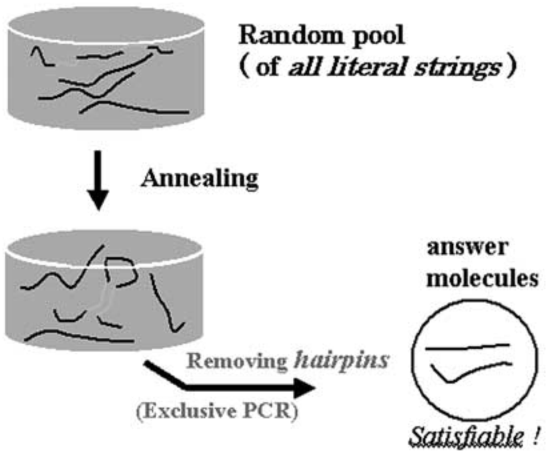


*Figure 21.* SAT algorithmic schema.

$$\wedge(\neg a \vee \neg c \vee d) \wedge (a \vee \neg c \vee e) \wedge (a \vee d \vee \neg f)$$
$$\wedge(\neg a \vee c \vee d) \wedge (a \vee c \vee \neg d) \wedge (\neg a \vee \neg c \vee \neg d) \wedge (\neg a \vee c \vee \neg d).$$

The total number of literal strings of $F$ is $3^{10}(= 59,049)$ among which only 24 literal strings are satisfiable (in fact, these 24 literal strings are for the identical truth value assignment).

In their experiment, each literal was conceptually encoded onto a single stranded DNA molecule (called *literal DNA*) of length 30 as follows:

| | |
|---|---|
| $a$: | TTGGTTGATAGACCCAGGATCGAGTGCCAT |
| $b$: | TTGGCATAAGTTGCCAGGCAGGAACTCCAT |
| $c$: | TTGGAACGTAGTACCAGGAGTCTCCTCCAT |
| $d$: | TTGGTGATACGGACCAGGCCTTCTTACCAT |
| $e$: | TTGGTTGCCCTCTCCAGGTGACTAATCCAT |
| $f$: | TTGGACTGCTCATCCAGGACTGAAGACCAT, |

while the negative counterpart $\neg x$ of $x$ ($x$ in $\Gamma = \{a, b, c, d, e, f\}$) was encoded as the complementary sequence of $x$.

(Step 1) was actually carried out by concatenating with ligase all literal DNAs for $x$ and $\neg x$ ($x \in \Gamma$), where in an actual implementation, each literal DNA was designed as a dsDNA[11] with sticky ends at either side, playing a role of a linker for concatenation (this is because ligase used for concatenation only acts to not ssDNAs but dsDNAs). Another purpose of a linker is to ensure that the resultant of concatenating literal DNAs contains exactly one literal from each clause. A final pot of random pool consisting of all molecules for $LS(F)$ was created autonomously. The experimental result of (Step 1) is reported to have been successful.

In (Step 2) each dsDNA in the pot was first heated to separate into two ssDNAs, and one of which was washed out. Then, remaining ssDNAs are now ready to form *hairpin structures* if they are DNAs encoding unsatisfiable assignments. In other words, if there exist ssDNAs remaining without forming hairpin structures, then it means that there exist truth value assignments for a given $F$ (that is, the formula $F$ is satisfiable). Note that each literal DNA contains a recognition site of a restriction enzyme *Bst*NI, and this hidden sequence will play an important role, as seen below.

In (Step 3), by using a restriction enzyme *Bst*NI, the hairpin structure formed in (Step 2) is cut out, so that all DNAs containing hairpin structures loose a chance to survive for the next procedure in which applying PCR will multiply remaining DNAs for sequencing them.

In practice, 6 literal strings were finally obtained which include

$$bc\neg d\neg ae\neg f\neg ac\neg a\neg a \text{ and } bc\neg d\neg ae\neg f\neg a\neg d\neg a\neg a.$$

It is reproted that all those literal strings were for the unique assignment: $(a, b, c, d, e, f) = (0, 1, 1, 0, 1, 0)$ which provides, in fact, a correct solution to the example formula.

## 7.3. *Hairpin languages in DNA computing*

Formal language characterizations on the computing power of molecular structures are explored in Păun et al. (2001).

The molecules used in the first experiment in the DNA computing reported in (Adleman, 1994) were linear non-branching DNA molecules. One of the important developments in DNA computing was the use of more involved molecules, which do perform specific computations essentially through the process of self-assembly. Some of these molecules are quite sophisticated and “custom made” (see, e.g., Jonoska et al., 1998; LaBean et al., 1999; Lagoudakis et al., 1999; Winfree et al., 2000; Winfree et al., 1999).

Molecules with hairpin structure(s) form a natural extension of linear non-branched molecules, because a hairpin results from a linear molecule that folds on itself. Hairpin molecules found quite many applications in DNA computing. For example, as we have seen above, the use of "hairpin formation" of single-stranded DNA molecules in solving 3-SAT problems is demonstrated in Sakamoto et al. (2000). Also, a model of molecular computing based on self-assembly called YAC uses the power of hairpin formation in finalizing its computation process (Yokomori, 1999a).

It should be noted that the two computation models mentioned above share a common schema (which follows the general computation strategy called "filtering" and formalized in language theoretic terms by "computing by carving" in Păun (1999)): (i) first preparing the initial random pool of molecules, (ii) then extracting only target molecules with (or without) hairpin formation from the pool.

This naturally brings about the following question: what sort of problems in general can be computed by such a schema? In other words, how powerful is this computation schema? Investigating this question leads one to consider sets of string molecules which contain complementary sequences of potential hairpin formations, that is, the set of strings containing a pair of complementary subsequences.

In (Păun et al., 2001), several types of such "hairpin languages" are considered and their language theoretic complexity (using the Chomsky hierarchy) are investigated.

A *Watson-Crick* morphism over an alphabet $V$ is a coding $h : V \longrightarrow V$ (extended in the natural way to $h : V^* \longrightarrow V^*$) such that $h(h(a)) = a$ for each $a \in V$. We may have $h(a) = a$ for some or for all symbols $a \in V$. If $h(a) = b$, then (obviously $h(b) = a$ and) we say that $a, b$ are *complementary* to each other. The mirror image of $x \in V^*$ is denoted by $mi(x)$. Unless otherwise stated, $V$ is an arbitrary fixed alphabet containing at least two symbols. Also, $h$ is an arbitrary Watson-Crick morphism on $V$.

When forming a hairpin molecule it is necessary that the annealed sequence is "long enough" in order to ensure the stability of the construct, so we always impose that this sequence is of length at least $k$, for a given $k$.

Let then $k$ be a positive integer. The *unrestricted hairpin language* (of degree $k$) is defined by

$$uH_k = \{zvwxy \mid z, v, w, x, y \in V^*, x = mi(h(v)), \text{ and } |x| \geq k\}.$$

By imposing restrictions on the location where the annealing may occurr, one gets various sublanguages of $uH_k$ (Figure 22 illustrates the hairpin constructions corresponding to these languages):

$$bH_k = \{vwxy \mid v, w, x, y \in V^*, x = mi(h(v)), \text{ and } |x| \geq k\},$$

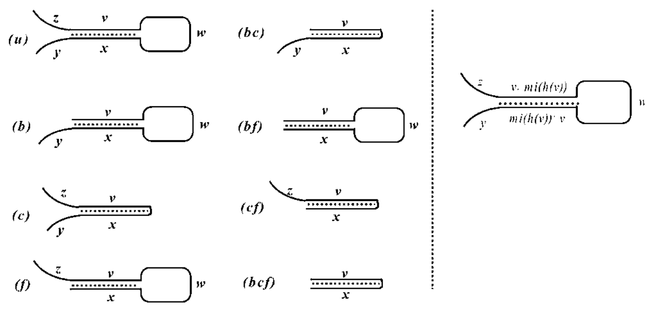


*Figure 22.* (Left) Hairpin constructions corresponding to eight languages; (Right) A hairpin formation to be removed in solving DHPP.

$$cH_k = \{zvxy \mid z, v, x, y \in V^*, x = mi(h(v)), \text{ and } |x| \geq k\},$$
$$fH_k = \{zvwx \mid z, v, w, x \in V^*, x = mi(h(v)), \text{ and } |x| \geq k\},$$
$$bcH_k = \{vxy \mid v, x, y \in V^*, x = mi(h(v)), \text{ and } |x| \geq k\},$$
$$bfH_k = \{vwx \mid v, w, x \in V^*, x = mi(h(v)), \text{ and } |x| \geq k\},$$
$$cfH_k = \{zvx \mid z, v, x \in V^*, x = mi(h(v)), \text{ and } |x| \geq k\},$$
$$bcfH_k = \{vx \mid v, x \in V^*, x = mi(h(v)), \text{ and } |x| \geq k\}.$$

The computation schemas mentioned above fit here as follows: in Sakamoto et al. (2000) one subtracts a language $uH_k$, for some $k$, from a (regular) language, while in YAC (Yokomori, 1999a) one intersects a given (linear) language with $bcH_1$. We will therefore consider languages of the form $\alpha H_k$ and $V^* - \alpha H_k$, for $\alpha \in \{u, b, c, f, bc, bf, cf, bcf\}$. We begin by considering the languages $\alpha H_k$.

THEOREM 14.
*(1). All languages $\alpha H_k$, for $\alpha \in \{u, b, f, c, bf\}$ and $k \geq 1$, are regular.*
*(2). The languages $bcH_k$, $cfH_k$, $bcfH_k$, $k \geq 1$, are linear, but not regular.*

In order to show that those in (2) are not regular, we consider the alphabet $V = \{a, b\}$, with $a, b$ complementary to each other, and intersect these languages with the regular language $a^+b^+$. We obtain then:

$$bcH_k \cap a^+b^+ = \{a^n b^m \mid m \geq n \geq k\},$$
$$cfH_k \cap a^+b^+ = \{a^n b^m \mid n \geq m \geq k\},$$
$$bcfH_k \cap a^+b^+ = \{a^n b^n \mid n \geq k\},$$

and none of these languages is regular. Hence, none of the languages from the statement of the theorem is regular.

The next consequence is of interest from the viewpoint of DNA computing by hairpin removing.

THEOREM 15. *If $L \in \mathcal{FA}$ for a family $\mathcal{FA}$ which is closed under intersection with regular languages, then $L - \alpha H_k \in \mathcal{FA}$, for all $\alpha \in \{u, b, c, f, bf\}$ and all $k \geq 1$.*

**Complements of hairpin languages**

In turn one can consider languages $bcH_k, cfH_k$, and $bcfH_k$. By (2) of Theorem 14, we know already that the complements of $bcH_k, cfH_k, bcfH_k$ are not necessarily regular. However, the languages $bcfH_k, k \geq 1$, are not "very non-regular" in the following sense that for each $k \geq 1$, we have that $V^* - bcfH_k$ is a linear language.

For more details, the next results are obtained:

THEOREM 16.
*(i) If $L$ is a regular language, then $L - bcfH_k, k \geq 1$, is a linear language which does not have to be regular.*
*(ii) If $L$ is a context-free language, then $L - bcfH_k, k \geq 1$, is a context-sensitive language which does not have to be context-free.*

On the other hand, for the languages $bcH_k$ and $cfH_k$, the situation is more involved.

THEOREM 17.
*(1). The languages $V^* - bcH_k,\ V^* - cfH_k, k \geq 1$, are not context-free.*
*(2). If $L$ is a regular language, then the languages $L - bcH_k$ and $L - cfH_k, k \geq 1$, are context-sensitive, but not necessarily context-free.*

It remains as an *open problem* to investigate the place of the languages $L - bcH_k$ and $L - cfH_k$ with respect to families of languages intermediate between the context-free and the context-sensitive families (such as matrix or other regulated rewriting families (Dassow and Păun, 1989)).

**A hairpin solution to DHPP**

As a further example of hairpin computation schema discussed here, one can also present a solution for the Directed Hamiltonian Path Problem as well.

The suggested solution method begins with making a random pool of all possible solution path sequences which start at the initial city and end with the goal city, with the length exactly corresponding to a correct path, i.e., the number of cities $N$ times the length of DNA sequence encoding each city. This task may take a certain amount of time and requires conventional and/or recently developed DNA computing techniques. By $L$ we denote the set of all such DNA sequences (then, it holds that $L \subseteq C^N$, where $C =$

$\{c_1, \ldots, c_N\}$ is the set of encoding sequences for cities). Here we may assume that $L$ is eventually obtained as a set of single-stranded DNA sequences. Now, from $L$ we want to remove all sequences containing double occurrences of an identical city, which is performed by utilizing hairpin formation. We encode each city $c_i$ as $v \cdot mi(h(v))$ for an appropriate $v$, as shown in (Right) of Figure 22. If a single-stranded DNA sequence from $L$ contains double occurrences of $c_i$, then it will form a hairpin structure. Thus, we have only to detect whether or not the final pool obtained by removing all hairpin molecules from $L$ is empty.

Thus, in spite of its simplicity, "hairpin computation" as the underlying schema for DNA computing seems to be (unexpectedly) quite promising.

We have seen some basic classification results for hairpin languages, and indicated some further use of hairpin structures in DNA computing. There seems to be a need for a systematic theory of "hairpin computation", i.e., computation *essentially* using the formation of hairpins, although only the beginning of such a theory has been introduced here.

## 8. Conclusion

### 8.1 *Supplementary notes*

This last section is devoted to supplement some of the recent developments that missed being referred in this paper.

First, a topic of such kinds concerns *molecular logic models*. It is known that any Boolean circuit can be realized using only NAND gates, where NAND (meaning "Not AND") takes two binary inputs and produces as output 0 (if both inputs are 1) and 1 (otherwise). Such a circuit is called NAND circuit. Amos et al. (1998) and Ogihara and Ray (1999) discuss the way of simulating a NAND circuit with DNA molecules. Each NAND gate is encoded on a single stranded molecule comprising three subsequences two of which are for two inputs and the third one is for the output. In order to realize *invisible wiring* between two NAND gates, a distinguished idea is employed where two subsequences to be connected with wiring are encoded to be complementary each other, so that once being released, the fragment from the NAND gate may become mobile to reach and hybridize with its counterpart.

Another method for evaluating a Boolean DNF formula with DNA molecules is proposed in connection with an interesting application of DNA computing to learning problem of Boolean formulas in Sakakibara (2001a). Each Boolean formula in DNF is first encoded onto a single stranded DNA where an idea is to make use of the "stopper" trick employed in Whiplash

PCR (Hagiya et al., 1999; Sakamoto et al., 1999) for being able to make each term of the formula an individual evaluation in a parallel manner. Further, this method is also successfully applied to a problem in genome informatics (gene expression profiles analysis) in Sakakibara and Suyama (2000). Further, it is demonstrated that Whiplash PCR is nicely applicable to simulate a logical inference with DNA molecules (Kobayashi, 1999). For a logic formula, e.g., $c(X, Z) : -a(X, Y), b(Y, Z)$ to be executed, the problem is how to make copies of argument information $X$ of $a(X, Y)$ to that of $c(X, Z)$, and how to check the equivalence of $Y$ from $a(X, Y)$ and $b(Y, Z)$. Kobayashi has shown that using Whiplash PCR and the stopper trick together with append operation, these two tasks can be readily realized with DNA molecules.

Second, unique is the work reported by Nishikawa et al. (1999; 2001) concerning a "DNA computer" simulator based on *VNA* (*virtual nucleic acid*). It turned out that the simulator can be effective in checking the validity and feasibility of proposed DNA computing algorithms before actual implementation.

Third, there are quite a number of molecular computing models to be referred, most of which achieve the Turing universal computability. Those include (not exhaustively at all but) Watson-Crick Automata (Freund et al., 1999), DNA-EC Model based on string equivalence checking (Yokomori and Kobayashi, 1999), Insertion-Deletion Systems (Kari el al., 1999), a great number of extended variants of H-systems (Păun, 1996), (Păun et al., 1998), and a series of works on P-systems (Păun, 2002).

Finally, it should be remarked that with the help of electronic interface device, Suyama's group has developed a hybrid hardware of DNA computers which is the world first implementation and the only one that has been in operation (at least to author's knowledge) (Yoshida and Suyama, 2000). It is reported with the DNA computer they have succeeded at recent experiment in solving an instance of 3-SAT of 43 clauses with 10 variables, for the latest best performance, which took 50 hours for computation and 50 hours for error-trimming.

## 8.2 *Future perspectives*

At the beginning when Adleman's groundbreaking lab experiment was reported in 1994, it was mostly understood that DNA computers were trying to compete or even surpass conventional electronic computers. However, unlike the early desire and expectation from DNA computing paradigm, nowadays it is widely recognized that this new area of molecular computing should aim at being a broader discipline of research dealing with from fundamentals to applications of information processing at the molecular level. In fact, research on applying molecular computing to biotechnology/nanotechnology has already been actively conduced (see, e.g., Sakakibara and Suyama, 2000).

A sequence design of DNA molecules is one of the most important and well studied issues in the area of molecular computing. A natural generalization of this sequence design for DNA computing will lead us to a concept of *molecular programming* that purposes to establish a programming methodology for having control over chemical reactions of biomolecules. That is, one is aiming at developing a systematic design methodology to put one's planned biochemical reaction into practice. These programs comprise two aspects of programming: one for programming DNA sequences (to create a desired self-assembly complex) and the other for programming a bio-experimental protocol. A good cooperative work of these two programs is expected to materialize biomolecular structures or molecular systems of one's goal (Hagiya, 2001).

One of the hardest issues to overcome about molecular computing models is to prevent the computing process from various kinds of errors. An idea introduced in *Amorphous computing* (Abelson et al., 2000) recently proposed seems to be promising in this regard, because the computing model is in its nature designed from such a philosophy as fault-tolerant computing. Also, it is surely useful to consider the problem from a viewpoint of exploring a possibility of developing *approximate computing* models, and computing models based on majority decision seem to be worth further investigation (Sakakibara, 2001b).

Finally, the reader who may have interests in more details about various topics discussed in this paper is cordially advised to consult reference papers or appropriate bibliographic sources listed below.

## Acknowledgements

The author is very much grateful to Prof. G. Rozenberg for giving him an opportunity to make this survey article possible. He is also indebted to Prof. M. Hagiya and his project members for constant support and warm encouragement.

This work is supported in part by the Okawa Foundation for Information and Telecommunications no.01-22, Waseda University Grant for Special Research Projects no.2001B-009, and Grant-in-Aid for Scientific Research on Priority Area no.14085205, Ministry of Education, Culture, Sports, Science and Technology of Japan.

## Notes

[1] In an actual experiment by Adleman, a DNA molecule of length 20 was used for encoding each node and edge.
[2] In his article, Head refers to DNA recombination as "splicing".
[3] In a practical sense, one may set $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$. However, we consider here arbitrary strings of any alphabet $\Sigma$ rather than four letters of DNAs.
[4] Theoretically $R$ can be infinite. Further, since $R$ is a set of strings, it is possible to take $R$ as a member of Chomsky's hierarchy.
[5] A language family closed under operations: union, intersection with regular sets, kleene closure, homomorphism, inverse homomorphism, is called full AFL (Abstract Family of Language).
[6] This fact gives the base of a proof technique called *rotate-simulate method* to prove that a computing model has the universal computability.
[7] See, e.g., (Albert et al., 1998) for biochemical background of symport/antiport.
[8] The rules to be used and the objects to evolve are randomly chosen.
[9] In each step, all objects which can evolve must do it.
[10] Single stranded DNA.
[11] Double straded DNA.

## References

Adleman L (1994) Molecular computation of solutions to combinatorial problems. Science 266: 1021–1024

Adleman L (1996) On constructing a molecular computer. In: Lipton RJ and Baum EB (eds) DNA Based Computers, Series in Mathematics and Theoretical Computer Science, Vol. 27, pp. 1–22. American Mathematical Society

Abelson H, Allen D, Coore D, Hanson C, Homsy G, Knight TF Jr., Nagpal R, Rauch E, Sussman GJ and Weiss R (2000) Amorphous computing. Communications of the ACM 43(5): 74–82

Albert B et al. (1998) Essential Cell Biology. An Introduction to the Molecular Biology of the Cell. Garland Publ. Inc. New York, London

Amos M, Dunne PE and Gibbons A (1998) DNA simulation of Boolean circuits. In: Koza et al. (eds) Proc. of the Third Annual Conference On Genetic Programming, pp. 679–683. Morgan Kaufmann

Arita M, Hagiya M and Suyama A (1997) Joining and rotating data with molecules. IEEE International Conference on Evolutionary Computation, pp. 243–248 The revised version available from `http://ylab-gw.cs.uec.ac.jp/MCP/`

Arita M, Nishikawa A, Hagiya M, Komiya K, Gouzu H and Sakamoto K (2000) Improving sequence design for DNA computing. Proceedings of 5th Genetic and Evolutionary Computation Conference, pp. 875–882. Las Vegas

Arita M, Nishikawa A and Hagiya M (2000) Improving sequence design for DNA computing. Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2000, July 10–12, pp. 875–882. Las Vegas, Nevada

Baum EB et al. (eds) (1996) Second Annual Meeting on DNA Based Computers. Princeton University, Department of Computer Science

Baum EB (1996) DNA sequences useful for computation. Second Annual Meeting on DNA Based Computers, pp. 122–130. Princeton University, Department of Computer Science

Benenson Y, Paz-Elizur T, Adar R, Keinan E, Livneh Z and Shapiro E (2001) Programmable and autonomous computing machine made of biomolecules. Nature 414: 430–434

Berry G and Boudol G (1992) The chemical abstract machine. Theoretical Computer Science 96: 217–248

Besozzi D, Ferretti C, Mauri G and Zandron C (2002) Parallel rewriting P systems with deadlock. Preliminary Proc. of 8th International Meeting on DNA Based Computers, pp. 171–183. Hokkaido University

Birge RR (1995) Protein-based computer. Scientific American 272: 66–71

Boneh D, Dunworth G, Lipton RJ and Sgall J (1995) On the computational power of DNA, Princeton CS tech-report CS-TR-499-95, 1995, available from `http://www.cs.princeton.edu/dabo/biocomp.html`

Boneh D, Dunworth G, Sgall J and Lipton RJ (1996) Making DNA computers error resistant. Second Annual Meeting on DNA Based Computers, pp. 102–110. Princeton University, Department of Computer Science

Brauer W, Ehrig H, Karhumäki J and Salomaa A (eds) (2002) Formal and Natural Computing. Lecture Notes in Computer Science, Vol. 2300, Springer

Breaker RR and Joyce GF (1994) Emergence of a replicating species from an in vitro RNA evolution reaction. *Proceedings of the National Academy of Sciences USA* 91: 6093–6097

Calude C and Păun Gh (2001) Computing with Cells and Atoms. Taylor and Francis

Calude C, Păun Gh, Rozenberg G and Salomaa A (2001) Multiset Processing. Lecture Notes in Computer Science, Vol. 2235. Springer

Cantor CR and Schimmel PR (1980) Biophysical Chemistry Part III: The Behavior of Biological Macromolecules. W.H. Freeman and company, New York

Conrad M (1974) Molecular automata. In: Conrad G. and Dal C. (eds) *Physics and Mathematics of the Nervous System*, pp. 419–430. Springer-Verlag, New York

Conrad M (1985) On design principles for a molecular computer. *Comm. ACM* 28(5): 464–480

Conrad M (1992) Molecular computing paradigms. IEEE Computer 25(11): 6–9

Conrad M and Zauner K-P (1998) DNA as a vehicle for the self-assembly model of computing. BioSystems 45(5): 59–66

Csuhaj-Varju E, Kari L and Păun Gh (1996) Test tube distributed systems based on splicing. Computers and AI 15: 211–232

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

Dassow J and Păun Gh (1989) Regulated Rewriting in Formal Language Theory. Springer-Verlag, Berlin

Deaton R, Murphy RC, Garzon M, Franceschetti DR and Stevens SE Jr. (1996) Good encodings for DNA-based solutions to combinatorial problems. Second Annual Meeting on DNA Based Computers, pp. 131–140. Princeton University, Department of Computer Science

Ehricht R, Ellinger T and McCaskill JS (1997) Cooperative amplification of templates by cross-hybridization (CATCH). European Journal of Biochemistry 243: 358–364

Eng T (1999) Linear DNA self-assembly with hairpins generates the equivalent of linear context-free grammars. DNA Based Computers III, DIMACS Series in Discrete Mathematics and Theoretical Computer Science 48: 289–296

Engelfriet J, Rozenberg G and Slutzki G (1980) Tree transducers, L systems, and two-way machines. Journal of Computer and System Sciences 20: 150–202

Fraenkel AS (1999) Protein folding, spin glass and computational complexity. *DNA Based Computers III, DIMACS Series in Discrete Mathematics and Theoretical Computer Science* 48: 101–121

Freund R, Kari L and Păun Gh (1995) DNA computing based on splicing: The existence of universal computers. Technical report, Fachgruppe Informatik, Tech. Univ. Wien. and Theory of Computing Systems 32: 69–112

Freund R, Păun Gh, Rozenberg G and Salomaa A (1999) Watson-crick finite automata. DNA Based Computers III, DIMACS Series in Discrete Mathematics and Theoretical Computer Science 48: 297–327

Gao Y, Garzon M, Murphy RC, Rose JA, Deaton R, Franceschetti DR, and Stevens SE Jr. (1999) DNA implementation of nondeterminism. DNA Based Computers III, DIMACS Series in Discrete Mathematics and Theoretical Computer Science 48: 137–148

Garzon M and Jonoska N (1998) The bounded complexity of DNA computing. Preliminary Proceedings, Fourth International Meeting on DNA Based Computers, pp. 71–82. University of Pennsylvania

Geffert V (1991) Normal forms for phrase-structure grammars. RAIRO. Th. Inform. and Appl. 25: 473–496

Gehani A and Reif JH (1978) Micro flow bio-molecular computation. Preliminary Proc. of Fourth International Meeting on DNA Based Computers, pp. 253–266. University of Pennsylvania

Gehani A, LaBean TH and Reif JH (1998) DNA-based cryptography. Preliminary Proc. of Fifth International Meeting on DNA Based Computers, pp. 231–245. MIT

Guarnieri F, Fliss M and Bancroft C (1996) Making DNA add. Science 273: 220–223

Guatelli KM, Whitfield JC, Kwoh DY, Barringer KJ, Richman DD and Gingeras TR (1990) Isothermal, in vitro amplification of nucleic acids by a multienzyme reaction modeled after retroviral replication. Proceedings of the National Academy of Sciences USA 87, pp. 1874–1878 [erratum, Proc Natl Acad Sci U S A, 87, 7797, 1990]

Hagiya M (1999) Perspectives on molecular computing. New Generation Computing 17: 131–151

Hagiya M (2001) From molecular computing to molecular programming. DNA6, Sixth International Meeting on DNA Based Computers, Lecture Notes in Computer Science 2054: 89–102

Hagiya M, Arita M, Kiga D, Sakamoto K and Yokoyama S (1999) Towards parallel evaluation and learning of Boolean $\mu$-formulas with molecules. DNA Based Computers III, DIMACS Series in Discrete Mathematics and Theoretical Computer Science 48: 57–72

Hagiya M and Ohuchi A (2002) Preliminary Proceedings of the Eighth International Meeting on DNA Based Computers. Hokkaido University, June 10–13

Head T (1987) Formal language theory and DNA: an analysis of the generative capacity of specific recombinant behaviors. Bull. Math. Biology 49: 737–759

Head T (1999) Circular suggestions for DNA computing. In: Carbone A, Gromov M and Pruzinkiewcz P (eds), Pattern Formation in Biology, Vision and Dynamics, pp. 325–335. World Scientific, Singapore and London

Head T, Rozenberg G, Bradergroen RS, Breek CKD, Lommerse PHM and Spaink HP (2000) Computing with DNA by operating on plasmids. BioSystems 57: 870–893

Head T, Păun Gh and Pixton D (1997) Language theory and molecular genetics. Generative mechanisms suggested by DNA recombination. In: Rozenberg G and Salomaa A (eds) Handbook of Formal Languages Vol. 2, pp.295-360. Springer-Verlag, Berlin

Head T, Yamamura M and Gal S (1999) Aqueous computing: Writing on molecules. Congress on Evolutionary Computation, July 6–9, 1999, Mayflower Hotel, Washington D.C., USA, pp. 1006–1010

Hennie FC (1965) One-tape, off-line Turing machine computations. Information and Control 8: 553–578

Jonoska N, Karl SA and Saito M (1998) Three dimensional DNA structures in computing. In: Kari L (ed), Proc. of the Fourth DIMACS Meeting on DNA Based Computers. University of Pennsylvania, June 16–19, pp. 189–200

Kari L (1996) DNA computers: Tomorrow's reality. Tutorial in the Bulletin of EATCS 59: 256–266

Kari L and Landweber LF (1999) Computing with DNA. Methods in Molecular Biology

Kari L, Păun Gh, Rozenberg G, Salomaa A and Yu S (1998) DNA computing, sticker systems, and universality. Acta Informatica 35: 401–420

Kari L, Păun Gh, Thierrin G and Yu S (1999) At the crossroads of DNA computing and formal languages: Characterizing recursively enumerable languages using insertion-deletion systems. DNA Based Computers III, DIMACS Series in Discrete, Mathematics and Theoretical Computer Science 48: 329–346

Karp R, Kenyon C and Waarts O (1996) Error-resilient DNA computations. Seventh ACM-SIAM Symposium on Discrete Algorithms: 458–467

Kim SM (1997) Computational modeling for genetic splicing systems. SIAM J. on Computing 26(5): 1284–1309

Kobayashi S (1999) Horn clause computation with DNA molecules. Journal of Combinatorial Optimization 3: 277–299

Kobayashi S and Sakakibara Y (1998) Multiple splicing systems and the universal computability, submitted to *Theoretical Computer Science*, 1998.

Kobayashi S, Yokomori T, Sanpei G and Mizobuchi K (1997) DNA Implementation of Simple Horn Clause Computation. IEEE International Conference on Evolutionary Computation 213–217

Komiya K, Sakamoto K, Gouzu H, Yokoyama S, Arita M, Nishikawa A and Hagiya M (2001) Successive state transitions with I/O interface by molecules. DNA6, Sixth International Meeting on DNA Based Computers, Lecture Notes in Computer Science 2054: 17–26

Kozyavkin SA, Mirkin SM and Amirikyan BR (1987) J. Biomol. Struct. Dyn. 5: 119

Kuhn H et al. (1998) Nucleic Acid Research 26: 582

Kurtz SA, Mahaney SR and Royer JS (1996) Active transport in biological computing. Second Annual Meeting on DNA Based Computers, pp. 111–121. Princeton University, Department of Computer Science

LaBean TH, Winfree E and Reif JH (1999) Experimental progress in computation by self-assembly of DNA tilings. In: Winfree E and Gifford DK (eds), DNA Based Computers VDIMACS Series in Discrete Mathematics and Theoretical Computer Science 54: 123–140

Lagoudakis MG and LaBean TH (1999) 2D DNA self-assembly for satisfiability. In: Winfree E and Gifford DK (eds), DNA Based Computers VDIMACS series in Discrete Mathematics and Theoretical Computer Science 54: 141–154

Laun E and Reddy K (1997) Wet splicing systems. Proc. of the Third DIMACS Meeting on DNA Based Computers. University of Pennsylvania, June 23–25, pp. 115–126

Lipton RJ (1995) DNA solution of hard computational problems. Science 268: 542–545

Lipton RJ and Baum EB (eds) DNA based computers. Series in Mathematics and Theoretical Computer Science 27. American Mathematical Society

Liu Q, Wang L, Frutos AG, Condon AE, Corn RM and Smith LM (2000) DNA computing on surfaces. Nature 403: 175–179

Lomakin A and Frank-Kamenetskii MD (1998) J. Mol. Biol. 276: 57

Mao C, LaBean TH, Reif JH and Seeman NC (2000) Logical computation using algorithmic self-assembly of DNA triple-crossover Molecules. Nature 407: 493–496

Marathe A, Condon AE and Robert MC (1999) On combinatorial DNA word design. DIMACS Series in Discrete Mathematics and Theoretical Computer Science 44: 75–87

Morimoto N, Arita M and Suyama A (1999) Solid phase DNA solution to the Hamiltonian path problem. *DNA Based Computers III, DIMACS Series in Discrete Mathematics and Theoretical Computer Science* 48: 193–206

Morimoto N, Arita M and Suyama A (1997) Stepwise generation of Hamiltonian path with molecules. Proc. of Bio-Computing and Emergent Computation, pp. 184–192

Nemoto N, Miyamoto-Sato E, Husimi Y and Yanagawa H (1997) In vitro virus: Bonding of mRNA bearing puromycin at the $3'$-terminal end to the C-terminal end of its encoded protein on the ribosome in vitro. FEBS Lett. 414: 405–408

Nishikawa A and Hagiya M (1999) Towards a system for simulating DNA computing with whiplash PCR. Proc. of the 1999 Congress on Evolutionary Computation 2: 960–966

Nishikawa A, Hagiya M and Yamamura M (1999) Virtual DNA simulator and protocol design by GA. Proc. of the Genetic and Evolutionary Computation Conference (GECCO99), vol. 2, pp. 1810–1816

Nishikawa A, Yamamura M and Hagiya M (2001) DNA Computation Simulator Based on Abstract Bases. Soft Computing 5: 25–38

Ogihara M and Ray A (1998) DNA-based self-propagation algorithm for solving bounded-fan-in Boolean Circuits. Proc. of the Third Annual Genetic Programming Conference (GP98), pp. 725–730

Ogihara M and Ray A (1999) Simulating Boolean circuits on a DNA computer. Algorithmica 25: 239–250

Ouyang Q, Kaplan PD, Liu S and Libchaber A (1997) DNA solution of the maximal clique problem. Science 278: 446–449

Păun Gh (1995) A challenge for formal language theorists. EATCS Bulletin 57: 183–194

Păun Gh (1996) Five (plus two) universal DNA computing models based on the splicing operation. Proc. of 2nd DIMACS Workshop on DNA Based Computers, Princeton, pp. 67–86

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

Păun Gh (1996) On the splicing operation. Discrete Applied Mathematics 70: 57–79

Păun Gh (2000) Computing with membranes. Journal of Computer and System Sciences 61(1): 108–143. Also, *TUCS Research Report*, No. 208, November 1998, `www.tucs.fi`.

Păun Gh (ed) (1998) *Computing with Bio-Molecules – Theory and Experiments*. Springer

Păun Gh (1999) (DNA) Computing by carving. Soft Computing 3(1): 30–36

Păun Gh (2002) Membrane Computing. Natural Computing Series, Springer

Păun A and Păun Gh (2002) The power of communication: P-systems with symport/antiport. New Generation Computing 20(3): 295–305

Păun Gh and Rozenberg G (1998) Sticker systems. Theoretical Computer Science 204: 183–203

Păun Gh, Rozenberg G and Salomaa A (1998) DNA Computing: New Computing Paradigms. Springer-Verlag

Păun Gh, Rozenberg G and Yokomori T (2001) Hairpin languages. International Journal of Foundations of Computer Science 12(6): 837–847

Peterson WW and Weldon EJ (1972) Error-Correcting Codes, 2nd ed. MIT Press

Pitt L and Valiant GL (1988) Computational limitations on learning from examples. J. Assoc. Comput. Mach. 35(4): 965–984

Pixton D (1995) Linear and circular splicing systems. Proc. of 1st Intern. Symp. on Intell. in Neural and Biological Systems, pp. 38–45. IEEE, Herndon

Poland D and Scheraga H (1970) Theory of Helix-Coil Transitions in Biopolymers. Academic Press, New York

Post E (1943) Formal reductions of the general combinatorial decision problem. American Journal of Mathematics 65: 197–215

Reif J (1995) Parallel molecular computation. Seventh Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'95), pp. 213–223

Reif J (1998) Paradigms for biomolecular computation. In: Calude CS, Casti J and Dinneen MJ (eds), Unconventional Models of Computation, pp. 72–93. Springer-Verlag, New York

Reif J (2002) The emerging discipline of biomolecular computation in the US. New Generation Computing 20(3): 217–236

Roweis S, Winfree E, Burgoyne R, Chelyapov NV, Goodman MFM Rothemund PWK and Adleman LM (1999) A sticker based model for DNA computation. DNA Based Computers II, DIMACS Series in Discrete Mathematics and Theoretical Computer Science 44: 1–29

Roberts RW and Szostak JW (1997) RNA-peptide fusions for the in vitro selection of peptides and proteins. Proceedings of the National Academy of Sciences USA 94: 12297–12302

Rose JA and Deaton RJ (2001) The fidelity of annealing-ligation: A theoretical analysis. Proc. of the Sixth International Meeting on DNA Based Computers(DNA6), Lecture Notes in Computer Science 2054, pp. 321–246

Rozenberg G and Salomaa A (eds) (1997) Handbook of Formal Languages, 3 volumes. Springer-Verlag, Berlin

Rozenberg G and Salomaa A (1999) DNA computing: New ideas and paradigms. LNCS 1644, pp. 106–118. Springer-Verlag

Salomaa A (1985) Computation and Automata. Cambridge University Press, Cambridge

Saitou K (1998) Self-assembling automata: A model of conformational self-assembly. Pacific Symposium on Biocomputing'98, pp. 609–620

Saitou K and Jakiela MJ (1996) On classes of one-dimensional self-assembling automata. Complex Systems 10(6): 391–416

Sakakibara Y (2001a) Solving computational learning problems of Boolean formulae on DNA computers. Proc. of the 6th International Meeting on DNA Based Computers(DNA6), Lecture Notes in Computer Science 2054: 220–230

Sakakibara Y (2001b) Population computation and majority inference in test tube. Proc. 7th International Meeting on DNA Based Computers, pp. 84–93

Sakakibara Y and Ferretti C (1997) Splicing on tree-like structures. Proc. of the Third DIMACS Meeting on DNA Based Computers. University of Pennsylvania, June 23–25, pp. 348–358

Sakakibara Y and Kobayashi S (2001) Sticker systems with complex structures. Soft Computing 5: 114–120

Sakakibara Y and Suyama A (2000) Intelligent DNA chips: Logical operation of gene expression profiles on DNA computers. Genome Informatics 2000 (Proc. of 11th Workshop on Genome Informatics), pp. 33–42. Universal Academy Press

Sakamoto K, Gouzu H, Komiya K, Kiga D, Yokoyama S, Yokomori T and Hagiya M (2000) Molecular computation by DNA hairpin formation. Science 288: 1223–1226

Sakamoto K, Kiga D, Komiya K, Gouzu H, Yokoyama S, Ikeda S, Sugiyama H and Hagiya M (1999) State transitions by molecules. BioSystems 52(1–3): 81–91

Sambrook J and Russell DW (2001) Molecular Cloning: A Laboratory Manual, Vol. 1–3. Cold Spring Harboer Laboratory Press

SantaLucia J Jr. (1998) Proc. Natl. Acad. Sci. 95: 1460

Shimada T, Hagiya M, Arita M, Nishizaki S and Tan C-L (1995) Knowledge-based simulation of regulatory action in lambda Phage. First International IEEE Symposium on Intelligence in Neural and Biological Systems (INBS'95), pp. 92–99. Also in International Journal of Artificial Intelligence Tools 4(4): 511–524

Siromoney R, Subramanian KB and Rajkumar Dare V (1992) Circular DNA and splicing systems. Proc. of Parallel Image Analysis, LNCS 654, pp. 260–273. Springer-Verlag, Berlin

Smullyan RM (1961) Theory of Formal Systems. Princeton University Press

Stemmer WP (1994) Rapid evolution of a protein in vitro by DNA shuffling. Nature 370: 389–391

Suyama A, Arita M and Hagiya M (1997) A heuristic approach for Hamiltonian path problem with molecules. In: Koza et al. (eds) Proc. of Genetic Programming Conference (GP-97), pp. 457-462.

Suzuki Y, Fujiwara Y, Takabayashi J and Tanaka H (2001) Artificial life applications of a class of P systems: Abstract rewriting systems on multisets. In: Calude et al. (eds), Multiset Processing, Lecture Notes in Computer Science 2235: 299–346

Takahara A and Yokomori T (2002) On the computational power of insertion-deletion systems. Proc. of 8th International Meeting on DNA-based Computers, pp. 139–150

Taylor MF, Pauauskis JD, Weller DD and Kobzik L (1996) In vitro efficacy of morpholino-modified antisense oligomer directed against tumor necrosis factor-alpha mRNA. Journal of Biological Chemistry 271: 17445–17452

Tsuchihashi Z, Khosla M and Herschlag D (1993) Protein enhancement of hammerhead ribozyme catalysis. Science 262: 99–102

Vaintsvaig MN and Liberman EA (1973) Formal description of cell molecular computer. Biofizika 18: 939–942

Wartell R and Benight A (1985) PHYSICS REPORTS (Review Section of Physics Letters), vol. 126, p. 67

Waterman MS (1995) Introduction to Computational BiologyMap, Sequences, and Genomes. Chapman & Hall

Weiss R and Knight TF Jr. (2000) Engineered communications for microbial robotics. DNA6, Sixth International Meeting on DNA Based Computers. Leiden Center for Natural Computing, June 13–17, 2000, pp. 5–19

Wetmur JG (1999) Physical chemistry of nucleic acid hybridization. In: Rubin H and Wood D (eds), DNA Based Computers III, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pp. 1–23

Winfree E, Yang X and Seeman NC (1999) Universal computation via self-assembly of DNA: Some theory and experiments. DNA Based Computers II, DIMACS Series in Discrete Mathematics and Theoretical Computer Science 44: 191–213

Winfree E, Liu F, Wenzler LA and Seeman NC (1998) Design and self-assembly of two-dimensional DNA crystals. Nature 394: 539–544

Winfree E (1996) On the computational power of DNA annealing and ligation. In: Lipton R and Baum E (eds), DNA Based Computers: Proc. of a DIMACS Workshop. DIMACS Series in Discrete Mathematics and Theoretical Computer Science Vol. 27, American Mathematical Society, pp. 199–221

Winfree E (1998) Simulations of computing by self-assembly. In: Kari L (ed), Proc. of the Fourth DIMACS Meeting on DNA Based Computers. University of Pennsylvania, June 16–19, pp. 213–239

Winfree E, Eng T and Rozenberg G (2000) String tile models for DNA computing by self-assembly. In: Proc. of the 6th International Meeting on DNA Based Computers, Leiden University, June 13–17, pp. 65–84

Winfree E and Rothemund PWK (2000) The program-size complexity of self-assembled squares. In: Proc. of 32th Annual ACM Symposium on Theory of Computing

Wittung P, Nielsen PE, Buchardt O, Egholm M and Nordén B (1994) DNA-like double helix formed by peptide nucleic acid. Nature 368: 561–563

Yamamoto Y, Komiya S and Husimi Y (2001) Stabilized 3SR against evolutionary instability. Chem. Lett., submitted

Yamamoto Y, Suzuki M and Husimi Y (2001) Dynamics in a 3SR evolution reactor and its applications. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, in press

Yamamura M, Hiroto Y and Matoba T (2001) Another realization of aqueous computing with peptide nucleic acid. Preliminary proceedings of the 7th International Meeting on DNA Based Computers (DNA7), pp. 219–230

Yokomori T, Kobayashi S and Ferretti C (1997) On the power of circular splicing systems and DNA computability. Proc. of IEEE International Conference on Evolutionary Computation, pp. 219–224

Yokomori T and Kobayashi S (1999) DNA-EC: A model of DNA computing based on equality checking. DNA Based Computers III, DIMACS Series in Discrete Mathematics and Theoretical Computer Science 48: 347–360

Yokomori T (1999a) YAC: Yet another computation model of self-assembly. In: Winfree E and Gifford DK (eds), DNA Based Computers V DIMACS series in Discrete Mathematics and Theoretical Computer Science, Vol. 54, pp. 155–169

Yokomori T (1999b) Computation = self-assembly + conformational change: Toward new computing paradigms. Proc. of 4th International Conference on Developments in Language Theory(DLT'99), Aachen, July, pp. 21–30

Yokomori T, Sakakibara Y and Kobayashi S (2002) A magic pot: Self-assembly computation revisited. In: Brauer W, Ehrig H, Karhumaki J and Salomaa A (eds), Formal and Natural Computing, LNCS 2300, pp. 418–429. Springer

Yoshida H and Suyama A (2000) Solutions to 3-SAT by breadth first search. In: Winfree E and Gifford DK (eds), DNA Based Computers V, DIMACS series in Discrete Mathematics and Theoretical Computer Science, Vol. 54, pp. 9–22

Yurke B (2002) DNA based molecular motors. Preliminary Proc. of 8th International Meeting on DNA Based Computers, Hokkaido University, pp. 185–197

Molecular Computer Project HomePage: `http://nicosia.is.s.u-tokyo.ac.jp/MCP/index_j.html`