UC'07

LANGUAGE THEORY IN BIOCOMPUTING
WORKSHOP - August 15, 2007

Language Theory in Biocomputing

Proceedings

UC'07 – Unconventional Computation
Kingston, Ontario, Canada

Edited by Michael Domaratzki and Kai Salomaa

Table of Contents

Preface

The **Language Theory in Biocomputing Workshop** was held on August 15, 2007 as part of the **Unconventional Computation UC'07** conference. The conference venue was the Four Points Hotel in Kingston, Ontario, Canada.

These proceedings contain the six accepted contributions. Each contribution was refereed by at least three Program Committee members or external reviewers.

Revised versions of the papers from this workshop will be considered for publication in a special issue of the *International Journal of Foundations of Computer Science*.

We would like to thank the sponsors of the UC'07 conference for their support:

- Faculty of Arts and Science, Queen's University
- Fields Institute
- School of Computing, Queen's University
- Department of Biology, Queen's University
- Office of Research Services, Queen's University
- MITACS–Mathematics of Information Technology and Complex Systems
- The Campus Bookstore, Queen's University
- IEEE, Kingston Section

Program Committee

Mark Daley	University of Western Ontario
Michael Domaratzki (co-chair)	University of Manitoba
Lucian Ilie	University of Western Ontario
Stavros Konstantinidis	St. Mary's University
Ian McQuillan	University of Saskatchewan
Andrei Păun	Louisiana Tech University
Kai Salomaa (co-chair)	Queen's University

Sponsors

School of Computing

Queen's University

Department of Biology

Queen's University

Faculty of Arts and Science

Queen's University

IEEE

MITACS

FIELDS

Office of Research Services

Queen's University

The Campus Bookstore at Queen's University

Automaton Models Inspired by Peptide Computing*

M. Sakthi Balan

Department of Computer Science
The University of Western Ontario
London, Ontario, Canada, N6A 5B7
sakthi@csd.uwo.ca

Abstract. We define two accepting devices inspired by peptide computing called string binding-blocking automata and rewriting binding-blocking automata. In both the devices there is an option to mark some symbols – in one case, it is used to postpone reading of marked symbols to a later part of processing and in other case, it is used to store some information. These ideas are inherited from peptide computing. We show a nice hierarchy between these two systems and the previous model, the binding-blocking automaton.

1 Introduction

We propose two new automaton models inspired by peptide computing: string binding-blocking automaton and rewriting binding-blocking automaton. In both the automaton models we impart a function for blocking and unblocking of a string of symbols. Blocking and unblocking facilitates postponing the reading of some string of symbols to a later part of processing. The idea of blocking some symbols from being read by the head is borrowed from peptide computing [5] where some regions of peptide sequences are blocked from being processed and later unblocked when required.

Peptide computing proposed by H. Hug and R. Schuler [7] is a computational model where computing is done through interactions between peptides and antibodies. Interactions between peptides and antibodies occur by binding of antibodies to some specific spots, called epitopes, present in the peptide sequence. Elementary operations are binding/removal of antibodies from peptides that resembles a rewriting system like Turing machine.

In [7], H. Hug and R. Schuler showed how to solve the satisfiability problem using peptide computing. The following paper [5] presented a model to solve two further NP-complete problems – *Hamiltonian circuit* and *exact cover by 3-set*. Moreover in [5], a simulation of Turing machine by peptide computing is presented to show peptide computing is computationally complete. In all the above results processing is done by permanent or temporary elimination of some (part of) peptide sequences by attaching some higher affinity antibodies. In this paper we use the idea of temporary elimination called blocking in our model.

In our earlier work [4] we proposed a slightly different automaton model called binding-blocking automata (BBA), which is a model based on classical finite state automata together with blocking and unblocking of symbols and priority relation in reading some string of symbols. Blocking of symbols is analogical to temporary elimination used in peptide computing. While the model presented in [4] follows blocking of symbols, our model proposed here uses blocking of strings.

In this paper we study how blocking improves the acceptance power of the system. We show a nice hierarchy between all the three systems: binding-blocking automata, string binding-blocking automata and rewriting binding-blocking automata.

* A very short and preliminary version of this paper appeared in [1].

We prove that the power of these systems increases in the order given above. We also show that the power of rewriting binding-blocking automata is equivalent to that of Turing machine.

The main idea of this study is, to see how the system performs if ideas of peptide-antibody interactions in peptide computing [3, 5] are inserted into the definition of finite state automata. A theoretical model for peptide computing is proposed in [2, 3]. The computation in the model presented in [2] is not a sequential one. In this paper we use the ideas inherited from peptide computing but we use it in a sequential device, finite state automata.

In section 2, we present some preliminaries and notations followed in this paper. In section 3, we introduce string binding-blocking automaton and study the acceptance power of the proposed model. In section 4, we define rewriting binding-blocking automaton and prove that its acceptance power is equivalent to that of Turing machine. The paper concludes with some remarks in section 5.

2 Preliminaries

Let V be a finite set of symbols and let $y \in V^*$. We denote the set of all substrings of y by $sub(y)$; the set of all proper substrings of y as $Sub(y)$; the set of all prefixes of y as $pre(y)$; and, set of all proper prefixes of y as $Pre(y)$.

Random Context Grammars

Random context grammars is a variation of context-free grammars. There are two restrictions in this model when applying a rule: permitting contexts and forbidding contexts. Both the sets of contexts contain a subset of non-terminals, which are checked for its presence in the sentential form when applying a rule. In the following paragraphs we present a brief description of random context grammars and some of the results which we use in our paper. We follow the notations of [6] in the definition of random context grammars. For more details of this model refer [6].

A random context grammar is a system

$$G \;=\; (V_N, V_T, P, S)$$

where V_N, V_T, S are in a usual Chomsky grammar, and P is a finite set of random context rules, that is triplets of the form

$$(\alpha \longrightarrow \beta, Q, R),$$

where $\alpha \longrightarrow \beta$ is a rewriting rule over $V_G (V_G \;=\; V_N \cup V_T)$ and Q and R are subsets of V_N. For $x, y \in V_G^*$, we write $x \Longrightarrow y$ iff $x \;=\; x'\alpha x''$ for some $x', x'' \in V_G^*, (\alpha \longrightarrow \beta, Q, R)$ is a triple in P, all symbols of Q appear in $x'x''$, and no symbol of R appears in $x'x''$. Q is called the permitting context of $\alpha \longrightarrow \beta$ and R is the forbidding context of this rule.

The language generated by this grammar G is defined as

$$L(G) \;=\; \{x \mid x \in V_T^* \text{ and } S \stackrel{*}{\Longrightarrow} x\}$$

where again $\stackrel{*}{\Longrightarrow}$ is the reflexive and transitive closure of $\Longrightarrow$.

The grammar is of type i (λ-free)iff the rules $\alpha \longrightarrow \beta \in P$ are of type $i(\lambda$-free), $i \in \{0, 1, 2, 3\}$, respectively. We use random context grammar with context-free rules (type 2) unless otherwise mentioned.

We denote the families of languages generated by random context grammars with appearance checking by $\mathcal{L}(RC, CF, ac)$. If no appearance checking features are involved (i.e., $R = \phi$ for each rule $(\alpha \to \beta, Q, R)$ in random context grammars) then we erase the letters ac and obtain the families $\mathcal{L}(RC, CF)$. Some of the relevant results in this (see [6]) are,

Theorem 1.
1. $\mathcal{L}(CF)$ *is strictly contained in* $\mathcal{L}(RC, CF - \lambda)$.
2. $\mathcal{L}(RC, CF - \lambda) \subseteq \mathcal{L}(RC, CF) \subset \mathcal{L}(RC, CF, ac) = \mathcal{L}(RE)$.
3. $\mathcal{L}(RC, CF - \lambda) \subset \mathcal{L}(RC, CF - \lambda, ac) \subset \mathcal{L}(RC, CF, ac) = \mathcal{L}(RE)$.
4. $\mathcal{L}(RC, CF - \lambda, ac) \subset \mathcal{L}(CS)$. □

In the above $\mathcal{L}(CF)$ and $\mathcal{L}(CS)$ are the families of context-free languages and context sensitive languages respectively. $CF - \lambda$ denotes the context-free rules without lambda rules.

Binding-Blocking Automata

We present the definition of binding-blocking automaton proposed in [4] and state some of the results which we use in this paper.

A Binding-Blocking Automaton is a construct

$$\mathcal{P} = (Q, V, E, \delta, q_0, R, \beta_b, \beta_{ub}, Q_{accept}, Q_{reject})$$

where $Q = Q_{block} \cup Q_{unblock} \cup Q_{general}$ (pairwise disjoint sets), $q_0 \in Q$ is the start state, V is a finite set of symbols, E is the finite subset of V^*, δ is the transition function from $Q \times (E \cup \{\varepsilon\}) \longrightarrow 2^Q$, R is the partial order relation (called affinity relation) on E, simply $R \subseteq E \times E$, β_b is the blocking function from $Q_{block} \longrightarrow 2^V$, β_{ub} is the unblocking function from $Q_{unblock} \longrightarrow 2^V$ and $Q_{accept} \cup Q_{reject} \subseteq Q_{general}$ where Q_{accept} is the set of accepting states and Q_{reject} is the set of rejecting states.

The symbols which have been read by the head are called *marked* symbols, which are blocked are called as *blocked* symbols. Suppose a sequence of symbols, say $x = a_1 a_2 \cdots a_n,\ a_i \in V,\ 1 \leq i \leq n$ is read, then it is marked and denoted by $\begin{matrix} a_1\ a_2 \cdots a_n \\ \#\ \#\ \cdots\ \# \end{matrix}$. Suppose the symbol $a \in V$ is blocked then it is denoted by $\begin{matrix} a \\ \$ \end{matrix}$. All other symbols which are not blocked and not read by the head are denoted by $\begin{matrix} a \\ - \end{matrix}$. If no symbol is read or blocked in a string x then simply we denote it by $\begin{matrix} x \\ - \end{matrix}$. Likewise if all the symbols have been read by the system then it is denoted by $\begin{matrix} x \\ \# \end{matrix}$. Similarly if all the symbols in the string x are blocked then it is denoted by $\begin{matrix} x \\ \$ \end{matrix}$.

The initial configuration of the automaton will be,

$$\begin{matrix} q_0\ a_1\ a_2 \cdots a_n \\ \uparrow\ -\ -\ \cdots - \end{matrix}$$

where $a_1 a_2 \cdots a_n$ is the input string on the tape of the system.

We have four types of transitions of head which give two types of instantaneous description of the system. The instantaneous description or configuration of the automaton is,

$$\begin{matrix} a_1\ a_2 \cdots a_{i-1}\ q\ a_i\ a_{i+1} \cdots a_n \\ X\ X \cdots X \quad \uparrow Y\ Y \quad \cdots Y \end{matrix}$$

wherein

- in one case $X \in \{\#, \$\}$ and $Y \in \{-, \#, \$\}$ and
- in the other case $X \in \{-, \#, \$\}$ and $Y \in \{-, \#, \$\}$.

The first transition is called as *leftmost and blocked reading*, denoted by $\vdash_{(l,b)}$, the second transition as *leftmost and free reading*, denoted by $\vdash_{(l,f)}$, the third one as *locally leftmost and blocked reading*, denoted as $\vdash_{(ll,b)}$ and the last one *locally leftmost and free reading*, denoted as $\vdash_{(ll,f)}$. Leftmost reading is the one where reading always starts from the leftmost unmarked and unblocked symbol; locally leftmost reading need not start from the leftmost unmarked and unblocked symbol; free reading is the one where a blocked symbol can be read if there is a transition defined on it; whereas, blocked reading is the one where no blocked symbol is read.

We denote by $\vdash_D^*$ the reflexive and transitive closure of the relation $\vdash_D$ (denoting the four transitions) where $D \in \{(l,b),(l,f),(ll,b),(ll,f)\}$.

The language acceptance of the BBA $\mathcal{P}$ is defined as

$$L_D(\mathcal{P}) \;=\; \{w \in V^* \mid \begin{smallmatrix} q_0 & w \\ \uparrow & \_ \end{smallmatrix} \vdash_D^* \begin{smallmatrix} w & q_f \\ \# & \uparrow \end{smallmatrix},\ \text{for } q_f \in Q_{accept}\}$$

where $D \in \{(l,b),(l,f),(ll,b),(ll,f)\}$.

We denote the family of BBA following (l,b) transition as $BBA_{(l,b)}$, likewise $BBA_{(l,f)}$, $BBA_{(ll,b)}$ and $BBA_{(ll,f)}$ are also defined. If we are not bothered about whether the system is following free or blocked transition, then we simply omit f or b from the notation. Same is the case for l or ll also. Similarly, the family of languages $\mathcal{BBA}_{(l,b)}$ denote the set of languages accepted by BBA in $BBA_{(l,b)}$. The family of languages $\mathcal{BBA}_{(l,f)}$, $\mathcal{BBA}_{(ll,b)}$ and $\mathcal{BBA}_{(ll,f)}$ are similarly defined.

If the affinity relation R is empty then the family of BBA is denoted by BBA_{np}. If the system reads only one symbol at a time then the BBA is called as a simple BBA and its family is denoted by $SBBA$. The set of languages accepted by X_y is denoted by $\mathcal{X}_y$, where $X \in \{BBA, SBBA\}$, $y \in \{p, np\}$ and $\mathcal{X} \in \{\mathcal{BBA}, \mathcal{SBBA}\}$. If $D \in \{(l,b),(l,f),(ll,b),(ll,f)\}$ and y are to be specified then we denote the systems by $X_{y,D}$ and the set of languages by $\mathcal{X}_{y,D}$. If the system is referred simply as BBA then it denotes an arbitrary system.

We state some of the results, proved in [4], in the next theorem.

Theorem 2.

- $\mathcal{BBA}_{(l,b)} \subset \mathcal{BBA}_{(l,f)}$.
- $\mathcal{BBA}_{(ll,b)} \subseteq \mathcal{BBA}_{(ll,f)}$.
- $\mathcal{BBA}_{np,(l,x)} \subset \mathcal{BBA}_{p,(l,x)},\ x \in \{b,f\}$.
- *For every language $L \in k$-$SNFA$ there is a language $L' \in BBA_{(ll,b)}$ such that L can be written in the form $h^{-1}(L')$ where h is a homomorphism from L to L'.*
- *For every $L \in BBA_{(l,x)}$ there exists $BBA_{(ll,x)}$ $\mathcal{P}$ such that $L(\mathcal{P}) = L$ where $x \in \{b,f\}$ ($L(\mathcal{P})$ denotes the language accepted by $\mathcal{P}$).*
- $BBA_{(l,b)} \subset BBA_{(ll,b)}$.
- *For every $L \in \mathcal{BBA}_{np,(l,b)}$ there is a k-NFA which accepts the same language.*

□

The blocking can also be called as marking since when we speak of blocking some symbols, it intuitively means marking those symbols.

3 String Binding-Blocking Automata

The basic structure is a finite state automaton but with an additional flexibility to postpone reading those string of symbols to a later part of processing by blocking some string of symbols. The finite control of the automaton is divided into three sets of states, namely blocking states, unblocking states and general reading states. In general reading state, the head can read a string of symbols from its present position. When a string is read they are marked. In a blocking a state, some string of symbols (starting from the position of the head) are blocked from being read by the head. So, only those symbols which are not marked and not blocked are read by the head. We note that the blocking in this model is done in string-wise fashion unlike the BBA where the blocking is done over individual symbols. Similarly in an unblocking state, the system unblocks the corresponding strings for further reading.

The formal definition of the system is as follows:

Definition 1. *A String Binding-Blocking Automaton (StrBBA) is a construct*

$$\mathcal{P} = (Q, V, E, \delta, q_0, R, \beta_b, \beta_{ub}, Q_{accept})$$

where $Q = Q_{block} \cup Q_{unblock} \cup Q_{general}$ *is the set of states (pairwise disjoint);* $q_0 \in Q$ *is the start state,* V *is a finite set of symbols;* E *is the finite subset of* V^**,* δ *is the transition function from* $Q \times (E \cup \{\varepsilon\}) \longrightarrow 2^Q$*;* R *is the partial order relation (called affinity/priority relation) on* E*, i.e.,* $R \subseteq E \times E$*;* β_b *is the blocking function from* $Q_{block} \longrightarrow \mathcal{L}$*;* β_{ub} *is the unblocking function from* $Q_{unblock} \longrightarrow \mathcal{L}'$ *where* $\mathcal{L}$ *and* $\mathcal{L}'$ *are finite set of family of languages over* V*, i.e.,* $\mathcal{L} = \{L_1, L_2, \cdots, L_k\}$*, and* $\mathcal{L}' = \{L'_1, L'_2, \cdots L'_r\}$*; and* $Q_{accept} \subseteq Q$ *where* Q_{accept} *is the set of accepting states.*

For $1 \leq i \leq k$*,* $L_i \in \mathcal{L}$ *is said to be a blocking language.* $\mathcal{L}$ *is called as the family of blocking languages. Similarly for* $1 \leq i \leq r$*,* $L'_i \in \mathcal{L}'$ *is said to be an unblocking language and* $\mathcal{L}'$ *is called as the family of unblocking languages.*

In the sequel we describe how the system behaves in reading and (un)blocking states. Let us suppose the input string is $y (y \in V^*)$. At an instant the system is at any one of the three states - reading state, blocking state or unblocking state. In reading state, the system reads a string of symbols (say l symbols) at a time and moves the head l positions to the right. In the blocking state q, the system blocks a string of symbols, say x where $x \in L$, $L \in \beta_b(q)$, $x \in Sub(y)$ starting from the position of the head. The string x satisfies the maximal property i.e., there exists no $z \in L$ such that $x \in Pre(z)$ and $z \in Sub(y)$ starting from the position of the head. When the system is in the unblocking state q, all the blocked strings of the form $x \in Sub(y)$ and $x \in L$ where $L \in \beta_{ub}(q)$ are unblocked. We note that the head can only read symbols which are neither read nor blocked. If more than one string can be read by the head then the affinity relation R says which string has to be read. At any point of time the head reads the string which has more affinity, i.e., if the head can read either of the strings, x or y $(x, y \in V^+)$ and if $(x, y) \in R$ then y is read.

The symbols that are read by the head are called *marked* symbols; that are blocked are called as *blocked* symbols. Suppose a sequence of symbol $x = a_1 a_2 \cdots a_n$, $a_i \in V,\ 1 \leq i \leq n$ is read, then it is marked and denoted by $\begin{smallmatrix} a_1\ a_2\ \cdots\ a_n \\ \#\ \#\ \cdots\ \# \end{smallmatrix}$. If the string $x \in V^*$ is blocked, then it is denoted by $\begin{smallmatrix} x \\ \$ \end{smallmatrix}$. The symbols which are not blocked and not read by the head are denoted by $\begin{smallmatrix} a \\ \_ \end{smallmatrix}$. If no symbol is read or blocked in a string

x, then we denote it by $\begin{smallmatrix} x \\ - \end{smallmatrix}$. Likewise, if all the symbols in a string x have been read by the system then it is denoted by $\begin{smallmatrix} x \\ \# \end{smallmatrix}$. For simplicity, we denote $\begin{smallmatrix} a_1 & a_2 & \cdots & a_n \\ X_1 & X_2 & \cdots & X_n \end{smallmatrix}$ where $X_i \in \{-, \#, \$\}, 1 \leq i \leq n$ as $\begin{smallmatrix} x \\ \mathcal{X} \end{smallmatrix}$ where $x = a_1 a_2 \cdots a_n$ and $\mathcal{X} = X_1 X_2 \cdots X_n$. If $X_i \in \{\#, \$\}$ then it is denoted by $\begin{smallmatrix} x \\ \mathcal{X}-\{-\} \end{smallmatrix}$.

The initial configuration of the tape is,

$$\begin{smallmatrix} q_0 & a_1 & a_2 & \cdots & a_n \\ \uparrow & - & - & \cdots & - \end{smallmatrix}$$

where $a_1 a_2 \cdots a_n$ is the input string on the tape of the system.

We have two types of instantaneous description (ID) of the system:

1. $\begin{smallmatrix} a_1 & a_2 & \cdots & a_{i-1} & q & a_i & a_{i+1} & \cdots & a_n \\ X & X & \cdots & X & \uparrow & Y & Y & \cdots & Y \end{smallmatrix}$ is an ID for $\mathcal{P}$ and $X \in \{\#, \$\}$ and $Y \in \{-, \#, \$\}$ (called as leftmost); and
2. $\begin{smallmatrix} a_1 & a_2 & \cdots & a_{i-1} & q & a_i & a_{i+1} & \cdots & a_n \\ X & X & \cdots & X & \uparrow & Y & Y & \cdots & Y \end{smallmatrix}$ is an ID for $\mathcal{P}$ and $X \in \{-, \#, \$\}$ and $Y \in \{-, \#, \$\}$ (called as locally-leftmost).

In both the cases, q is the state of the system, head is at the i^{th} position, $a_1 a_2 \cdots a_n$ is the input string on the tape of the system. The first transition always reads a sequence of symbol starting from the symbol which is the leftmost unread and unblocked symbol. So it is equivalent to saying whenever unblocking of symbols take place, the head comes back to the starting symbol on the tape and slides through all the marked and blocked symbols until it sees the first unblocked and unmarked symbol. Unlike the first transition, in the second one, the head reads from the first unblocked and unmarked symbol from its present position. The first transition is called as *leftmost reading*, denoted by $\vdash_l$, and the second transition is called as *locally leftmost reading*, denoted by $\vdash_{ll}$. Both the above transitions are explained below:

Case 1: $q \in Q_{general}$

We say the system makes the move,

$$\begin{smallmatrix} a_1 & a_2 & \cdots & a_{i-1} & q & a_i & a_{i+1} & \cdots & a_j & a_{j+1} & \cdots & a_n \\ X & X & \cdots & X & \uparrow & - & - & \cdots & - & Y & \cdots & Y \end{smallmatrix}$$

$$\vdash_l$$

$$\begin{smallmatrix} a_1 & a_2 & \cdots & a_{i-1} & a_i & a_{i+1} & \cdots & a_j & p & a_{j+1} & \cdots & a_n \\ X & X & \cdots & X & \# & \# & \cdots & \# & \uparrow & Y & \cdots & Y \end{smallmatrix}$$

if $\delta(q, x)$ contains p, where

- $p \in Q,\ x = a_i a_{i+1} \cdots a_j \in V^*,\ 1 \leq i < j \leq n,\ a_k \in V, 1 \leq k \leq n, X \in \{\#, \$\}$ and $Y \in \{-, \#, \$\}$.
- there is no $z \in E$ such that $z > x$ in R with $\delta(q, z)$ being non-empty where either $(z \in Pre(x))$ or $(x \in Pre(z)$ and $a_i \cdots a_j \cdots a_l = z, l \leq n$, and no symbols from a_{j+1} to a_l is blocked or marked).

Similarly we can define the transition for $\vdash_{ll}$; the only change in the above will be $X \in \{-, \$, \#\}$.

Case 2: $q \in Q_{block}$

Let $\beta_b(q) = L$ then we say that system makes the move,

$$\begin{matrix} \cdots & \cdots & q & x & \cdots & \cdots \\ \cdots & \cdots & \uparrow & - & \cdots & \cdots \end{matrix} \vdash_D \begin{matrix} \cdots & \cdots & x & p & \cdots & \cdots \\ \cdots & \cdots & \$ & \uparrow & \cdots & \cdots \end{matrix}$$

if $\delta(q, \varepsilon) = p$, where $p \in Q$, $D \in \{l, ll\}$, $x \in L$ and there is no $z \in L$ such that $x \in Pre(z)$. So the entire string x is blocked from being read.

Case 3: $q \in Q_{unblock}$

Let $\beta_{ub}(q) = L$ then the system makes the following move,

$$\begin{matrix} \cdots & x_1 & \cdots & x_2 & \cdots & q & \cdots & x_l & \cdots \\ \cdots & \$ & \cdots & \$ & \cdots & \uparrow & \cdots & \$ & \cdots \end{matrix} \vdash_l \begin{matrix} p & \cdots & x_1 & \cdots & x_2 & \cdots & x_l & \cdots \\ \uparrow & \cdots & - & \cdots & - & \cdots & - & \cdots \end{matrix}$$

in case of the leftmost reading and

$$\begin{matrix} \cdots & x_1 & \cdots & x_2 & \cdots & q & \cdots & x_l & \cdots \\ \cdots & \$ & \cdots & \$ & \cdots & \uparrow & \cdots & \$ & \cdots \end{matrix} \vdash_{ll} \begin{matrix} \cdots & x_1 & \cdots & x_2 & \cdots & p & \cdots & x_l & \cdots \\ \cdots & - & \cdots & - & \cdots & \uparrow & \cdots & - & \cdots \end{matrix}$$

in the case of locally leftmost reading, if $\delta(q, \varepsilon) = p$, where $p \in Q$ and $x_i \in L$. The head movements after unblocking in the l and ll transitions are different – in l transitions, the head positions to the leftmost unmarked and unblocked symbol, where as in ll transition the head remains at the same position.

We denote by $\vdash_D^*$, the reflexive and transitive closure of the relation $\vdash_D$ where $D \in \{l, ll\}$.

We emphasize here that when blocking is done, a single string is blocked starting from the position of the head and when unblocking occurs several substrings are unblocked.

The language acceptance of the StrBBA system $\mathcal{P}$ is defined as

$$L_D(\mathcal{P}) = \{w \in V^* \mid \begin{matrix} q_0 & w \\ \uparrow & - \end{matrix} \vdash_D^* \begin{matrix} w & q_f \\ \# & \uparrow \end{matrix}, \text{ for } q_f \in Q_{accept}\}.$$

The class of string binding-blocking automata with D-transition is denoted by $StrBBA_D$ and the language accepted by the above classes are denoted by $\mathcal{StrBBA}_D$ respectively. If the blocking languages are finite languages, i.e., every $L_i \in \mathcal{L}$ is a finite language, then the above class is represented by $StrBBA_D(Fin)$. If there is no priority then the above class is denoted by $StrBBA_{np,D}$ and the set of languages accepted by the above class is denoted by $\mathcal{StrBBA}_{np,D}$. Similarly if the blocking languages are finite and there is no priority then it is denoted by $\mathcal{StrBBA}_{np,D}(Fin)$.

We note that unlike in the binding-blocking automata model [4] where we use two more transitions, namely, blocked and free transitions in addition to leftmost and locally leftmost transitions, here we consider only blocked transitions.

3.1 Power of Acceptance

Example 1. We give an example of a StrBBA system $\mathcal{P}$ (without priority) working in l transition which accepts $L = \{a^n b a^n \mid n \geq 1\}$.

$Q_{general} = \{q_0, q_{a_1}, q_{a_2}, q_f\}$,
$Q_{block} = \{q^{block_a}\}$,
$Q_{unblock} = \{q^{unblock_a}\}$,
$Q_{accept} = \{q_f\}$,
$\beta_b(q^{block_a}) = \{a^n b \mid n \geq 0\}$,
$\beta_{ub}(q^{unblock_a}) = \{a^n b \mid n \geq 0\}$,

$\delta(q_0, a) = \{q^{block_a}\}$,
$\delta(q^{block_a}, \varepsilon) = \{q_{a_1}\}$,
$\delta(q_{a_1}, a) = \{q_{a_2}\}$,
$\delta(q_{a_2}, \varepsilon) = \{q_{unblock_a}\}$,
$\delta(q^{unblock_a}, \varepsilon) = \{q_0\}$
$\delta(q_0, b) = \{q_f\}$,

With little work we can see that the above system accepts the language L in l transition. The instantaneous descriptions are depicted in the following for a particular string a^3ba^3.

$q_0\ a\ a\ a\ b\ a\ a\ a$ $\uparrow\ -\ -\ -\ -\ -\ -\ -$	$\rightarrow$	$a\ a\ a\ b\ q^{block_a}\ a\ a\ a$ $\#\ \$\ \$\ \$\ \uparrow\ \ \ -\ -\ -$	$\rightarrow$
$a\ a\ a\ b\ q^{a_1}\ a\ a\ a$ $\#\ \$\ \$\ \$\ \uparrow\ \ -\ -\ -$	$\rightarrow$	$a\ a\ a\ b\ a\ q^{a_2}\ a\ a$ $\#\ \$\ \$\ \$\ \#\ \uparrow\ \ -\ -$	$\rightarrow$
$a\ q^{unblock_a}\ a\ a\ b\ a\ a\ a$ $\#\ \uparrow\ \ \ -\ -\ -\ \#\ -\ -$	$\rightarrow$	$a\ q_0\ a\ a\ b\ a\ a\ a$ $\#\ \uparrow\ \ -\ -\ -\ \#\ -\ -$	$\rightarrow$
$a\ a\ a\ b\ a\ q^{block_a}\ a\ a$ $\#\ \#\ \$\ \$\ \#\ \uparrow\ \ \ -\ -$	$\rightarrow$	$a\ a\ a\ b\ a\ q_{a_1}\ a\ a$ $\#\ \#\ \$\ \$\ \#\ \uparrow\ \ -\ -$	$\rightarrow$
$a\ a\ a\ b\ a\ a\ q_{a_2}\ a$ $\#\ \#\ \$\ \$\ \#\ \#\ \uparrow\ \ -$	$\rightarrow$	$a\ a\ q^{unblock_a}\ a\ b\ a\ a\ a$ $\#\ \#\ \uparrow\ \ \ -\ -\ \#\ \#\ -$	$\rightarrow$
$a\ a\ q_0\ a\ b\ a\ a\ a$ $\#\ \#\ \uparrow\ -\ -\ \#\ \#\ -$	$\rightarrow$	$a\ a\ a\ b\ a\ a\ q^{block_a}\ a$ $\#\ \#\ \#\ \$\ \#\ \#\ \uparrow\ \ \ -$	$\rightarrow$
$a\ a\ a\ b\ a\ a\ q_{a_1}\ a$ $\#\ \#\ \#\ \$\ \#\ \#\ \uparrow\ \ -$	$\rightarrow$	$a\ a\ a\ b\ a\ a\ a\ q_{a_2}$ $\#\ \#\ \#\ \$\ \#\ \#\ \#\ \uparrow$	$\rightarrow$
$a\ a\ a\ q^{unblock_a}\ b\ a\ a\ a$ $\#\ \#\ \#\ \uparrow\ \ \ -\ \#\ \#\ \#$	$\rightarrow$	$a\ a\ a\ q_0\ b\ a\ a\ a$ $\#\ \#\ \#\ \uparrow\ -\ \#\ \#\ \#$	$\rightarrow$
$a\ a\ a\ b\ q_f\ a\ a\ a$ $\#\ \#\ \#\ \#\ \uparrow\ \#\ \#\ \#$			

The above language L is not accepted by any BBA_l. In the case of BBA_l, in order to equate the number of $a's$ on either side of b the BBA system has to first block the symbol a. But blocking $a's$ will block both the strings of a and if the system unblocks to equate the second string with the first string of a then the head comes to the first string of a, since the transition is *leftmost*. So the second string of $a's$ can not be equated with the first string of $a's$. Hence $L \notin \mathcal{BBA}_l$. Relying on the above observations it is possible to give a rigorous proof for the following lemma.

Lemma 1. *There are $StrBBA_l$ systems that accepts languages not accepted by any BBA_l.* □

The above language L is accepted by $StrBBA$ in locally leftmost transition (without priority) by the following example:

Example 2. Following is the $StrBBA$ system $\mathcal{P}$ which accepts

$$L = \{a^nba^n \mid n \geq 1\}$$

in ll transition mode.

$Q_{general} = \{q_0, q_{a_1}, q_{a_2}, q_f\}$,
$Q_{block} = \{q^{block_a}, q^{block_{a,b}}\}$,
$Q_{unblock} = \{q^{unblock_a}, q^{unblock_{a,b}}\}$,
$Q_{accept} = \{q_f\}$,
$\beta_b(q^{block_a}) = \{a^nb \mid n \geq 0\}$,
$\beta_b(q^{block_{a,b}}) = \{w \mid w \in \{a,b\}^*\}$,
$\beta_{ub}(q^{unblock_a}) = \{a^nb \mid n \geq 0\}$,
$\beta_{ub}(q^{unblock_{a,b}}) = \{w \mid w \in \{a,b\}^*\}$,

$\delta(q_0, a) = \{q^{unblock_{a,b}}\}$,
$\delta(q^{unblock_{a,b}}, \varepsilon) = \{q^{block_a}\}$,
$\delta(q_0, b) = \{q_f\}$,
$\delta(q^{block_a}, \varepsilon) = \{q_{a_1}\}$,
$\delta(q_{a_1}, a) = \{q_{a_2}\}$,
$\delta(q_{a_2}, \varepsilon) = \{q^{unblock_a}\}$,
$\delta(q^{unblock_a}, \varepsilon) = \{q^{block_{a,b}}\}$,
$\delta(q^{block_{a,b}}, \varepsilon) = \{q_0\}$.

The above example shows that $StrBBA$, without priority, accepts languages not accepted by BBA_l systems.

The following example shows that $StrBBA_{ll}$ accepts languages not accepted by BBA_{ll}.

Example 3. Following is the StrBBA system $\mathcal{P}$ (with no priority) which accepts

$$L = \{a^{2n}(aca)^n \mid n \geq 1\}$$

in ll transition mode.

$Q_{general} = \{q_1, q_2, q_3, q_4, q_5, q_b, q_f\}$, q_1 is the start state,
$Q_{block} = \{q^{block_a}, q^{block_{aca}}\}$,
$Q_{unblock} = \{q^{unblock_a}, q^{unblock_{aca}}\}$,
$Q_{accept} = \{p_f\}$,
$\beta_b(q^{block_a}) = \{a^{2n} \mid n \geq 1\}$,
$\beta_b(q^{block_{aca}}) = \{(aca)^n \mid n \geq 1\}$,
$\beta_{ub}(q^{unblock_a}) = \{a^{2n} \mid n \geq 1\}$,
$\beta_b(q^{unblock_{aca}}) = \{(aca)^n \mid n \geq 1\}$,

$\delta(q_1, a) = \{q_2\}$,
$\delta(q_2, a) = \{q^{unblock_{aca}}\}$,
$\delta(q^{unblock_{aca}}, \varepsilon) = \{q^{block_a}\}$,
$\delta(q^{block_a}, \varepsilon) = \{q_3\}$,
$\delta(q_3, a) = \{q_4\}$,
$\delta(q_4, c) = \{q_5\}$,
$\delta(q_5, a) = \{p_f, q^{block_{aca}}\}$,
$\delta(q^{block_{aca}}, \varepsilon) = \{q^{unblock_a}\}$,
$\delta(q^{unblock_a}, \varepsilon) = \{q_1\}$.

The above language L is not accepted by any BBA_{ll}. When constructing a BBA_{ll} for L, in order to match a with a aca, the system has to know from where the substring $(aca)^n$ starts. Moreover, in order to equate each a with the substring aca the system has to block all $a's$ then look for aca. But blocking of a will block all $a's$ in the substring aca. This shows the system can neither equate a with aca nor it knows the position where the string aca starts. Hence there is no BBA_{ll} system which accepts the language L. The observations above gives an informal proof for the following lemma.

Lemma 2. *There are $StrBBA_{ll}$ systems that accepts languages not accepted by any BBA_{ll}.* □

It is interesting to see that even if the set of blocking languages are finite languages, the system is able to accept non-regular languages as the following example shows:

Example 4.

$Q_{general} = \{q_0, q_a, q_b\}$,
$Q_{block} = \{q^{block_a}, q^{block_b}\}$,
$Q_{unblock} = \{q^{unblock_a}, q^{unblock_b}\}$,
$Q_{accept} = \{q_0\}$,
$\beta_b(q^{block_a}) = \{a\}, \beta_b(q^{block_b}) = \{b\}$,
$\beta_{ub}(q^{unblock_a}) = \{a\}, \beta_{ub}(q^{unblock_b}) = \{b\}$,

$\delta(q_0, a) = \{q^{block_a}, q_a\}$,
$\delta(q^{block_a}, \varepsilon) = \{q^{block_a}, q_a\}$,
$\delta(q_a, b) = \{q^{unblock_a}\}$,
$\delta(q^{unblock_a}, \varepsilon) = \{q_0\}$,
$\delta(q_0, b) = \{q^{block_b}, q_b\}$,
$\delta(q^{block_b}, \varepsilon) = \{q^{block_b}, q_b\}$,
$\delta(q_b, a) = \{q^{unblock_b}\}$,
$\delta(q^{unblock_b}, \varepsilon) = \{q_0\}$.

The above StrBBA accepts all strings over $\{a, b\}$ with equal number of $a's$ and $b's$. If the system reads a symbol a (b), then it is followed by reading b (a). This makes sure that the string contains equal number of a and b. Hence we have the theorem,

Theorem 3. $REG \subset \mathcal{StrBBA}_{np,D}(Fin), D \in \{l, ll\}$. □

Theorem 4. *For any BBA in BBA_D there is an equivalent $StrBBA$ in $StrBBA_D$ where $D \in \{l, ll\}$.*

Proof. We give the construction for ll-transition. The same construction will also hold for l-transitions. Let $\mathcal{P} = (Q, \Sigma, \delta, q_0, \phi, \beta_b, \beta_{ub}, Q_{accept}, Q_{reject})$ be a BBA working in ll transition. We construct a $StrBBA_{ll}$,

$$\mathcal{Q} = (Q', \Sigma, \delta', q^{\phi}, \beta'_b, \beta'_{ub}, Q'_{accept}, Q'_{reject})$$

as follows:

1. $Q' = \{q^X, q^X_{block,X}, q^X_{unblock,Y} \mid q \in Q,\ X, Y \in 2^V\}$,
2. $Q'_{accept} = \{q^X \mid q \in Q_{accept}, X \in 2^V\}$,
3. $Q'_{reject} = \{q^X \mid q \in Q_{reject}, X \in 2^V\}$,
4. $Q'_{block} = \{q^X_{block,X} \mid q \in Q_{block}, X \in 2^V\}$,
5. $Q'_{unblock} = \{q^X_{unblock,Y} \mid q \in Q_{unblock}, X, Y \in 2^V\}$,
6. $q^X_{block,X} \in \delta'(p^X, a)$ if $q \in \delta(p, a)$ where $p \in Q_{general}$ and $X \in 2^V$,
7. $q^{X \cup \beta_b(p)}_{block, X \cup \beta_b(p)} \in \delta'(p^X, a)$ if $q \in \delta(p, a)$ where $p \in Q_{block}, a \in V \cup \{\varepsilon\}$ and $X \in 2^V$,
8. $q^{X - \beta_{ub}(p)}_{unblock, \beta_{ub}(p)} \in \delta'(p^X, a)$ if $q \in \delta(p, a)$ where $q \in Q_{unblock}, a \in V \cup \{\varepsilon\}$ and $X \in 2^V$,
9. $q^X \in \delta'(q^X_{block,X}, \varepsilon), X \in 2^V$,
10. $q^X_{block,X} \in \delta'(q^X_{unblock,Y}, \varepsilon), X, Y \in 2^V$,
11. $\beta'_b(q^X_{block,X}) = X^*, X \in 2^V$,
12. $\beta'_{ub}(q^X_{unblock,Y}) = Y^*, X, Y \in 2^V$,

In a BBA system when blocking occurs, symbols from the set X are blocked wherever it is present in the tape. But in a $StrBBA$ system only a string, starting from the position of the head, is blocked. Hence when constructing a $StrBBA$ system, we have to take note that all the symbols in the set X, to its right of the head, is blocked at each transition. In order to perform this, we insert states of the form q^X where X denotes the set of symbols blocked when the system is in the state q. So, whenever the system reads a symbol it reaches the state of the form $q^X_{block,X}$ that blocks a string of symbol from the set X, before reading the next symbol. Hence for each reading and blocking transitions we have two transitions – one, to the state of the form $q^X_{block,X}$ which blocks the string over X and the other one, to the state of the form q^X from where next transition is followed. When the system reaches an unblocking state, the first transition gets the system in the state of the form $q^X_{unblock,Y}$ where the symbols over Y are unblocked, and the symbols in X are got after removing all the symbols in Y. In the next move, the system goes to the state of the form $q^X_{block,X}$ which again blocks any string over X. The third move gets back the system to the state of the form q^X.

It is easy to see that if a string y is accepted by $\mathcal{P}$, then it will be accepted by $\mathcal{Q}$ and vice-versa. □

Hence by Lemmas 1 and 2, and by Theorem 4 we have the following

Theorem 5.

$$\mathcal{BBA}_D \subset \mathcal{StrBBA}_D$$

where $D \in \{l, ll\}$. □

We conjecture that the acceptance power of SBBA working in l transition without priority can not exceed the generative power of random-context grammars. The following is our proposed construction of a random-context grammar for a given StrBBA. We note here that this construction works only for some examples. In the following proof we take a strict sub-class of $\mathcal{StrBBA}_{l,np}$ where $L \in \mathcal{StrBBA}_{l,np}$ has no iterative blocking, i.e., there is no additional blocking of strings when there is a blocking of strings present already.

Conjecture 1. For every $L \in \mathcal{StrBBA}_{l,np}$ there exists a random-context grammar RC with context-free rules (without appearance-checking) such that $L(RC) = L$.

Let $\mathcal{P} = (Q, \Sigma, \delta, q_0, \beta_b, \beta_{ub}, Q_{accept}, Q_{reject})$ where $\mathcal{P} \in \mathcal{StrBBA}_{l,np}$. Also, let $L(\mathcal{P}) = L$. In our construction, we assume that the system $StrBBA$ has no iterative blocking. We construct an equivalent random-context grammar $RC = (V_N, V_T, P, S_{q_0})$ as follows:

1. $V_N = \{S_{q_0}, A_q, A'_q, B_q, B'_q, B', E \mid q \in Q\}$,
2. $V_T = \Sigma$,

The production rules of RC includes the following, ($a \in V \cup \{\varepsilon\}$ and $r \in Q$ wherever applicable)

1. $(S_{q_0} \longrightarrow aA_qB', \phi, \phi)$ if $q \in \delta(q_0, a)$,
2. $(A_p \longrightarrow aA_q, \{B', B'_r\}, \phi)$ if $q \in \delta(p, a)$ and $p, q \in Q_{general}$,
3. $(A_p \longrightarrow aA'_q, \{B', B'_r\}, \phi)$ if $q \in \delta(p, a)$ and $q \in Q_{block}$, $p' \in \delta(q, \varepsilon)$ and $L \in \beta_b(q)$,
4. $(B' \longrightarrow B_q, \{A'_q\}, \phi)$,
5. $(B_p \longrightarrow aB_q, \{A'_r\}, \phi)$ if $q \in \delta(p, a)$ and $p, q \in Q_{general}$,
6. $(B_p \longrightarrow aB'_q, \{A'_r\}, \phi)$ if $q \in \delta(p, a)$ and $q \in Q_{unblock}$ and $L \in \beta_{ub}(q)$,
7. $(A'_r \longrightarrow A_q, \{B'_q\}, \phi)$,
8. $(A_p \longrightarrow E, \{B', B'_r\}, \phi)$ if $p \in Q_{accept}$,
9. $(B'_r \longrightarrow \lambda, \{E\}, \phi)$,
10. $(B' \longrightarrow \lambda, \{E\}, \phi)$,
11. $(E \longrightarrow \lambda, \phi, \phi)$.

In the following we explain the above construction. When no blocking takes place the non-terminal A_q generates symbols in the same way as a regular grammar will do. When A_q is generating, B is passive. When blocking occurs for the first time, the RC system introduces the non-terminals A'_q in the sentential and the non-terminal B' takes over the control. The non-terminal A'_q is passive when B_q generates symbols. If $q \in Q_{unblock}$ the control transfers to the non-terminal A'_q which cycles back to A_q to generate symbols as above. This procedure is repeated for every blocking and unblocking. When $q \in Q_{accept}$ the RC system introduces the non-terminal E which stops the generation of symbols.

From the above construction, it seems that this can be extended to $StrBBA$ with iterative blocking. If we prove that the number of blockings occurring in a sentential form is bounded by a finite number, similar to the result that blocking quotient of BBA is finite (see [4]), then the above idea can be extended to all $StrBBA$.

We note here that the above construction uses λ rules but forbidden context is not used.

It should be easy to note here that the language ww is not accepted by StrBBA working in either l or ll transitions. The reason is that the system can not guess correctly from where the duplication starts. Suppose the system has to check for the

duplication the only possibility is that it reads symbols from both the parts of the string, one or a finite number at a time, alternatively, and match them accordingly. But for this procedure to be carried out, the system has to block some symbols (i.e., a suffix of the string w). But the string w is an arbitrary string over $\{a, b\}$. So the blocking has to be done over the language $\{a, b\}^*$. If this is done then the entire input string will be blocked which will make the matching of the symbols impossible. But a random context grammar without appearance-checking generates ww. This makes it clear that StrBBA do not accept all languages accepted by random context grammars with context-free rules and without appearance-checking.

4 Rewriting Binding-Blocking Automata

In this section we define another variant of binding-blocking automaton called as rewriting binding-blocking automaton.

In this variant we use markers and state-affinity relation. The set of markers are generalization of blocking symbols, since we can consider blocking of symbols in BBA and $StrBBA$ as done by a single marker. In this model, we have more than one marker to mark the symbols. There is a finite set of states and a relation called as state-affinity relation, which basically relates each state to a poset over the set of markers. By this relation the system can rewrite in the tape by removing the lower affinity marker by a marker having higher affinity.

The basic model consists of a *finite control*, an infinite tape (one end is fixed and other end is infinite) which is divided into cells, and a *tape head* which scan a cell at a time. The tape has two tracks - the first one consists of the input symbols and the next one contains the markers. Each cell of the tape hold exactly one of a finite number of *tape symbols*. The input string is left justified. Initially, the finite control is in the state q_0 and is scanning the leftmost symbol of a string of symbols which appear on the input tape. At any time the head, which is 2-way, can read a symbol on each track at the same time. As and when the symbols are read they are marked. Marking is done with help of a particular set called *Marker set*, denoted by M. There is a set of poset relations, $\mathcal{P}$ where each poset is defined on the set M. This poset relation helps to replace the markers when the necessity arises. The idea of marking and replacing one marker with a marker with higher affinity is borrowed from peptide computing where one antibody binding to a peptide sequence can be replaced by an antibody with a higher affinity [3, 5].

The formal definition of the system is as follows:

Definition 2. *A Rewriting Binding-Blocking Automaton (RBBA) is a construct*

$$\Gamma \;=\; (Q, \Sigma, V, \delta, M, \mathcal{R}, \mathcal{P}, q_0, F)$$

where Q is the finite set of states and $q_0 \in Q$ is the start state; Σ is the finite set of tape alphabet; $V \subseteq \Sigma$ is a finite set of symbols called input alphabet; δ is the transition function from $Q \times \binom{\Sigma}{V} \longrightarrow 2^{Q \times \{L,R\}}$; $M \subseteq V$ is called the set of markers; $\mathcal{R}$ is the set of posets over M called as affinity set (i.e, each $R \in \mathcal{R}$ is a subset of $M \times M$); $\mathcal{P}$ is defined by, $\mathcal{P} : Q \longrightarrow \mathcal{R}$ called as state-affinity function; and, $F \subseteq Q$ where F is the set of accepting states.

The symbols which are read by the head are called *marked* symbols. Suppose a sequence of symbol $x \;=\; a_1 a_2 \cdots a_n,\ a_i \in V,\ 1 \leq i \leq n$ is read, then it is marked

and denoted by $\begin{smallmatrix} a_1 & a_2 & \cdots & a_n \\ A_1 & A_2 & \cdots & A_n \end{smallmatrix}$. All other symbols not read by the head are denoted by $\begin{smallmatrix} a \\ - \end{smallmatrix}$. If no symbol is read or blocked in a string x, then we denote it by $\begin{smallmatrix} x \\ - \end{smallmatrix}$. Likewise, if all the symbols have been read by the system, then it is denoted by $\begin{smallmatrix} x \\ M \end{smallmatrix}$. If the string x contains both read and unread symbols then it is denoted by $\begin{smallmatrix} x \\ V \end{smallmatrix}$.

The initial configuration of the tape is,

$$\begin{array}{l} q_0\ a_1\ a_2 \cdots a_n\ \not b\ \not b \cdots \\ \uparrow\ -\ -\ \cdots -\ -\ -\ \cdots \end{array}$$

where $a_1a_2 \cdots a_n\ \not b\ \not b \cdots$, $a_i \in V$ and $\not b$ denotes blank cells is the input string on the tape of the system.

The instantaneous description of the system is as follows:

$$\begin{array}{l} a_1\ a_2\ \cdots a_{i-1}\ q\ a_i\ a_{i+1}\ \cdots a_n\ \not b \quad \not b \quad \cdots \not b\ \cdots \\ A_1\ A_2 \cdots A_{i-1} \uparrow A_i\ A_{i+1} \cdots A_n\ A_{n+1}\ A_{n+2} \cdots - \cdots \end{array}$$

is an ID for $\mathcal{P}$ and $a_i, A_i \in \Sigma$ where q is the state of the system, head of the system is at the i^{th} position, $a_1a_2 \cdots a_n$ is the input string on the tape of the system.

We say the system makes the move,

$$\begin{array}{l} a_1\ a_2\ \cdots a_{i-1}\ p\ a_i\ a_{i+1}\ \cdots a_j\ a_{j+1}\ \cdots a_n\ \not b \quad \not b \quad \cdots \cdots \\ A_1\ A_2 \cdots A_{i-1} \uparrow C_i\ C_{i+1} \cdots C_j\ C_{j+1} \cdots C_n\ C_{n+1}\ C_{n+2} \cdots \cdots \end{array}$$

$$\vdash$$

$$\begin{array}{l} a_1\ a_2\ \cdots a_{i-1}\ a_i\ q\ a_{i+1}\ \cdots a_j\ a_{j+1}\ \cdots a_n\ \not b \quad \not b \quad \cdots \cdots \\ A_1\ A_2 \cdots A_{i-1}\ B_i \uparrow C_{i+1} \cdots C_j\ C_{j+1} \cdots C_n\ C_{n+1}\ C_{n+2} \cdots \cdots \end{array}$$

if $\delta(p, \begin{smallmatrix} a_i \\ C_i \end{smallmatrix})$ contains (q, R) where $p, q \in Q, a_i, B_i, A_i \in V, C_i \in V \cup \{-\}, (B_i, C_i) \in \mathcal{P}(q)$ and $B_i \in max(\mathcal{P}(q))$.

or

$$\begin{array}{l} a_1\ a_2\ \cdots a_{i-1}\ p\ a_i\ a_{i+1}\ \cdots a_j\ a_{j+1}\ \cdots a_n\ \not b \quad \not b \quad \cdots \cdots \\ A_1\ A_2 \cdots A_{i-1} \uparrow C_i\ C_{i+1} \cdots C_j\ C_{j+1} \cdots C_n\ C_{n+1}\ C_{n+2} \cdots \cdots \end{array}$$

$$\vdash$$

$$\begin{array}{l} a_1\ a_2\ \cdots q\ a_{i-1}\ a_i\ a_{i+1}\ \cdots a_j\ a_{j+1}\ \cdots a_n\ \not b \quad \not b \quad \cdots \cdots \\ A_1\ A_2 \cdots \uparrow A_{i-1}\ B_i\ C_{i+1} \cdots C_j\ C_{j+1} \cdots C_n\ C_{n+1}\ C_{n+2} \cdots \cdots \end{array}$$

if $\delta(p, \begin{smallmatrix} a_i \\ C_i \end{smallmatrix})$ contains (q, L) where $p, q \in Q, a_i, B_i, A_i \in V, C_i \in V \cup \{-\}, (B_i, C_i) \in \mathcal{P}(q)$ and $B_i \in max(\mathcal{P}(q))$.

The language acceptance of the PA system $\mathcal{P}$ is defined as

$$L(\mathcal{P}) \ = \ \left\{ w \in V^* \mid \begin{array}{l} q_0\ w\ \not b\ \not b \cdots \\ \uparrow\ - - - \cdots \end{array} \vdash^* \begin{array}{l} w\ q_f\ \not b\ \not b\ \cdots \\ V \uparrow\ A_1\ A_2 \cdots \end{array}, \text{ for } q_f \in F \right\}.$$

Theorem 6. *For any Turing machine TM there is an equivalent $RBBA$ system which accepts the same language as TM.*

Proof. Let $TM = (Q, V, \Sigma, \delta, q_0, F)$ be a Turing machine. We construct a $RBBA$, $\Gamma = (Q', \Sigma', V', \delta', M, \mathcal{R}, \mathcal{P}, q_0', F')$ as follows:

1. $Q' = Q \times V \times Q \cup \{q_0', [q_0', \lambda, q_0]\} \cup F'$, $V' = V$, $\Sigma' = \Sigma \cup \{-\}$, $F' = \{p_f', p_f'' \mid p \in F\}$, $M = V'$,

We construct the affinity set $\mathcal{R}$ as follows:

2. for every $R \in \mathcal{R}$,
   - there exists $[q, A, p] \in Q'$ such that $R = \mathcal{P}([q, A, p])$,
   - $R' = \{(B, A) \mid (p, B, Mov) \in \delta(q, A)\}$ where $Mov \in \{L, R\}$,
   - $R = R' \cup \{(B, -), (A, -) \mid (B, A) \in R'\}$

The transition function δ' is defined as follows:

3. $([q_0', \lambda, q_0], R) \in \delta'(q_0', \lambda)$,
4. $([q, a, p], L/R) \in \delta'([r, b, q]], \begin{smallmatrix} a \\ \_ \end{smallmatrix})$ if $(p, A, L/R) \in \delta(q, a)$,
5. $([q, A, p], L/R) \in \delta'([r, C, q], \begin{smallmatrix} d \\ A \end{smallmatrix})$ if $(p, B, L/R) \in \delta(q, A)$, $d \in V$,
6. For $(p, B, L/R) \in \delta(q, A)$ with $p \in F$, $(p_f, L/R) \in \delta'([q, A], \begin{smallmatrix} d \\ A \end{smallmatrix})$,
7. $(p_f', L) \in \delta'(p_f, \begin{smallmatrix} b/ \\ A \end{smallmatrix})$,
8. $(p_f', L) \in \delta'(p_f', \begin{smallmatrix} b/ \\ A \end{smallmatrix})$,
9. $(p_f'', R) \in \delta'(p_f, \begin{smallmatrix} d \\ A \end{smallmatrix})$, $d \in V$,
10. $(p_f'', R) \in \delta'(p_f'', \begin{smallmatrix} d \\ A \end{smallmatrix})$, $d \in V$,

The states of the constructed system is taken as 3-tuples $[q, a, p]$. The reason for taking 3-tuples is the following: First it has to be noted that the rewriting technique of Turing machine can be simulated in rewriting binding-blocking automata only by defining affinity relation with respect to each states and symbols read. This affinity relation will make sure that the old symbol read is rewritten by some new symbol. So the crucial part of the construction of rewriting binding-blocking automata lies in defining the affinity relation. Each of the states of the form $[q, a, p]$ denotes the following,

- The system is in the state p,
- The system reached p by reading the symbol a when in the state q.

Now, if suppose the system rewrites the symbol a as A, then affinity relation for the state $[q, a, p]$ includes the element (A, a). Hence the rewriting binding-blocking automata rewrites the symbol a as A according to its affinity relation. The movement of the head of the rewriting binding-blocking automata is just same as the movement of the Turing machine.

In the above construction, we shall make it clear that when the symbols are of the form $\begin{smallmatrix} a \\ \_ \end{smallmatrix}$ then the system reads the top symbol which is a, otherwise, if it is of

the form $\begin{smallmatrix} a \\ A \end{smallmatrix}$, then the system reads only the bottom symbol A. This is to make sure that the system does not read any symbol which has been rewritten.

The other important issue in the construction is that some Turing machines can accept the input string even if it has not read the full input string. In this case we use a new state p_f which positions the head at the end of the input string w.

Finally when TM reaches a final state p, Γ reaches either of the states, p'_f or p''_f, and moves its head to the end of the input.

Hence any string accepted by the TM is accepted by Γ and no other string is accepted by Γ. □

5 Conclusion

We proposed two new automaton models – string binding-blocking automaton and rewriting binding-blocking automaton. Both these models use the idea of marking some symbols, in the former model it is used to postpone reading of those marked symbols and in the latter model it is used for storing some information. Both these models are more powerful than that of binding-blocking automata. We strongly feel that the acceptance power of string binding-blocking automata working in leftmost transition do not exceed the generative power of a random-context grammar. In the case of rewriting binding-blocking automaton, we generalized the blocking of symbols with a set of markers and imparted a state-affinity relation on the set of states. We proved that this model is universally complete.

Both the models proposed here used ideas from peptide computing where computing occurs through binding, removing and replacing of antibodies. The main motivation for this study was to use some of the ideas of peptide computing in classical automata theory and to study their acceptance power. In peptide computing, the processing is completely random (for example, antibody can bind with a substring present anywhere in a sequence) whereas, in classical automata the processing takes place sequentially. This study was to show how the system behaves if we make the processing sequential together with the use of abstract ideas from peptide computing.

References

1. M. S. Balan: String binding-blocking automata. In *Genetic and Evolutionary Computation Conference, LNCS 2723*. 425–426, 2003.
2. M. S. Balan, H. Jürgensen: Peptide computing: Universality and theoretical model. In *Unconventional Computation*, LNCS 4135. 57–71. Springer-Verlag, 2006.
3. M. S. Balan, H. Jürgensen: On the universality of peptide computing. *Natural Computing* (2007). In print.
4. M. S. Balan, K. Krithivasan. Binding-blocking automata. Communicated, 2007.
5. M. S. Balan, K. Krithivasan, Y. Sivasubramanyam: Peptide computing: Universality and computing. In N. Jonoska, N. Seeman (editors): *Proceedings of Seventh International Conference on DNA based Computers, LNCS 2340*. 290–299, 2002.
6. J. Dassow, G. Păun: *Regulated Rewriting in Formal Language Theory*. Springer, Berlin, 1989.
7. H. Hug, R. Schuler: Strategies for the developement of a peptide computer. *Bioinformatics* **17** (2001), 364–368.

Computation by annotation: modelling epigenetic regulation.*

Franziska Biegler[1], Mark Daley[1,2] and M. Elizabeth O. Locke[1]
[1]Department of Computer Science
[2]Department of Biology
University of Western Ontario
London, ON N6A 5B7, Canada

Abstract

We present a formal model inspired by the epigenetic process of gene annotation via histone modification. In particular, we study the generative capacity of a system in which annotations on a set of strings control which substrings are ultimately produced by the system and in which only the annotations, and not the strings themselves, may be rewritten. On a biological level this represents a first attempt to better understand the computational limits of this form of epigenetic regulation. We introduce two different derivation modes for our formal system and show that these systems are actually quite weak. The weaker of the derivation modes is directly capable only of generating a subset of the regular languages while the more powerful derivation mode is also only capable of generating all regular languages modulo a begin- and an end-marker.

1 Motivation and Biological Background

The DNA of a cell encodes information by joining four different phosphonucleotides: adenine, cytosine, guanine and thymine, commonly labelled A,C,G and T, respectively, into a long linear sequence. Genes are just one of many types of information stored in DNA and encode proteins which are constructed through the process of gene expression. In this process, a sequence of nucleotides are translated into a sequence of amino acids which then fold into a finished, biologically active, protein.

The regulation of gene expression is very important, as every protein must be expressed at the proper level to maintain a healthy cell state. Many cancers and disease states are associated with abberant regulation of gene expression, particularity of genes involved in the cell cycle (for an extensive review, see [15]). Along with the genes themselves, DNA also encodes information about both gene expression (through promoters and termination sequences), and gene regulation (through enhancers, silencers and other control regions).

Regulation is a very complex process involving many components, of which the histone proteins are a significant example. The structure of these proteins is like a

*This research was supported by funds from the Natural Sciences and Engineering Research Council of Canada and the SHARCNET Research Chairs program.

small spool which can have DNA wrapped about two and a half times around its core. Many succesive histone proteins will associate along each DNA string and the resulting structure, called chromatin, looks like beads on a string, where the DNA 'string' wraps around successive histone 'beads' (for a review, see [9]).

The histone proteins have tails (C-terminal and N-terminal ends of the protein) which stick out from the central core or spool region. The tails can be modified through biochemical processes, e.g., acetylation (and deacetylation), methylation and phosporylation, and these modifications can be thought of as flags which can be added or removed from the histone tails. Different combinations of flags on particular histones may change the regulation pattern of genes in that region [1] either by inducing structural changes or by recruiting proteins to the region which perform various actions (for reviews see [8] and [3]). For example, when histones are methylated, the histones, and in turn the DNA, are closer together and form highly condensed structures, see [13], whereas histone regions that are acetylated are generally highly expressed and thought to be much less condensed, as described in [4] and reviewed in [5]. In either case, the histone flags are forming a so-called 'epigentic code' which can be read by the cellular machinery, and result in various actions on the DNA and histones [1] ultimately affecting the regulation of genes in that region.

Abstractly, these flags are making comments on how genes in this region of chromatin (DNA and its associated histones) should be regulated. This parallels the idea of annotation, where a writer can make a comment on a string of text by writing something in the margin nearby, or directly linking a comment to a certain part of the text. A more modern example is that of markup languages, where tags are used to make formating and other comments on the underlying string using HTML and XML. Formalizations of these use the nested and hierarchical nature of strict markup languages to create tree grammars, as described in [12], but these are not flexible enough to model the annotations made on DNA strings.

In this paper we propose an extension of the formal model proposed in [11], inspired by histone annotation, which uses annotations to indicate comments on an underlying string, in the hopes that it will aid the formal study of DNA based languages. With the study of epigenetic regulation gaining in popularity and significance, we are particularly interested in the effect that this type of in-place string annotation can have, in general, on the regulation of gene expression. We aim to make our formalization as abstract as possible, in order to incorporate other types of regulatory elements that may be discovered in the future.

2 Preliminaries and notation

Let Σ be a finite alphabet and Σ^* be the free monoid on Σ, where we denote the empty word by λ. Let $\mathcal{A}$ be a finite set of *annotation labels*. An *annotated word* is a pair (w, A) where $w \in \Sigma^*$ and A is a set of annotations of the form $(n_1, n_2, a) \in \mathbb{N}_{|w|} \times \mathbb{N}_{|w|} \times \mathcal{A}$, where $\mathbb{N}_{|w|} = \{0, 1, ..., |w|\}$, which intuitively means that the subword starting at the n_1-th letter of w and ending with the n_2-th letter of w (including the letters at positions n_1 and n_2) is annotated by the label a. By

abuse of notation, we denote the set of all annotated words on Σ^* as $\Sigma^*_{\mathcal{A}}$.

We say that w' is the subword of w at index k, denoted by $(w', k) \leq_i w$ if $w = w_1 w' w_2$ with $|w_1| = k - 1$. Intuitively w' starts at the k-th letter of w. For a word w, we denote by $w[k]$ the k-th letter of w.

A right-linear grammar is a 4-tuple $G = (N, T, P, S)$, where N and T are finite alphabets of nonterminal and terminal symbols, $S \in N$ is the initial nonterminal and P is a set of productions of the form $A \to aB$, with $a \in T$ and $A, B \in N$ or $A \to a$ with $A \in N$, $a \in T \cup \{\lambda\}$. Right-linear grammars generate exactly the family of regular languages, which we denote by $\mathcal{L}(\mathrm{REG})$, see [7, 14] for more details. A (non-deterministic) finite automaton is a construct $A = (Q, \Sigma, \delta, Q_0, F)$, where Q is a finite set of states, Σ is a finite alphabet, $Q_0 \subseteq Q$ and $F \subseteq Q$ are finite sets of initial and final states, respectively, and $\delta \subseteq Q \times (\Sigma \cup \{\lambda\}) \times Q$ is the (non-deterministic) transition function. Finite automata accept also precisely the family of regular languages.

Below we define which annotations of a word w we associate with a subword w' of w.

Definition 2.1. *Let (w, A) be an annotated word from $\Sigma^*_{\mathcal{A}}$ let $(w', k) \leq_i w$, for some $1 \leq k \leq |w| - |w'| + 1$, and let $l = k + |w'| - 1$.*

We say (w', A') is a weakly annotated subword *of (w, A), denoted by*

$$(w', A') \leq^{\mathbf{W}}_{\mathrm{ann}} (w, A),$$

if and only if $A' = \{(n_1, n_2, a) \in A \mid k \leq n_1 \leq l \text{ or } k \leq n_2 \leq l \text{ or } n_1 < k \text{ and } n_2 > l\}$.

The set of all weakly annotated subwords of (w, A) is denoted by $\mathrm{ann}^{\mathbf{W}}(w, A)$.

Intuitively a subword w' of an annotated word w is weakly annotated by all those annotations of w which "overlap" with w'.

We note that one could also define strongly annotated subwords, which require that a particular symbol annotate the subword exactly, and, similiarly, infix annotated subwords. This is the subject of future research.

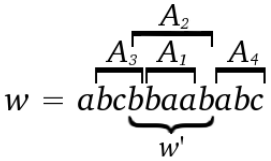
We also define a projection function that returns, for a given (indexed) subword of a word, the set of annotation labels that this subword is annotated with.

Definition 2.2. *Let (w, A) be an annotated word from $\Sigma^*_{\mathcal{A}}$, let $(w', k) \leq w$ and let $(w', A_w) \leq^{\mathbf{W}}_{ann} (w, A)$. We define*

$$\mathrm{lab}^{\mathbf{W}}(w', k) = \{a \mid (n_1, n_2, a) \in A_w\}.$$

Example 2.1. *Consider the following annotated word*

$$(w, A) = (abcbbaabab c, \{(5, 7, A_1), (4, 8, A_2), (2, 4, A_3), (9, 11, A_4)\}.$$



Obviously $(w', 4) \leq_i w$ *with* $w' = bbaab$, $(w', A) \leq^{\mathbf{w}}_{\text{ann}} (w, A)$ *and, thus* $\text{lab}^{\mathbf{w}}(w', 4) = \{A_1, A_2, A_3\}$.

We now describe a system which annotates a set of strings according to fixed annotation rules and then 'expresses' only those strings which are both in an expressible format and correctly annotated. This is intended as a high-level model of epigenetic regulation. The base set of strings represents a collection of chromosomes, each containing genes as subwords (marked, like real genes, with a 'start symbol', \$, and a 'stop symbol', #). The annotation rules represent the actions of histone annotation and the resulting set of 'expressed' strings represents the product of expressed genes. We note that we have deliberately excluded any other form of regulation from this model as our intent is to study solely the regulatory power of gene annotation. Formal models of traditional gene expression regulation have been studied extensively in the literature, see, e.g., [2, 10, 6].

Definition 2.3. *An* annotation system *is a 5-tuple*

$$G = (\Sigma, (\$, \#), \mathcal{A}, E, P)$$

where Σ *is a finite alphabet,* $(\$, \#)$ *is a pair of start- and stop-symbols,* $\mathcal{A}$ *is a finite set of annotation labels,* $E \subseteq \mathcal{A}$ *is a finite set of expressible annotation labels and* $P \subseteq \Sigma^* \times 2^{\mathcal{A}} \times \mathcal{A}$ *is a finite set of annotation rules.*

The start- and stop-symbols can be thought of as the formal equivalent of start and stop codons, and related regulatory elements. This definition can, of course, be generalized to sets of start and stop symbols, but this generalization does not effect the generative capacity so we consider these symbols to be unique in the following.

We now define the derivation relation. Note that only the annotations on a string change during a derivation; the underlying word always remains the same. There are two different modes, which differ only in whether additional annotations (which do not appear in the rules) are allowed to be present during a rule application.

Definition 2.4. *Let* $G = (\Sigma, (\$, \#), \mathcal{A}, E, P)$ *be an annotation system and let* $(w, A), (w, A') \in \Sigma^*_{\mathcal{A}}$. *Then* (w, A) *directly derives* (w, A') *in*

- subset mode *(shortly* s*-mode), denoted by* $(w, A) \Rightarrow_{\text{s}} (w, A')$, *if and only if there exists a word* v *and a number* k, *with* $(v, k) \leq_i w$ *and* $(v, B) \in \text{ann}^{\mathbf{w}}(w, A)$, *and there exists a rule* $(v, B', a) \in P$ *with* $B' \subseteq B$ *and* $A' = A \cup \{(k, k + |v| - 1, a)\}$.
- equality mode *(shortly* e*-mode), denoted by* $(w, A) \Rightarrow_{\text{e}} (w, A')$, *if and only if there exists a word* v *and a number* k, *with* $(v, k) \leq_i w$ *and* $(v, B) \in \text{ann}^{\mathbf{w}}(w, A)$, *and there exists a rule* $(v, B, a) \in P$ *and* $A' = A \cup \{(k, k + |v| - 1, a)\}$.

We allow for the possibility of two modes in order to allow us to capture the underlying biology in the greatest generality. There are many types of processes regulated by histone annotation with some operating in a subset-like mode while others operate in a strict equality mode. We also note that the equality mode allows for an implicit form of "forbidding" annotations.

As usual, we use $\Rightarrow_{\mathrm{e}}^*$ and $\Rightarrow_{\mathrm{s}}^*$ to denote the reflexive and transitive closures of $\Rightarrow_{\mathrm{e}}$ and $\Rightarrow_{\mathrm{s}}$.

Finally, recalling that our goal is to model regulation of gene *expression*, we define the languages generated (expressed) by an annotation system. Intuitively, a word w' will be generated by an annotation system if it is a subword of an annotated word (w, A) which begins with the start symbol \$, ends with end symbol # and the entirety of w' can, in the given mode, be annotated by one of the expressible annotation labels.

Definition 2.5. *Let $G = (\Sigma, (\$, \#), \mathcal{A}, E, P)$ be an annotation system. For an annotated word $(w, A) \in (\Sigma \cup \{\$, \#\})^*_{\mathcal{A}}$, the set of words expressed by (w, A) is defined as*

$$\mathrm{x}(w, A) = \{w' \mid \exists k \text{ with } w' \in \$\Sigma^*\#, (w', k) \leq_i w \text{ and } \exists \alpha \in E \text{ such that } \forall i \text{ with } k \leq i < k + |w'| \text{ we have } \alpha \in \mathrm{lab}^{\mathbf{W}}(w, i)\}$$

For a word $w \in (\Sigma \cup \{\$, \#\})^*$, the set of expressed words generated from w by G in z-mode, $z \in \{\mathrm{e}, \mathrm{s}\}$, is defined as

$$\mathrm{x}_z(w) = \{w' \mid (w, \emptyset) \Rightarrow_z^* (w, A) \text{ and } w' \in \mathrm{x}(w, A)\}.$$

The language generated by G in z-mode is defined as

$$L_z(G) = \bigcup_{w \in (\Sigma \cup \{\$, \#\})^*} \mathrm{x}_z(w).$$

By $\mathcal{L}_z(\mathrm{ANN})$, we denote the family of all languages that can be generated by an annotation system in z-mode, for $z \in \{\mathrm{e}, \mathrm{s}\}$.

Note that all words in $L(G)$ have to start with \$ and end with #.

The following result is obvious from the definitions of the two modes.

Proposition 2.1. *For all annotation systems G, we have $L_{\mathrm{e}}(G) \subseteq L_{\mathrm{S}}(G)$.*

The following example should help to clarify the above definitions.

Example 2.2. *$G = (\Sigma, (\$, \#), \mathcal{A}, E, P)$ with $\Sigma = \{a, b\}$, $\mathcal{A} = \{A_1, A_2, A_3, \star\}$, $E = \{\star\}$, and P defined by the following rules:*

$$(\$ab, \emptyset, A_1), \quad (abab, \{A_1\}, A_2),$$
$$(abab, \{A_2\}, A_3), \quad (abab, \{A_3\}, A_1),$$
$$(ab\#, \{A_3\}, \star).$$

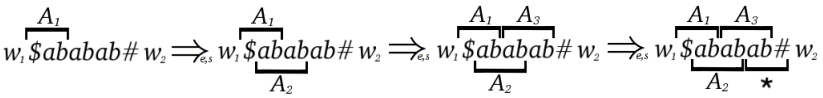
Also, for $X \subseteq \{A_1, A_2, A_3\}$, the following rules are in P

$$(ab, \{\star\} \cup X, \star), \quad (ba, \{\star\} \cup X, \star),$$
$$(\$a, \{\star\} \cup X, \star).$$

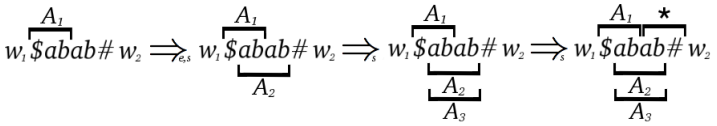
The latter set of rules annotates all symbols in subwords of the form $\$w\#$, every letter of which is already annotated by some annotation label, by the $\star$-symbol. This

enables us to express these words. The example derivations found below do not apply these rules, for reason of better readability. The derivations stop once every letter of a subword $w# has been annotated.

From the following partial derivations, we see that both modes can generate $\$(ab)^3\#$.



However, we see that, when annotating $\$(ab)^2\#$*, in e-mode the derivation is blocked after* A_2 *is added because the subword abab is not annotated exactly by* $\{A_2\}$*, but by* $\{A_1, A_2\}$.



So the system G *generates* $L_{\mathrm{e}}(G) = \$((ab)^3)^*\#$ *and* $L_{\mathrm{S}}(G) = \$(ab)^*\#$.

3 Computational power of annotation

In this section we investigate the generative capacity of annotation systems in both modes. As our main results, we show that annotation systems in e-mode generate exactly the regular languages modulo a begin- and an end-marker, while annotation systems in s-mode generate only a proper subset of the e-mode annotation languages. We conjecture that the two families of annotation languges are separated by so-called *count-loop* languages, which are defined in this section.

Our first main result shows that annotation systems in e-mode generate only regular languages.

Theorem 3.1. $\mathcal{L}_{\mathrm{e}}(\mathrm{ANN}) \subseteq \mathcal{L}(\mathrm{REG})$.

Proof. Let $G = (\Sigma, (\$, \#), \mathcal{A}, E, P)$ be an annotation system. Let $m = \max\{n \mid (w, X, Y) \in P, n = |w|\}$ We construct a (non-deterministiitc) finite automaton $A = (Q, \Sigma', \delta, Q_0, \{f\})$ as follows:

$$Q = \{(X, a_1 \cdots a_n, (A_0, \ldots, A_n), (B_0, \ldots, B_n), f) \mid \begin{array}{l} X \in E, 0 \leq n \leq m, a_j \in \Sigma, \\ 2 \leq j < n, a_1 \in \Sigma \cup \{\$\}, \\ a_n \in \Sigma \cup \{\#\}, A_i \subseteq \mathcal{A} \times 2^{\mathcal{A}}, \\ B_i \subseteq \mathcal{A}, 1 \leq i \leq n\}; \end{array}$$

$\Sigma' = \Sigma \cup \{\$, \#\}$, $Q_0 = \{(X, \lambda, (), ()) \mid X \in E\}$ and δ is defined further down.

The idea of the construction is to traverse the word from left to right with a buffer of up to m letters by the automaton and add annotations to the current buffer. In the annotation system, annotations are not necessarily added in a left-to-right fashion. In order to be able to still simulate every possible combination of annotations, we are allowed to assume annotations for letters in the buffer which will be added once the buffer has been moved further to the right. The buffer can

only be moved to the right if the letters "leaving" the buffer on the left end are annotated by the necessary annotation (as defined by the expression set) and no annotations are assumed for these letters anymore.

The states consist of a member of the expression set, a string of length at most m, which is the maximal length of a substring to be annotated and for each letter of this string we store two sets of annotations. For the i-th letter, the set A_i consists of pairs $(Y, \mathcal{Y})$, meaning that the i-th letter is annotated by Y and that this annotation (transitively) made use of assuming the annotations in $\mathcal{Y}$ while B_i consists of the "assumed" annotations of letter i, which might be necessary in order to get the annotations in A_i or some of the neighbouring letters (which can be up to at most $m-1$ characters away).

If we have, e.g. a pair $(X, \{U, Y, Z\}) \in A_1$, then this means that the annotation X of letter number i in the current buffer was obtained (transitively) by assuming the annotations U, Y and Z for some letter (not necessarily letter i). This information has to be stored to prevent X from being used to annotate anything by U, Y or Z.

We define a projection function $\pi_1 : 2^{(\mathcal{A}\times 2^{\mathcal{A}})} \to 2^{\mathcal{A}}$ with $\pi(A) = \{Y \mid (Y, \mathcal{Y}) \in A\}$, for every $A \subseteq \mathcal{A} \times 2^{\mathcal{A}}$, which is used to extract the first components of the pairs in the A_i.

We also define, for each annotation label $D \in \mathcal{A}$, a projection function $\pi_D : 2^{(\mathcal{A}\times 2^{\mathcal{A}})} \to 2^{\mathcal{A}}$ with $\pi_D(A) = \{Y \mid (Y, \mathcal{Y}) \in A, D \notin \mathcal{Y}\}$, which is used to extract only those first components of the pairs in the A_i that did *not* make use of assuming D.

There is a set of initial states, one for each expression set. Thus, the automaton can be thought of as a finite union of automata, one for each expression set.

In the transitions, moving the buffer and annotating parts of the buffer is never done simultaneously.

The following set of transitions consists of transitions leading out of the intitial states. In each initial state only \$ can be read as only \$ can start a word. For $X \in E$, let

$$\delta((X, \lambda, (), ()), \$) = (X, \$, (\emptyset), (\emptyset)). \tag{1}$$

The following set of transitions lead into the final state. They are only possible if the last letter is #, there are no more assumed annotations and the expression set associated with the current state is a subset of the annotation sets for all letters. For $X \in E$, $n \geq 0$, $a_1 \in \Sigma \cup \{\$\}$, $a_2, \ldots, a_n \in \Sigma$ and $A_1, \ldots, A_n, A_\# \subseteq \mathcal{A} \times 2^{\mathcal{A}}$ with $X \in \pi_1(A_1), \ldots, \pi_1(A_n), \pi_1(A_\#)$ let

$$f \in \delta((X, a_1 \cdots a_n\#, (A_1, \ldots, A_n, A_\#), \emptyset^{n+1}), \lambda). \tag{2}$$

The next set of transitions reads a new symbol into the buffer. These transitions are only possible if the current number of symbols in the buffer is less than the maximum m. The annotation set and assumed annotation set for the new letter are initialized by the empty set. For $X \in E$, $0 \leq n < m$, $a_1 \in \Sigma \cup \{\$\}$, $a_2, \ldots, a_n \in \Sigma$, $a \in \Sigma \cup \{\#\}$, $A_1, \ldots, A_n \subseteq \mathcal{A} \times 2^{\mathcal{A}}$, $B_1, \ldots, B_n \subseteq \mathcal{A}$, let

$$\begin{aligned}\delta((X, a_1 \cdots a_n, (A_1, \ldots, A_n), (B_1, \ldots, B_n)), a) \\ = (X, a_1 \cdots a_n a, (A_1, \ldots, A_n, \emptyset), (B_1, \ldots, B_n, \emptyset)).\end{aligned} \tag{3}$$

The following set of transitions allow us to remove symbols at the beginning of the buffer. We can only remove a prefix of the current string in the buffer if the expression set associated with the current state is a subset of the annotation sets of all the letters in the prefix to be removed and also the sets of assumed annotations for these letters have to be empty. For $X \in E$, $0 < j \leq n \leq m$, $a_1 \in \Sigma \cup \{\$\}$, $a_2, \ldots, a_{n-1} \in \Sigma$, $a_n \in \Sigma \cup \{\#\}$, $A_1, \ldots, A_n \subseteq \mathcal{A} \times 2^{\mathcal{A}}$, $B_1, \ldots, B_n \subseteq \mathcal{A}$, with $X \in \pi_1(A_1), \ldots, \pi_1(A_j)$, $\emptyset = B_1 = \cdots = B_j$ let

$$(X, a_{j+1} \cdots a_n, (A_{j+1}, \ldots, A_n), (B_{j+1}, \ldots, B_n)) \\ \in \delta((X, a_1 \cdots a_n, (A_1, \ldots, A_n), (B_1, \ldots, B_n), \lambda). \quad (4)$$

The following set of transitions allow us to add assumed annotations at any time. For $X \in E$, $0 < i \leq n \leq m$, $a_1 \in \Sigma \cup \{\$\}$, $a_2, \ldots, a_{n-1} \in \Sigma$, $a_n \in \Sigma \cup \{\#\}$, $A_1, \ldots, A_n \subseteq \mathcal{A} \times 2^{\mathcal{A}}$, $B_1, \ldots, B_n \subseteq \mathcal{A}$, $B'_i = B_i \cup \{D\}$, for some $D \in \mathcal{A}$, such that $D \notin \pi(A_i)$ let

$$(X, a_1 \cdots a_n, (A_1, \ldots, A_n), (B_1, \ldots, B_{i-1}, B'_i, B_{i+1}, \ldots, B_n) \\ \in \delta((X, a_1 \cdots a_n, (A_1, \ldots, A_n), (B_1, \ldots, B_n)), \lambda) \quad (5)$$

The following set of transitions essentially simulate the rules of the annotation system. An annotation can be added to a substring of the current buffer, resulting in adding it to all the letters of this substring, if the substring is already annotated by the annotations necessary for the simulated rule. These needed annotations can be present either in the first components of the A_i sets or in the assumed B_i sets. Annotations in the A_i sets cannot be used to add a certain annotation if they were themselves generated with this annotation being assumed. If an annotation is added then this annotation is removed from the assumptions.

Let $X \in E$, $0 \leq i < j \leq n \leq m$, $a_1 \in \Sigma \cup \{\$\}$, $a_2, \ldots, a_{n-1} \in \Sigma$, $a_n \in \Sigma \cup \{\#\}$, $A_1, \ldots, A_n \subseteq \mathcal{A} \times 2^{\mathcal{A}}$, $B_1, \ldots, B_n \subseteq \mathcal{A}$ and $(a_i \cdots a_j, C, D) \in P$ with $C = \{C_1, \ldots, C_k\} \subseteq \mathcal{A}$, $0 \leq k \leq |\mathcal{A}|$ and let $D \in \mathcal{A}$. Furthermore we must have $\pi_1(A_l) = \pi_D(A_l)$ for $i \leq l \leq j$, ensuring that D was not used as an assumption and we let $C = \bigcup_{i \leq l \leq j}(\pi_D(A_1) \cup B_l)$, $A'_l = A_l \cup \{(D, C \setminus A_D)\}$, $B'_l = B_l \setminus \{D\}$, for $i \leq l \leq j$, let

$$(X, a_1 \cdots a_n, (A_1, \ldots, A_{i-1}, A'_i, \ldots, A'_j, A_{j+1}, \ldots, A_n), \\ (B_1, \ldots, B_{i-1}, B'_i, \ldots, B'_j, B_{j+1}, \ldots, B_n)) \\ \in \delta((X, a_1 \cdots a_n, (A_1, \ldots, A_n), (B_1, \ldots, B_n)), \lambda). \quad (6)$$

It is obvious that all words accepted by the automaton can also be expressed by the annotation system and, hence, every word in $L(A)$ is also in $L(G)$.

To show that $L(G) \subseteq L(A)$ we look at a derivation of G and look at the dependence structure of the annotations. Adding the annotations to the initially not annotated string induces a partial order. We number the annotations in such a way that an annotation never depends on an annotation with a higher number. Now when traversing the string from left to right by the automaton we keep adding new symbols to the buffer by transitions from (3) until we have a substring in the buffer

that has an annotation associated with it. If adding this annotation in the original derivation of the annotation system depended on other annotations being present which we have not reached yet, then we assume these annotations for the appropriate symbols. By doing that we can simulate every derivation of the annotation system by the automaton. □

We now show that all regular languages can be generated by an annotation system in e-mode, modulo a begin- and an end-marker.

Theorem 3.2. *Let $L \in \mathcal{L}(\mathrm{REG})$ and let $\$, \#$ be two symbols that do not appear in the alphabet of L. Then there exists an annotation system G, such that $L_{\mathrm{e}}(G) = \$L\#$.*

Proof. Let $G = (N, T, P, S)$ be a right-linear grammar. We construct an annotation system $G' = (T, (\$, \#), \mathcal{A}, E, P')$ as follows. We let $\mathcal{A} = \{X, X' \mid X \in N\} \cup \{\star\}$ and $E = \{\star\}$. $\mathcal{A}$ essentially mimics the nonterminals. The marked versions of the nonterminals are used to remove cycles of the from $A \to aA$ from the grammar, by replacing them with $A \to aA'$ and $A' \to aA$. The star is used only to label correct derivations in order to express them.

The intuitive idea behind the construction is to first annotate the \$ sign by the initial nonterminal and then to keep annotating symbol pairs xy by X if x was already annotated by Y and there exists a production $Y \to yX$. A left-to-right character of the annotations is enforced, as letters annotated by more than one symbol cannot be used for any future annotation (besides the $\star$-symbol), as we are working in e-mode.

We let $(\$, \emptyset, S) \in P'$, which acts as the "seed" of annotating a subword.

For $A \to yB \in P$ with $y \in T$, $A, B \in N$, $B \neq A$ and $x \in T \cup \{\$\}$, we let

$$(xy, \{A\}, B) \in P' \text{ and } (xy, \{A'\}, B) \in P'.$$

For $A \to yA \in P$ with $y \in T$, $A \in N$ and $x \in T \cup \{\$\}$, we let

$$(xy, \{A\}, A') \in P' \text{ and } (xy, \{A'\}, A) \in P'.$$

These rules simulate the non-terminating derivations of the grammar.

In addition, to simulate the terminating productions, for $A \to y \in P$ with $y \in T$, $A \in N$ and $x \in T \cup \{\$\}$, we let

$$(xy\#, \{A\}, \star) \in P' \text{ and } (xy\#, \{A'\}, \star) \in P'.$$

For $A \to \lambda \in P$ with $A \in N$ and $x \in T \cup \{\$\}$, we let

$$(x\#, \{A\}, \star) \in P' \text{ and } (x\#, \{A'\}, \star) \in P'.$$

It is easy to see that, whenever we have a subword $\$w\#$, that is fully annotated (i.e. every letter of it is annotated by some annotation symbol), then $w \in L(G)$.

The following rules are used to annotate everything in between a \$-symbol and a #-symbol by $\star$, if it is already annotated by something else. This allows us to then express the subword between \$ and #. The $\star$-annotation is added from right to left. For all $x \in T \cup \{\$\}$, $y \in T \cup \{\#\}$, we let $(xy, \{\star\}, \star) \in P'$.

It is straightforward that $L_{\mathrm{e}}(G) = \$L\#$. □

Next we show that the family of all languages generated in s-mode is a subset of the family of all e-mode languages. This inclusion is shown to be proper in the next result.

Theorem 3.3. $\mathcal{L}_{\mathrm{S}}(\mathrm{ANN}) \subseteq \mathcal{L}_{\mathrm{e}}(\mathrm{ANN})$.

Proof. Let $G = (\Sigma, (\$, \#), \mathcal{A}, E, P)$ be an annotation system. We construct an annotation system $G' = (\Sigma, (\$, \#), \mathcal{A}, E, P')$, such that $L_{\mathrm{e}}(G') = L_{\mathrm{S}}(G)$ as follows:

For $(w, X, Y) \in P$ with $w \in \{\$, \lambda\}\Sigma^*\{\#, \lambda\}$, $X \in 2^{\mathcal{A}}$, $Y \in \mathcal{A}$, let the set $\{(w, X \cup X', Y) \mid X' \in 2^{\mathcal{A}}\}$ be contained in P'.

It is obvious that $L_{\mathrm{e}}(G') = L_{\mathrm{S}}(G)$. □

Lemma 3.1. *There exists a language* $L \in \mathcal{L}_{\mathrm{e}}(\mathrm{ANN}) \setminus \mathcal{L}_{\mathrm{S}}(\mathrm{ANN})$.

Proof. Let $L = \$(bbb)^*\#$. As $(bbb)^*$ is a regular language, $L \in \mathcal{L}_{\mathrm{e}}(\mathrm{ANN})$. Now assume $L \in \mathcal{L}_{\mathrm{S}}(\mathrm{ANN})$ and let $G = (\Sigma, (\$, \#), \mathcal{A}, E, P)$ be an annotation system such that $L = L_{\mathrm{S}}(G)$. In order to express precisely the language L, the annotations somehow have to encode which b is a "first", "second" or "third" b of the recurring sequence bbb. Obviously there have to be rules of the form (b^n, X, Y) for some $n \geq 2$, $X \subseteq \mathcal{A}$, $Y \in \mathcal{A}$. However, applying a rule of this type to a long enough substring of b's, the annotation Y can slide. More precisely if $b^{n-1}b^m b^{n-1}$ is a subword of some word we are annotating, such that the central b^m is annotated by X already, than any subword of $b^{n-1}b^m b^{n-1}$ which is of length n can be annotated by Y. But this leads to Y "forgetting" which b's it is annotating, thus leading to generate words in $b^* \setminus (bbb)^*$, a contradiction. □

Very similar to the proof of Lemma 3.1, one can also proof that the language $L = ((ab)^3)^*$ which is generated in e-mode by the system of Example 2.2 cannot be generated by any annotation system in s-mode.

Definition 3.1. *A right-linear grammar* $G = (N, T, P, S)$ *is said to have a* count-loop *if there exist* $X_1, X_2, \ldots, X_n \in N$, *where* $X_i \neq X_j, \forall i \neq j$, $w \in T^+$ *and productions* $X_1 \to w_1X_2, X_2 \to w_2X_3, \ldots, X_n \to w_nX_1$ *with* $w_1w_2 \cdots w_n = w^k$ *for some* $w \in T^*$ *with* $|w| \geq 1$ *and* $k > 1$. *A regular language is called a* count-loop regular language *if every right-linear grammar generating it has a count-loop. The family of count-loop regular languages is denoted by* $\mathcal{L}(\mathrm{clREG})$.

We conjecture that the count-loop regular languages separate the families of e-mode and s-mode annotation languages. Proving this will be subject of future research.

4 Discussion

Inspired by recent advances in the understanding of epigenetic regulation through histone annotation, we have formulated a formal model of computation based on the annotation of strings and investigated its computational power. We offer two modes of operation for this model: a weak mode (s-mode) which allows annotations to be applied only if certain other annotations are already present, and a stronger mode

(e-mode) which allows annotations only if a precise set of other annotations, and no others, are present. We demonstrated that e-mode annotation systems are not capable of generating non-regular languages while any regular language can be generated modulo start and end markers on each word. As one would expect, the family of languages generatable in the weaker s-mode was shown to be strictly contained within the family of languages generated in e-mode; in particular we conjecture that the family of count-loop regular languages separates these two families.

We note with interest that the computational power of string annotation is, in general, quite weak and suggest that this may be of interest to biologists studying the superficially complex process of epigenetic regulation.

References

[1] V.G. Allfrey, R. Faulkner, and A.E. Mirsky. Acetylation and methylation of histones and their possible role in the regulation of rna synthesis. *PNAS*, 51:786–793, 1964.

[2] A. Arkin and H.H McAdams. Stochastic kinetic analysis of developmental pathway bifurcation in phage lambda-infected escherichia coli cells. *Genetics*, 149:1633–1648, 1998.

[3] S.L. Berger. Histone modifications in transcriptional regulation. *Current Opinion in Genetics & Development*, 12:142–148, 2002.

[4] James E. Brownell, Jianxin Zhou, Tamara Ranalli, Ryuji Kobayashi, Diane G. Edmondson, Sharon Y. Roth, and C. David Allis. Tetrahymena histone acetyltransferase a: A homolog to yeast gcn5p linking histone acetylation to gene activation. *Cell*, 84:843–851, 1996.

[5] A. Eberharter and P. Becker. Histone acetylation: a switch between repressive and permissive chromatin. *EMBO reports*, 3:224–229, 2002.

[6] Radek Erban, Ioannis G. Kevrekidis, David Adalsteinsson, and Timothy C. Elston. Gene regulatory networks: A coarse-grained, equation-free approach to multiscale computation. *J. Chem. Phys.*, 124:084106, 2006.

[7] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, MA, 1979.

[8] T. Jenuwein and C.D. Allis. Translating the histone code. *Science*, 293:1074–1080, 2001.

[9] Roger D. Kornberg and Yahli Lorch. Twenty-five years of the nucleosome, fundamental particle of the eukaryote chromosome. *Cell*, 98:285–294, 1999.

[10] Harri Lähdesmäki, Ilya Shmulevich, and Olli Yli-Harja. On learning gene regulatory networks under the boolean network model. *Machine Learning*, 52:147–167, 2003.

[11] M. Elizabeth O. Locke. Formal model of histone annotation on DNA strings. Technical Report 678, Department of Computer Science, University of Western Ontario, 2006.

[12] Makoto Murata, Dongwon Lee, Murali Mani, and Kohsuke Kawaguchi. Taxonomy of xml schema languages using formal language theory. *ACM Trans. Inter. Tech.*, 5(4):660–704, 2005.

[13] S. Rae, F.Eisenhaber, D. O'Carroll, B.D. Strahl, Z.W. Sun, M. Schmid, S. Opravil, K. Mechtler, C.P. Pontig, C.D. Allis, and T. Jenuwein. Regulation of chromatin structure by site-specific histone h3 methyltransferases. *Nature*, 406:593–599, 2000.

[14] A. Salomaa. *Formal Languages*. Academic Press, New York, 1973.

[15] K. Vermeulen, D.R. Van Bockstaele, and Z.N. Berneman. The cell cycle: a review of regulation, deregulation and theraputic targets in cancer. *Cell Proliferation*, 36:131–149, 2003.

Theoretical and computational properties of transpositions

Mark Daley[1,2,*], Ian McQuillan[2,†], James M. McQuillan[3]

[1] Department of Computer Science, Department of Biology
University of Western Ontario
London, ON N6A 5B7, Canada
daley@csd.uwo.ca

[2] Department of Computer Science
University of Saskatchewan
Saskatoon, SK, S7N 5A9, Canada
mcquillan@cs.usask.ca

[3] Department of Computer Science
Western Illinois University
Macomb, IL, 61455-1390, USA
jm-mcquillan@wiu.edu

Abstract

Transposable genetic elements are prevalent across many living organisms from bacteria to large mammals. Given the linear primary structure of genetic material, this process is natural to study from a theoretical perspective using formal language theory. We abstract the process of genetic transposition to operations on languages and study it combinatorially and computationally. It is shown that the power of such systems is large relative to the classic Chomsky Hierarchy. However, we are still able to algorithmically determine whether or not a string is a possible product of the iterated application of the operations.

1 Introduction

There are many different types of changes which can occur to a genome throughout evolution. These include mutations, and small insertions and deletions. In addition, there is a large class of DNA sequences which can move from one location to

*This work was supported by grants from NSERC & SHARCNET
†Supported by grants from NSERC and The University of Saskatchewan.

another within a genome. These are collectively known as *transposable elements* or *transpositions*. They are extremely important biologically, as they are estimated to occupy between $64\% - 73\%$ of corn [6]. Moreover, the dispersed repetitive fraction of the human genome is estimated to be 46% [1].

See [4] for a good survey of transpositions. There are two main classes of transposable elements. Class I are referred to as *retroelements*. These elements are produced by using reverse transcription (which allows DNA to be produced from RNA) to make copies of themselves. The new DNA then integrates at a new location of the genome. Intuitively, this class of transpositions operates similar to a *copy-and-paste* mechanism. Class II are known as DNA transposons. They operate with a *cut-and-paste* mechanism. Furthermore, many types of transposable elements have a target site preference, which will affect the contexts around the DNA in which the elements will be re-inserted.

These elements are quite natural to study theoretically using formal language theory, abstracted to operations on words and languages. In this paper, we attempt to lay the foundations for the study of this operation both mathematically and algorithmically. We then analyze some basic properties of the operation.

In section 2, we give some mathematical preliminaries necessary for the rest of the paper. In section 3, we give the definitions of different types of transpositions abstracted to operations on words and languages. Sections 4 and 5 investigate the computational power of applying both classes of transpositions iteratively. Section 6 investigates the properties of each type being applied a single time. This section is useful towards the algorithmic study of transpositions, which is investigated in section 7.

We were able to demonstrate that iterated type 1 transpositions produce arbitrary Recursively Enumerable languages from finite initial and transposition languages modulo right quotient with a regular language. In addition, we were able to closely characterize the non-iterated versions of the operations. Then we determined that we could algorithmically determine if a string is a possible product of either iterated type 1 or type 2 transpositions, or even both together at once, so long as we can determine membership in the initial and transposition language.

We hope that the study is useful for mathematical modelling, to improve our understanding of transpositions, and also towards bioinformatics.

2 Preliminaries

Let $\mathbb{N}$ be the set of positive integers and let $\mathbb{N}_0$ be the set of nonnegative integers.

We refer to [8] for language theory preliminaries. Let Σ be a finite alphabet. We denote, by Σ^* and Σ^+, the sets of all words and non-empty words, respectively, over Σ and the empty word by λ. A language L is any subset of Σ^*. Let $L, R \subseteq \Sigma^*$. We denote by $R^{-1}L = \{z \in \Sigma^* \mid yz \in L \text{ for some } y \in R\}$ and $LR^{-1} = \{z \in \Sigma^* \mid zy \in L \text{ for some } y \in R\}$.

Let $x \in \Sigma^*$. We let $|x|$ denote the length of x. For each $a \in \Sigma$, we let $|x|_a$ be the number of occurrences of a in x. If $y = (x_1, \ldots, x_n)$ is some n-tuple over Σ^*, then $|y| = |x_1| + \cdots + |x_n|$.

For $i \in \mathbb{N}$, let $x(i)$ be a_i if $x = a_1 \cdots a_i \cdots a_n$, $a_j \in \Sigma, 1 \leq j \leq n$, and undefined otherwise.

For $n \in \mathbb{N}_0$, let $\Sigma^n = \{w \in \Sigma^* \mid |w| = n\}$, $\Sigma^{\geq n} = \{w \in \Sigma^* \mid |w| \geq n\}$ and $\Sigma^{\leq n} = \{w \in \Sigma^* \mid |w| \leq n\}$.

Let $x, y \in \Sigma^*$. We say x is a *prefix* of y, written $x \leq_{\mathrm{p}} y$, if $y = xu$, for some $u \in \Sigma^*$. We say x is a *suffix* of y, written $x \leq_{\mathrm{s}} y$, if $y = ux$, for some $u \in \Sigma^*$. We say x is an *infix* of y, written $x \leq_{\mathrm{i}} y$, if $y = uxv$, for some $u, v \in \Sigma^*$.

For each word $x \in \Sigma^*$ with $\Sigma = \{a_1, \ldots, a_n\}$ using some fixed ordering, we associate its *Parikh vector*, $p_\Sigma(x)$ by $p_\Sigma(x) = (|x|_{a_1}, |x|_{a_2}, \ldots, |x|_{a_n})$. We extend p_Σ to languages in the natural way. We omit Σ if it is understood. Given Σ and Parikh vector v, the *Parikh inverse set* of v, $p^{-1}(v)$, is $p^{-1}(v) = \{x \in \Sigma^* \mid p(x) = v\}$. If $x \in \Sigma^*$ and $L \subseteq \Sigma^*$, then let $\mathrm{perm}(x) = \{y \in \Sigma^* \mid p_\Sigma(y) = p_\Sigma(x)\}$ and $\mathrm{perm}(L) = \bigcup_{x \in L} \mathrm{perm}(x)$. Note that $p^{-1}(p(x)) = perm(x)$ for all $x \in \Sigma^*$.

A set of vectors is called *semilinear* if it can be represented as a union of a finite number of sets of the form $\{v_0 + \sum_{i=1}^{m} \alpha_i v_i \mid \alpha_i \in \mathbb{N}_0 \text{ for } 1 \leq i \leq m\}$ where $v_i \in \mathbb{N}_0^n$ for $0 \leq i \leq m$. A language L is called *semilinear* if the set $p_\Sigma(L)$ is a semilinear set. Two languages L, L' are called *letter-equivalent* if and only if $p_\Sigma(L) = p_\Sigma(L')$. Thus, if two languages are letter equivalent, then one is semilinear if and only if the other one is also. It is known that a language L is semilinear if and only if it is letter-equivalent to a regular language [2].

A directed graph is a tuple $G = (V, E)$ in the usual way. For $v \in V$, we let $C(v)$ be the connected component of v viewed as a subgraph of G.

3 Transpositions

Following the biology of Class I and Class II transpositions, we define two types of formal transposition as follows. Let Σ be a finite alphabet. A 4-tuple $t = (x, u, y, z)$ is a *transposition* if $x, y \in \Sigma^*$, $u \in \Sigma^+$, and $z \in \{1, 2\}$. The transposition t is *type 1* if $z = 1$; it is *type 2* if $z = 2$.

The *transposition-concatenation map* $h^c : \Sigma^* \times \Sigma^+ \times \Sigma^* \times \{1, 2\} \to \Sigma^+$ is defined by $h^c(x, u, y, z) = xuy$, where $t = (x, u, y, z)$ is a transposition. Also, we define h_i^c to be the i^{th} coordinate of the transposition, for $1 \leq i \leq 4$, and $h^c(T) = \{h^c(t) \mid t \in T\}$. We also define $T^{\leq n} = \{t \in T \mid |h^c(T)| \leq n\}$.

A *transposition schema* is an ordered pair $\gamma = (\Sigma, T)$ where Σ is an alphabet and T is a set of transpositions.

Given a transposition schema $\gamma = (\Sigma, T)$ and $r, s \in \Sigma^*$, we say that r *is T-translatable to* s (or s *is T-translatable from* r), denoted $r \to_T s$ if at least one of the following two conditions are true:

1. there exists $(x, u, y, 1) \in T$ such that $r = \alpha u \beta = x'xyy'$ and $s = x'xuyy'$, for some $\alpha, \beta, x', y' \in \Sigma^*$,
2. there exists $(x, u, y, 2) \in T$ such that $r = \alpha u \beta,\ \alpha\beta = x'xyy',\ s = x'xuyy'$, for some $\alpha, \beta, x', y' \in \Sigma^*$.

We abbreviate $r \to_T s$ with $r \to s$ when T is understood. We say that $r \to_T^+ s$ if there exists $k \geq 0$ and $r_1, r_2, \ldots, r_k \in \Sigma^*$ such that $r \to_T r_1 \to_T \cdots \to_T r_k \to_T s$.

We say that $r \rightarrow_T^* s$ if $r \rightarrow_T^+ s$ or $r = s$.

Intuitively, $r \rightarrow_T s$ implies that applying some transposition in T to r can produce s. For a type 1 transposition $(x, u, y, 1)$ to be applied, u must appear in r, and then is pasted between sites x and y. This closely reflects the copy-and-paste functionality of class I transposable elements. With type 2, u gets cut out of r before it is pasted between x and y.

Let $\gamma = (\Sigma, T)$ be a transposition schema and let $L \subseteq \Sigma^*$. We define the *non-iterated transposition of* L to be $\gamma(L) = \{s \mid r \rightarrow s,\ r \in L\}$. Furthermore, we define the following inductively,

$$\begin{aligned}
\gamma^0(L) &= L, \\
\gamma^{i+1}(L) &= \gamma(\gamma^i(L)), \text{ for } i \geq 0, \\
\gamma^*(L) &= \bigcup_{i \geq 0} \gamma^i(L), \\
\gamma^+(L) &= \bigcup_{i > 0} \gamma^i(L), \\
\gamma^{\leq i}(L) &= \bigcup_{0 \leq j \leq i} \gamma^j(L),
\end{aligned}$$

where $\gamma^*(L)$ is referred to as the *iterated transposition of* L.

We say a transposition schema is a *type 1* (respectively *type 2*) transposition schema if all transpositions are of type 1 (respectively type 2). In this case, we identify the transpositions as ordered triples and omit the final coordinate. We say a set of transpositions has *finite contexts* if $\{u_1u_3 \mid (u_1, u_2, u_3) \in T\}$ is finite. We say the set has *finite appearances* if $\{u_2 \mid (u_1, u_2, u_3) \in T\}$ is finite.

In order to discuss T as belonging to a language family, we will often write T as being a subset of $\#\Sigma^*\$\Sigma^*\$\Sigma^*\#$ by identifying $t = (x, u, v)$ in T as $t = \#x\$u\$v\#$ where \$ and # are any symbols not in Σ.

Note that $\gamma^*(L) = \gamma^+(L) \cup \{L\}$ in the above definition. Also, note that T is finite if and only if T has both finite contexts and finite appearances. Intuitively, $\gamma^*(L)$ will consist of all strings which can be produced after applying any arbitrary number of transpositions to any string in L. Note that "nested transpositions", or transpositions being inserted into another transposition have been found to occur [7].

4 Iterated type 1 transpositions

The contextual "copy-and-paste" nature of schemata restricted to type 1 transpositions suggests a natural relationship with the pure insertion grammars. Pure grammars have no nonterminal symbols and thus all words derivable from a finite set of axioms using a finite set of rules belong to the language generated by the pure grammar.

Proposition 4.1 *Let $L \subseteq \Sigma^*$ be a language generated by a pure insertion grammar $G = (\Sigma, A, P)$. There exist finite languages $L', T \subseteq \Sigma^*$ and a type 1 transposition schema $\gamma = (\Sigma \cup \{\$\}, T)$ such that $\gamma^*(L')(\$\Sigma^*)^{-1} = L$ where $\$ \notin \Sigma$.*

Proof. *(Sketch)*We recall the definition of a pure insertion grammar as a triple $G = (\Sigma, A, P)$ where Σ is a finite alphabet, $A \subseteq \Sigma^*$ is a finite set of axioms and $P \subseteq \Sigma^* \times \Sigma^* \times \Sigma^*$ is a finite set of insertion rules. The derivation relation ($\Rightarrow$) for a pure insertion grammar is defined such that for $u, v \in \Sigma^*$, $u \Rightarrow v$ iff $u = u'xyu''$, $v = u'xwyu''$, for some $(x, w, y) \in P$, $u, u'' \in \Sigma^*$. The language generated by such a grammar is defined in the usual way.

Given a pure insertion grammar G, we denote by w_P a word consisting of the concatenation of the second component of all rules in P with an arbitrary, but fixed, ordering on P. Formally, $w_P = \prod_{p \in P} \pi_2(p)$, where π is the projection function.

We now construct a finite language $L' \subseteq \Sigma \cup \{\$\}$ and a type-1 transposition schema $\gamma = (\Sigma \cup \{\$\}, T)$ as follows: for all $w \in A$, we include $w\$w_P \in L'$, as A is finite, so must be L'; for all rules $(x, u, y) \in P$ we include $(x, u, y) \in T$ which, again, must be finite.

It is true that $L(G) \subseteq \gamma^*(L')(\$w_p)^{-1}$, as each derivation applied in G can be simulated by the corresponding transposition in γ as the second coordinate of the transposition will appear in w_p. Thus, $L(G) \subseteq \gamma^*(L')(\$\Sigma^*)^{-1}$. It is clear that $\gamma^*(L')(\$\Sigma^*)^{-1} \subseteq L(G)$, as any transposition applied acts on either the part of the word before \$, or after \$. If it gets applied before the \$, then the segment before \$ is in $L(G)$. ■

The following corollaries are now immediate from [3], [5].

Corollary 4.1 *For every recursively enumerable language L there exist homomorphisms h and g, a finite language L' and a finite type 1 transposition schema γ such that $g(h^{-1}(\gamma^*(L')(\$\Sigma^*)^{-1})) = L$. There exist also a regular language R, a finite language L'' and a finite type 1 transposition schema γ_1 such that $\gamma_1^*(L'')(R)^{-1} = L$.*

Corollary 4.2 *There exist non-semilinear languages which can be generated by a finite type 1 transposition schema acting on a finite language.*

5 Iterated type 2 transpositions

In this section, we will explore basic mathematical properties and computational capacity of iterated type 2 transpositions.

The following are immediate from the definitions of type-2 transposition systems:

Lemma 5.1 *Let $\gamma = (\Sigma, T)$ be a type-2 transposition schema with $L \subseteq \Sigma^*$. Then the following are true:*

1. $p_\Sigma(L) = p_\Sigma(\gamma^*(L))$,
2. $\gamma^*(L) \subseteq \mathrm{perm}(L)$,
3. *$\gamma^*(L)$ is semilinear if and only if L is semilinear,*
4. *$w \in \gamma^*(L)$ if and only if $w \in \bar{\gamma}^*(L')$ where $\bar{\gamma} = (\Sigma, T^{\leq |w|})$ and $L' = \mathrm{perm}(w) \cap L$.*
5. *If L is finite, then $\gamma^*(L) = \bar{\gamma}^*(L)$, where $\bar{\gamma} = (\Sigma, T^{\leq n}), n = \max\{|v| \mid v \in L\}$*

Proof. The first part is immediate as $r \to s$ implies $p_\Sigma(r) = p_\Sigma(s)$, and the second and third parts follow from the first. We will prove the fourth statement as follows:

"$\Rightarrow$" Assume $w \in \gamma^*(L)$. It follows from part (2) that it is enough to use L'. There exists $v_0, \ldots, v_n \in \gamma^*(L)$ and $t_1, \ldots, t_n \in T$ such that $v_i \to_{\{t_{i+1}\}} v_{i+1}$ for every $0 \leq i < n$ where $v_0 \in L$. In particular, in order for each t_i to be used, by the definition of a transposition system, $|t_i| \leq |w|$.

"$\Leftarrow$" immediate.

The fifth statement follows from the fourth. ■

Thus, in terms of generative power, if L is finite, we can assume T is also, by ignoring rules of T which are too long to be used.

Lemma 5.2 *Let $\Gamma = (\Sigma, T)$ be a type-2 transposition schema with $L \subseteq \Sigma^*$ and $\{(\lambda, a, \lambda) \mid a \in \Sigma\} \subseteq T$. Then $\gamma^*(L) = \mathrm{perm}(L)$.*

Proof. "$\subseteq$" This follows from Lemma 5.1 (2).

"$\supseteq$" This follows as, for any $w \in L$, we can move every letter $a \in \Sigma$ to any position of w. Thus, $\mathrm{perm}(w) \subseteq \gamma^*(L)$, and hence $\mathrm{perm}(L) \subseteq \gamma^*(L)$. ■

The following is now immediate, since we know that the families of regular and context-free languages are not closed under permutation.

Corollary 5.1 *The families of regular and context-free languages are not closed under type-2 transpositions with finite languages. Indeed they are not closed under transposition languages of size $|\Sigma|$.*

6 Non-iterated transpositions

In this section, we will explore the power of applying the transformations a single time. These results will become important for the next section which studies the operations algorithmically.

First we will see that under common formal language theoretic operations, closure under iterated transpositions implies closure under non-iterated transpositions.

Proposition 6.1 *Let $z \in \{1, 2\}$. Let $\mathcal{L}_1$ be a language family closed under λ-free homomorphism, inverse homomorphism and intersection with regular languages, and let $\mathcal{L}_2$ be closed under inverse homomorphism and intersection with regular languages. If $\mathcal{L}_1$ is closed under iterated type-z transpositions from $\mathcal{L}_2$, then $\mathcal{L}_1$ is closed under non-iterated type-z transpositions from $\mathcal{L}_2$.*

Proof. First we will consider type-1. Let $L \in \mathcal{L}_1$, let $\gamma = (\Sigma, T)$ be a transposition schema, $T \in \mathcal{L}_2$, $T \subseteq \#\Sigma^*\$\Sigma^*\$\Sigma^*\#$, and let h be a homomorphism from $(\Sigma \cup \{\alpha\})^*$ to Σ^*, where α is a new symbol, defined by $h(a) = a$, for each $a \in \Sigma$ and $h(\alpha) = \lambda$. Let $R = (\alpha\Sigma)^*\alpha$, a regular language, and let $L' = h^{-1}(L) \cap R$. Thus, each word of L has α inserted between every two letters. Let $\bar{T} = h^{-1}(T) \cap \#(\alpha\Sigma)^*\alpha\$(\alpha\Sigma)^+\$(\Sigma\alpha)^*\#$, and let $\bar{\gamma} = (\Sigma \cup \{\alpha\}, \bar{T})$. Then, it is evident that $\bar{\gamma}^*(L') \cap ((\alpha\Sigma)^*\alpha\alpha(\Sigma\alpha)^* \cup (\alpha\Sigma)^*\alpha\alpha(\Sigma\alpha)^*\Sigma\Sigma\alpha(\Sigma\alpha)^*) = \bar{\gamma}(L')$ because any word

in $\bar{\gamma}^2(L') - \bar{\gamma}(L')$ would produce either three consecutive α's, or two sections of two α's. Furthermore, $h(\bar{\gamma}(L')) = \gamma(L)$, and every language family closed under λ-free homomorphism, inverse homomorphism and intersection with regular languages is also closed under linear-erasing homomorphisms. Thus type-1 follows.

The same proof works identically with type-2. ■

Proposition 6.2 *Let $z \in \{1, 2\}$. The family of counter languages (and the context-free languages) are not closed under non-iterated type-z regular transpositions with contexts of size 0. Furthermore, the same is true with iterated transpositions.*

Proof. We will start with type-1 transpositions. Assume otherwise. Let $L = \{a^n q a^n q \mid n \geq 0\}$, and let $T = \{(\lambda, qa^n q, \lambda) \mid n \geq 0\}$, and let $\gamma = (\{a, q\}, T)$ be a transposition system. Then consider $\gamma(L) \cap a^*qa^*qqa^*q = \gamma^*(L) \cap a^*qa^*qqa^*q = \{a^n qa^n qqa^n q \mid n \geq 0\}$, which isn't a counter language, a contradiction.

Next, we consider type-2 transpositions. Assume otherwise. Let $L = \{a^n qpa^n pa^m qa^m \mid n, m \geq 0\}$. Let $T = (\lambda, pa^n p, \lambda) \mid n \geq 0\}$, and let $\gamma = (\{p, q, a\}, T)$ be a transposition system. Then, $\gamma(L) \cap a^*qa^*qpa^*pa^* = \gamma^*(L) \cap a^*qa^*qpa^*pa^* = \{a^n qa^m qpa^n pa^m \mid n, m \geq 0\}$, which isn't a counter language, contradiction. ■

We see however, that for the non-iterated version, it is the unbounded appearances which make the difference.

Proposition 6.3 *Let $\mathcal{L}$ be a language family closed under inverse homomorphism, λ-free homomorphism and intersection with regular languages. Then the following are true:*

1. *$\mathcal{L}$ is closed under regular type-1 transpositions with finite appearances,*
2. *If $\mathcal{L}$ is closed under arbitrary homomorphism, then $\mathcal{L}$ is closed under regular type-2 transpositions with finite appearances,*
3. *If $\mathcal{L}$ is closed under arbitrary homomorphism, then $\mathcal{L}$ is closed under regular transpositions[1] with finite appearances.*

Proof. We will start with the second statement, as the first and third are easy variants of this. Let $\gamma = (\Sigma, T)$ be a transposition system where T is a regular language (which for simplicity's sake, we will denote as being a subset of $\#\Sigma^*\$\Sigma^*\$\Sigma^*\#$), with finite appearances. Let $M = (Q, \Sigma, q_0, F, \delta)$ be a deterministic finite automaton accepting T. Let $L \in \mathcal{L}, L \subseteq \Sigma^*$. Let $X = \{h_2^c(t) \mid t \in T\}$, which is finite, and for each $x \in X$, let $Q(x) = \{(q_1, q_2, q_3, q_4) \mid \#u_1\$x\$u_3\# \in T, \delta(q_0, \#) = q_1, \delta(q_1, u_1) = q_2, \delta(q_2, \$x\$) = q_3, \delta(q_3, u_3) = q_4\}$. Each of these sets are finite.

Let $\bar{\Sigma} = \{\bar{a} \mid a \in \Sigma\}$, and let h be a homomorphism from $(\Sigma \cup \bar{\Sigma})^*$ to Σ^* defined by $h(a) = h(\bar{a}) = a$, for each $a \in \Sigma$. Let $\bar{X}$ be the barred version of X and let $L' = h^{-1}(L) \cap \Sigma^*\bar{X}\Sigma^* \in \mathcal{L}$.

Next, we construct a nondeterministic gsm K which operates on L' as follows: first, on input $u_1u_2u_3, u_1u_3 \in \Sigma^*, u_2 \in \bar{\Sigma}^*$, K nondeterministically guesses $x \in X$, and $(q_1, q_2, q_3, q_4) \in Q(x)$ and in parallel does the following

[1] This can include both type-1 and type-2 transpositions.

1. verifies $h(u_2) = x$ and erases u_2,

2. while reading $u_1u_3 = a_1 \cdots a_m$ (ie. ignoring the barred letters), guesses positions $i, j, k, 1 \leq i \leq j < k$ and verifies that $\delta(q_1, a_i \cdots a_j) = q_2$, then outputs x, then verifies that $\delta(q_3, a_{j+1} \cdots a_k) = q_4$.

Then $K(L') = \gamma(L)$ and as every language family satisfying the assumptions is closed under nondeterministic gsms, it follows that $K(L') \in \mathcal{L}$.

We shall next deal with type-1 transpositions. If in the proof above, we modify K so that it does not erase u_2, but rather outputs $h(u_2) = x$, then K is λ-free, and $K(L') = \gamma(L)$.

Lastly, if T consists of both type-1 and type-2 transpositions, then the nondeterministic gsm can guess at the beginning of its computation. ■

7 Membership of transpositions

We now discuss the algorithmic problem of deciding whether a given string is a possible product of iterated transpositions. We will start with type-1 only.

Proposition 7.1 *Let $\gamma = (\Sigma, T)$ be a type-1 transposition schema with $L \subseteq \Sigma^*$ and $w \in \Sigma^*$. Then $w \in \gamma^*(L)$ if and only if $w \in \bar{\gamma}^{\leq |w|}(L')$ where $\bar{\gamma} = (\Sigma, T^{\leq |w|})$ and $L' = L \cap \Sigma^{\leq |w|}$.*

Proof. "⇒" We will proceed by induction on the length of w. If $|w| = 0$, then it is true. Assume it is true for every v with $|v| \leq n$. Let $|s| = n + 1$. If $s \in \bar{\gamma}^{\leq |s|-1}(L \cap \Sigma^{|s|-1})$, where $\bar{\gamma} = (\Sigma, T^{|s|-1})$, then we are done. Assume otherwise. Then $r \rightarrow_{\{t\}} s$, for some $r \neq s$ with $r \in \bar{\gamma}^{\leq |s|-1}(L \cap \Sigma^{\leq |s|-1})$ where $\bar{\gamma} = (\Sigma, T^{|s|-1})$. Then $t \in T^{\leq |s|}$ and thus $s \in \bar{\bar{\gamma}}^{\leq |s|}(L \cap \Sigma^{\leq |s|})$ where $\bar{\bar{\gamma}} = (\Sigma, T^{\leq |s|})$.

"⇐" immediate. ■

We see that there is an algorithm as long as we can determine if a given string is in L and T.

Proposition 7.2 *Let $\gamma = (\Sigma, T)$ be a type-1 transposition schema with $L \subseteq \Sigma^*$ where we can decide membership in L and T and let $w \in \Sigma^*$. Then we can decide whether $w \in \gamma^+(L)$.*

Proof. By Proposition 7.1, it suffices to decide if $w \in \bar{\gamma}^{\leq |w|}(L')$ where $\bar{\gamma} = (\Sigma, T^{\leq |w|})$ and $L' = L \cap \Sigma^{\leq |w|}$. As L and T have a decidable membership problem, we can effectively construct both L' and $T^{\leq |w|}$. As L' and $T^{\leq |w|}$ are both finite, it follows that we can effectively construct the finite language $\bar{\gamma}^{\leq |w|}(L')$ and then we can test if w is in this language. ■

Next, we will study the same question for type-2 transpostions.

Let $\gamma = (\Sigma, T)$ be a type-2 transposition schema and let $L \subseteq \Sigma^*$. We wish to find an algorithm that, given $w \in \Sigma^*$, will decide whether or not $w \in \gamma^+(L)$.

Definition 7.1 *Let $\gamma = (\Sigma, T)$ be a type-2 transposition schema and let $L \subseteq \Sigma^*$. The (γ, L)-graph, $G_{(\gamma,L)} = (V_{(\gamma,L)}, E_{(\gamma,L)})$, is the smallest directed graph[2] satisfying the following two properties:*

1. $x \in L \Rightarrow x \in V_{(\gamma,L)}$,
2. $x \in V_{(\gamma,L)}, x \rightarrow_T y \Rightarrow y \in V_{(\gamma,L)}, (x, y) \in E_{(\gamma,L)}$.

Given $v \in \Sigma^*$, denote by $G^v_{(\gamma,L)}$ the subgraph of $G_{(\gamma,L)} = (V^v_{(\gamma,L)}, E^v_{(\gamma,L)})$ consisting of all the connected components that have Parikh vector equal to that of v.

This leads naturally to the following proposition:

Proposition 7.3 *Let $\gamma = (\Sigma, T)$ be a type-2 transposition schema and let $L \subseteq \Sigma^*$. The following are true:*

1. *Given a vertex v in $G_{(\gamma,L)}$, every vertex in the connected component of v has the same Parikh vector,*
2. *Each connected component of $G_{(\gamma,L)}$ is finite,*
3. *If L is finite, then $G_{(\gamma,L)}$ is finite and $G_{(\gamma,L)}$ consists of at most $|L|$ connected components,*
4. *For any $v \in \Sigma^*$, $V^v_{(\gamma,L)} = \gamma^*(\mathrm{perm}(v) \cap L)$.*
5. *For any $v \in \Sigma^*$, it follows that $G^v_{(\gamma,L)}$ is finite.*

We solve the problem first if L and T are finite.

Proposition 7.4 *Let $\gamma = (\Sigma, T)$ be a type-2 transposition schema and $L \subseteq \Sigma^*$ where L and T are finite and effectively given, and let $w \in \Sigma^*$. Then there is an algorithm which can determine if $w \in \gamma^*(L)$.*

Proof. It is immediate from the definition that $\gamma^{i+1}(L) = \gamma^i(L)$ implies $\gamma^*(L) = \gamma^i(L)$. Consider $L' = \mathrm{perm}(w) \cap L$. Then $V^w_{(\gamma,L)} = \gamma^*(L')$ and $G^w_{(\gamma,L)}$ is finite, by Lemma 7.3 (4),(5). Then we construct $\gamma^i(L')$, for every i, until $\gamma^{i+1}(L') = \gamma^i(L') = \gamma^*(L') = V^w_{(\gamma,L)}$ which must occur since $G^w_{(\gamma,L)}$ is finite. Then, we test whether $w \in V^w_{(\gamma,L)}$. ■

More generally, we can decide membership in $\gamma^+(L)$ as long as we can within L and T.

Proposition 7.5 *Let $\gamma = (\Sigma, T)$ be a type-2 transposition schema, let $L \subseteq \Sigma^*$ where we can decide membership in L and T and let $w \in \Sigma^*$. Then it is decidable whether $w \in \gamma^*(T)$.*

[2]We define a smallest directed graph satisfying property p to be a graph satisfying p with a minimal number of vertices. In this case, there is only one such graph, so we refer to it as *the* smallest graph.

Proof. Indeed, as we can decide membership in L and T, it is possible to construct $L \cap \text{perm}(w)$ and $T^{\leq |w|}$ by testing whether v is in L, for each $v \in \text{perm}(w)$ and by testing whether x is in T, for each $|x| \leq |w|$. The proposition then follows from Lemma 5.1 (4) and Proposition 7.4. ■

Lastly, we will use these results to decide membership in $\gamma^+(L)$, even if γ contains both type-1 and 2 transpositions.

Proposition 7.6 *Let $\gamma = (\Sigma, T)$ be a transposition schema (potentially containing both type-1 and 2 rules), let $L \subseteq \Sigma^*$ where we can decide membership in L and T and let $w \in \Sigma^*$. Then it is decidable whether $w \in \gamma^*(T)$.*

Proof. Let T_1 (respectively T_2) be the subset of T with type-1 (respectively type-2) rules. Let $\gamma_1 = (\Sigma, T_1)$ and $\gamma_2 = (\Sigma, T_2)$.

Let $L_1 = L \cap \Sigma$ and for all $n \geq 1$, let

$$L_{n+1} = (\gamma_2^*(\gamma_1(L_n) \cup (L \cap \Sigma^{n+1})) \cap \Sigma^{\leq n+1}) \cup L_n.$$

We will show by induction that for all $n \geq 1$, $L_n = \gamma^*(L) \cap \Sigma^{\leq n}$.

It is clear that $L_1 = \gamma^*(L) \cap \Sigma^{\leq 1}$. Let $k \geq 1$ and assume $L_k = \gamma^*(L) \cap \Sigma^{\leq k}$. It is immediate that $L_{k+1} \subseteq \gamma^*(L) \cap \Sigma^{\leq k+1}$. Let $w \in \gamma^*(L) \cap \Sigma^{k+1}$. Thus, either $v \to v' \to^* w$ where $v \in L_k = \gamma^*(L) \cap \Sigma^{\leq k}$ and $v' \notin L_k$, or $v \to^* w$ where $v \in L \cap \Sigma^{k+1}$. Assume the first case. Then $|v| < |v'| = |w|$, and thus $|v'|$ must be obtained via one application of a rule from γ_1 followed by zero or more from γ_2. Thus, $w \in L_{k+1}$. Assume the second case. Then w must be obtained from v via zero or more applications of γ_2 from $L \cap \Sigma^{k+1}$ and thus $w \in L_{k+1}$. Hence, by induction, $L_n = \gamma^*(L) \cap \Sigma^{\leq n}$, for all $n \geq 1$.

Finally, to test whether $w \in \gamma^*(L)$, it suffices to check whether $\gamma^*(L) \cap \Sigma^{\leq |w|}$. To start, we can construct the finite languages $L' = L \cap \Sigma^{\leq |w|}$ and $T^{\leq |w|}$ by deciding membership in L and T. Then, we can iteratively construct $L_1, \ldots, L_{|w|}$ as follows: at iteration $i+1$, we add in all words in L of length $i+1$, all words of length $i+1$ obtained via one application of a type-1 rule from L_i, and then all words obtained via iterated type-2 transpositions via these new words which we can determine by Proposition 7.5. Thus, we can decide if $w \in L_{|w|} = \gamma^*(L) \cap \Sigma^{\leq |w|}$ and if $w \in \gamma^*(L)$.

■

8 Conclusions

We have abstracted the process of both classes of transposable elements to operations on strings and languages. We investigated some basic mathematical properties and the computational power of transpositions. In particular, it was shown that we can generate arbitrary recursively enumerable languages from finite initial and finite transposition languages, modulo right quotient by a regular set. Moreover, non-semilinear languages can be generated similarly. For type-2 transpositions, we can generate only semilinear languages, but the operation is strictly more powerful than permutation. Algorithmically, it was demonstrated that we can decide membership after application of both iterated type-1 and iterated type-2 transpositions, so long

as we can decide membership in both the initial and transposition languages. Then, we were able to use these results to show decidability even when both type-1 and 2 transpositions were present simultaneously.

Future work will consider the time complexity necessary to determine this and other decision questions. Furthermore, we would like to study these complexity questions under certain realistic assumptions, in an attempt for the algorithms to be useful from the perspective of bioinformatics and the analysis of data.

9 Acknowledgements

We are very grateful to a referee for many improvements to this paper.

References

[1] International Human Genome Sequencing Consortium. Initial sequencing and analysis of the human genome. *Nature*, 409:860–921, 2001.

[2] M. A. Harrison. *Introduction to Formal Language Theory.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1978.

[3] L. Kari and P. Sosík. On the weight of universal insertion grammars. In C. Mereghetti, B. Palano, G. Pighizzini, and D. Wotschke, editors, *Seventh Int. Workshop on Descriptional Complexity of Formal Systems*, pages 202–214. Universita degli Studi di Milano, 2005.

[4] M. G. Kidwell. Transposable elements. In T. Ryan Gregory, editor, *The Evolution of The Genome*, chapter 3. Elsevier Academic Press, 2005.

[5] C. Martin-Vide, Gh. Păun, and A. Salomaa. A characterization of recursively enumerable languages by means of insertion grammars. *Theoretical Computer Science*, 205:195–205, 1998.

[6] B.C. Meyers, S.V. Tingey, and M. Morgante. Abundance, distribution, and transcriptional activity of repetitive elements in the maize genome. *Genome Res.*, 11:1660–1676, 2001.

[7] H. Quesneville, C. M. Bergman, O. Andrieu, D. Autard, D. Nouaud, M. Ashburner, and D. Anxolabehere. Combined evidence annotation of transposable elements in genome sequences. *PLoS Computational Biology*, 1(2):166–175, 2005.

[8] A. Salomaa. *Formal Languages.* Academic Press, New York, 1973.

Computational Efficiency of Intermolecular Gene Assembly

Tseren-Onolt Ishdorj[1], Remco Loos[2], and Ion Petre[1]

[1]Computational Biomodelling Laboratory
Department of Information Technologies
Åbo Akademi University, Turku 20520 Finland
E-mail:{tishdorj,ipetre}@abo.fi
[2]Research Group on Mathematical Linguistics
Rovira i Virgili University
Plaça Imperial Tàrraco 1, 43005 Tarragona, Spain.
E-mail:remcogerard.loos@urv.cat

Abstract

In this paper, we investigate the computational efficiency of gene rearrangement operations found in ciliates, a type of unicellular organisms. We show how the so-called guided recombination systems, which model this gene rearrangement, can be used as problem solvers. Specifically, we prove that these systems can uniformly solve SAT in time $O(n \cdot m)$ for a boolean formula of m clauses over n variables.

1 Introduction

Ciliates are unicellular organisms [1] which have attracted attention from computer scientists because of the complex nature of the gene rearrangement of some species. Specifically, the DNA in their micronucleus, used for conjugation only, is transformed into shorter molecules used for transcription. This process is called *gene assembly* and is in some sense reminiscent of the use of "linked lists" in software engineering, see [1].

Two main computational models have been proposed to model gene assembly in ciliates. The so-called *intermolecular* model, introduced by Landweber and Kari [10, 8], allows for operations involving two molecules. The *intramolecular* model, proposed by Ehrenfeucht, Prescott and Rozenberg [2, 14], contains only operations acting on a single molecule. The computational power of the intermolecular model has been well studied, specifically in [8] it is shown that in some formulation the model is as powerful as a Turing machine. It was recently proved that also the intramolecular model is computationally universal [7].

Continuing the investigation of gene assembly from the perspective of computability theory, it was recently proved that the intramolecular model is computationally efficient: SAT may be solved in this model in linear time, see [6]. We refer to

[11] for a related result on splicing systems. In this paper we address the same question for the intermolecular model. We show how the guided recombination model of [8] can be regarded as a problem solving device. The model we consider involves the maximal parallel application of contextual recombination rules, as defined in this paper. We present an algorithm to show that in this model, SAT can be solved in time $O(n \cdot m)$ by a guided recombination system, with n denoting the number of variables and m the number of clauses.

2 Preliminaries and Notation

We assume the reader to be familiar with the basic elements of formal languages, Turing computability [15], DNA computing [13], and computational complexity [12]. We present here only some of the necessary notions and notation.

An *alphabet* is a finite set of symbols (letters), and a word (string) over an alphabet Σ is a finite sequence of letters from Σ; the empty word we denote by λ. The set of all words over an alphabet Σ is denoted by Σ^*. The set of all non-empty words over Σ is denoted as Σ^+, i.e., $\Sigma^+ = \Sigma^* \setminus \{\lambda\}$. The length $|x|$ of a word x is the number of symbols that x contains. The empty word has length 0.

We also define circular words over Σ by declaring two words u, v to be equivalent if and only if $u = xy$ and $v = yx$, for some words x, y. We also call u, v *conjugates*. Then the *circular word* $\bullet w$ is the equivalence class of w with respect to this relation, for all $w \in \Sigma^*$. The set of all circular words over Σ is denoted by $\Sigma^\bullet$.

A splicing scheme [5] is a pair $R = (\Sigma, \sim)$, where Σ is an alphabet and $\sim$, the pairing relation of the scheme, $\sim \subseteq \Sigma^*\Sigma^+\Sigma^* \times \Sigma^*\Sigma^+\Sigma^*$. Assume we have two strings x, y and a binary relation between two triples of words $(\alpha, p, \beta) \sim (\alpha', p, \beta')$, such that $x = x'\alpha p\beta x''$ and $y = y'\alpha' p\beta' y''$; then, the strings obtained by the recombination in the context from above are $z_1 = x'\alpha p\beta' y''$ and $z_2 = y'\alpha' p\beta x''$.

When having a pair $(\alpha, p, \beta) \sim (\alpha', p, \beta')$ and two strings x and y as above, $x = x'\alpha p\beta x''$ and $y = y'\alpha' p\beta' y''$, we consider just the string $z_1 = x'\alpha p\beta' y''$ as the result of the recombination (we call it one-output-recombination), because the string $z_2 = y'\alpha' p\beta x''$, we consider as the result of the one-output-recombination with the respect to the symmetric pair $(\alpha', p, \beta') \sim (\alpha, p, \beta)$.

A *Boolean expression* is an expression composed of variables, parentheses and the operators $\bar{\cdot}$, $\wedge$ and $\vee$. The variables can take values 0 (false) and 1 (true). An expression is satisfiable if there is some assignment of variables such that the expression is true. The *satisfiability problem*, commonly denoted as SAT, is to determine, given a Boolean expression, whether it is satisfiable. SAT is a well known NP-complete problem. A Boolean expression is said to be in *conjunctive normal form (CNF)* if it is of the form $E_1 \wedge E_2 \wedge \cdots \wedge E_k$, where each E_i, called a *clause*, is of the form $\alpha_{i1} \vee \alpha_{i2} \vee \ldots \vee \alpha_{ir_i}$, where each α_{ij} is a literal, that is either x or $\bar{x}$, for some variable x.

3 Guided recombination systems

A splicing scheme is a pair $P = (\Sigma, \sim)$, where Σ is an alphabet and $\sim$, the pairing relation of the scheme, $\sim \subseteq \Sigma^*\Sigma^+\Sigma^* \times \Sigma^*\Sigma^+\Sigma^*$. In the splicing scheme $P = (\Sigma, \sim)$ pairs $(\alpha, p, \beta) \sim (\alpha', p, \beta')$ define the contexts necessary for a recombination between the repeats p. Then the *contextual intramolecular recombination* was defined in [8].

$$(\mathsf{del}_p) \qquad \{upwpv\} \Longrightarrow_{\mathsf{del}_p} \{upv, \bullet wp\},$$
$$\text{where} \qquad u = u'\alpha, w = \beta w' = w''\alpha', v = \beta' v', \text{ and } (\alpha, p, \beta) \sim (\alpha', p, \beta').$$

This constrains intramolecular recombination within $upwpv$ to occur only if the restrictions of the splicing scheme concerning p are fulfilled, i.e., the first occurrence of p is preceded by α and followed by β and its second occurrence is preceded by α' and followed by β'.

Also, if $(\alpha, p, \beta) \sim (\alpha', p, \beta')$, then the *contextual intermolecular recombination* was defined in [8] as

$$(\mathsf{ins}_p) \qquad \{upv, \bullet wp\} \Longrightarrow_{\mathsf{ins}_p} \{upwpv\}$$
$$\text{where} \qquad u = u'\alpha, v = \beta v', w = w'\alpha' = \beta' w'', \text{ and } (\alpha, p, \beta) \sim (\alpha', p, \beta').$$

Intermolecular recombination between the linear strand upv and the circular strand $\bullet wp$ may take place only if the occurrence of p in the linear strand is flanked by α and β and its occurrence in the circular strand is flanked by α' and β'.

Definition 1 *For a splicing scheme $P = (\Sigma, \sim)$, we define the set of all contextual gene rearrangement operations under guiding of the splicing scheme P as follows:*

$$\widetilde{P} = \{\mathsf{ins}_p, \mathsf{del}_p \mid (\alpha, p, \beta) \sim (\alpha', p, \beta') \text{ for some } \alpha, \alpha', \beta, \beta' \in \Sigma^*\}.$$

We now define a *guided recombination system* that captures the series of dispersed homologous recombination events that take place during scrambled gene rearrangement in ciliates.

Definition 2 *A guided recombination system is a triple $R = (\Sigma, \sim, t)$ where $(\Sigma, \sim)$ is a splicing scheme, and $t \in \Sigma^+$ is a linear string called the axiom.*

A guided recombination system R defines a derivation relation that produces a new multiset from a given multiset of linear and circular strands, as follows. Starting from a "collection" (multiset) of strings with a certain number of available copies of each string, the next multiset is derived from the first one by an intra- or intermolecular recombination between existing strings. The strands participating in the recombination are "consumed" (their multiplicity decreases by 1) whereas the products of the recombination are added to the multiset (their multiplicity increases by 1).

For two multisets S and S' in $\Sigma^* \cup \Sigma^\bullet$, we say that S derives S' and we write $S \Longrightarrow_R S'$, iff one of the following two cases hold:

(1) there exist $x \in S, y, \bullet z \in S'$ such that

- $\{x\} \Longrightarrow_{\mathsf{del}} \{y, \bullet z\}$ according to an intramolecular recombination step in R,
- $S'(x) = S(x) - 1, S'(y) = S(y) + 1, S'(\bullet z) = S(\bullet z) + 1$, and $S'(u) = S(u)$ for all $u \notin \{x, y, \bullet z\}$;

(2) there exist $x', \bullet y' \in S, z' \in S'$ such that

- $\{x', \bullet y'\} \Longrightarrow_{\mathsf{ins}} \{z'\}$ according to an intermolecular recombination step in R,
- $S'(x') = S(x') - 1, S'(\bullet y') = S(\bullet y') - 1, S'(z') = S(z') + 1$, and $S'(u) = S(u)$ for all $u \notin \{x', y', \bullet z'\}$.

Those strands which, by repeated recombinations with initial and intermediate strands eventually produce the axiom, form the language of the guided recombination system. Formally,

$$L_a^k(R) = \{w \in \Sigma^* \mid \{(w, k)\} \Longrightarrow_R^* S \text{ and } t \in S\}$$

((w, k) indicates the fact that the multiplicity of w equals k).

The guided recombination systems are proved in [8] to be equivalent to Turing machine:

Theorem 1 ([8]) *Let L be a language over T^* accepted by a Turing machine $TM = (S, \Sigma \cup \{\#\}, P)$. Then there exist an alphabet Σ', a sequence $\pi \in \Sigma'^*$, depending on L, and a recombination system R such that a word w over T^* is in L iff $\#^6 s_0 w \#^6 \pi$ belongs to $L_a^k(R)$ for some $k \geq 1$.*

In line with this result, we define acceptance for guided recombination systems as follows.

Definition 3 *We say a guided recombination system $R = (\Sigma, \sim, t)$* accepts *a string w iff there exists a $k \geq 1$ such that $w \in L_a^k(R)$.*

In other words, a guided recombination system accepts a string w if it generates the axiom, when starting with some (sufficient) amount of copies of w.

We now consider the parallelism for the guided recombination model. Intuitively, a number of operations can be applied in parallel to a string if the applicability of each operation is independent of the applicability of the other operations.

In this paper we use a notion of parallelism following [6], which is the *maximally parallel* application of a rule to a string.

First, we define the working places of a operation $\pi \in \widetilde{P}$ on a given string where π is applicable.

Definition 4 *Let w be a string. The working places of a operation $\pi \in \widetilde{P}$ with respect to a multiset S for w is a set of substrings of w written as $Wp(\pi(w))$ and defined by*

$$\begin{aligned} Wp(\mathsf{del}_p(w)) &= \{upw'pv \in Sub(w) \mid \{upw'pv\} \Longrightarrow_{\mathsf{del}_p} \{upv, \bullet w'p\}\}, \\ Wp(\mathsf{ins}_p(w)) &= \{upv \in Sub(w) \mid \{upv, \bullet w'p\} \Longrightarrow_{\mathsf{ins}_p} \{upw'pv\} \\ &\qquad \textit{for some } \bullet w'p \in S\}. \end{aligned}$$

Definition 5 *Let w be a string. The smallest working places of a operation $\pi \in \widetilde{P}$ for w is a subset of $Wp(\pi(w))$ written as $Wp_s(\pi(w))$ and defined by*

$$\begin{aligned} Wp_s(\pi(w)) &= \{w_1 \in Wp(\pi(w)) \mid \text{ for all } w_1' \in Sub(w_1) \\ &\qquad \text{and } w_1' \neq w_1, w_1' \notin Wp(\pi(w))\}. \end{aligned}$$

Definition 6 *Let Σ be a finite alphabet and $\widetilde{P}$ the set of rules defined above. Let $\pi \in \widetilde{P}$ and $u \in \Sigma^*$. We say that $v \in \Sigma^*$ is obtained from u by applying π in a maximally parallel way, denoted $u \Longrightarrow_{\pi}^{max} v$, if*

$$u = \alpha_1 u_1 \alpha_2 u_2 \dots \alpha_k u_k \alpha_{k+1}, \text{ and } v = \alpha_1 v_1 \alpha_2 v_2 \dots \alpha_k v_k \alpha_{k+1},$$

where $u_i \in Wp_s(\pi)(u)$, $v_i \in \Sigma^*$ for all $1 \leq i \leq k$, and also, $\alpha_i \notin Wp(\pi(u))$, for all $1 \leq i \leq k+1$.

Example 1 *Let del_p be the contextual deletion operation applied in the context $(x_1x_2, p, x_3) \sim (x_3, p, x_1)$, and consider the string $u = x_1x_2px_3px_1x_2p\ x_3px_1$. The unique correct result obtained by maximally parallel application of del_p to u is:*

$$x_1x_2px_3px_1x_2px_3px_1 \Longrightarrow_{\mathsf{del}_p}^{max} x_1x_2px_1x_2px_1.$$

Finally, if in a guided recombination system $R = (\Sigma, \sim, t)$ for some multiplicity k $\{(w,k)\} \Longrightarrow_R^n S$, with $t \in S$, we say that R accepts a string w in time n.

4 Efficiency of guided recombination systems

In this section, we use guided recombination systems as decision problem solvers. A possible correspondence between decision problems and languages can be done via an encoding function which transforms an instance of a given decision problem into a word, see, e.g., [3].

Definition 7 *We say that a decision problem X is solved in time $O(t(n))$ by guided recombination systems if there exists a family $\mathcal{A}$ of guided recombination systems such that the following conditions are satisfied:*

1. *The encoding function of any instance x of X having size n can be computed by a deterministic Turing machine in time $O(t(n))$.*
2. *For each instance x of size n of the problem one can effectively construct, in time $O(t(n))$, an intermolecular guided recombination system $G(x) \in \mathcal{A}$ which accepts, again in time $O(t(n))$, the word encoding the instance x if and only if the solution to the given instance of the problem is* `YES`.

Moreover, we say that a solution is *uniform* if all instances of the same size are solved by the same guided recombination system.

Theorem 2 `SAT` *can be solved uniformly and deterministically by a guided recombination system in time $O(n \cdot m)$, where n denotes the number of variables and m the number of clauses.*

Proof. Let us consider a propositional formula ϕ of m clauses over n variables in the conjunctive normal form. Thus $\phi = C_1 \wedge \cdots \wedge C_m$, such that each clause $C_j, 1 \leq j \leq m$, is of the form $C_j = \langle y_{j,1} \vee \cdots \vee y_{j,k_j} \rangle, k_j \geq 1$, where $y_{j,k} \in \{x_i, \bar{x}_i \mid 1 \leq i \leq n\}, 1 \leq k \leq k_j$.

We encode each clause C_j as a string bounded by $\$_j$ in the following form:

$$c_j = \$_j \dagger C_j \dagger \$_j.$$

The instance ϕ is encoded as follows:

$$\hat{\phi} = c_1 \ldots c_m \$_{m+1} \dagger\dagger x_1 \dagger\dagger \bar{x}_1 \dagger\dagger \ldots \dagger\dagger x_n \dagger\dagger \bar{x}_n \dagger\dagger \$_{m+1} \$_{m+2}.$$

It is easily seen that the size of the encoding is linear in n and m.

The string appended to the formula contains both values for all variables in ϕ. We design a guided recombination system which solves the encoded instance of `SAT` in the following steps.

1. Excise the variable values.
2. Insert a valued variable after each clause.
3. Check if the inserted variable satisfies the clause.
4. Check if the inserted variables are consistent.
5. Generate the axiom if and only if both checks are successful.

Specifically, given a boolean formula ϕ with m clauses and over n variables, we construct a guided recombination system

$$G = (\Sigma, \sim, \$),$$

with

$$\begin{aligned} \Sigma &= \{\$_i \mid 1 \leq i \leq m+2\} \cup \{x_i, \bar{x}_i \mid 1 \leq i \leq n\} \cup \{\vee, \langle, \rangle, \dagger\}, \\ \$ &= \$_1 \$_2 \ldots \$_m \$_{m+1} \$_{m+2}. \end{aligned}$$

The relation $\sim$ is defined as follows, where $x \in \{x_i, \bar{x}_i \mid 1 \leq i \leq n\}$, $b \in \{\vee, \langle\}$, $e \in \{\vee, \rangle\}$ and $1 \leq j \leq m$. Also $\bar{x} = \bar{x}_i$ if $x = x_i$ and $\bar{x} = x_i$ if $x = \bar{x}_i$.

$$\begin{aligned} (\dagger, \dagger, x) &\sim (x, \dagger, \dagger), && (1) \\ (\rangle, \dagger, \$_j \$_{j+1}) &\sim (x, \dagger, x), && (2) \\ (b, x, e) &\sim (\rangle \dagger, x, \dagger), && (3) \\ (bx, \dagger, \lambda) &\sim (b\bar{x} \dagger \$_j \$_{j+1}, \dagger, \langle), && (4) \\ (\dagger \$_m, \$_{m+1}, \lambda) &\sim (\lambda, \$_{m+1}, \$_{m+2}), && (5) \\ (\lambda, \$_j, \dagger) &\sim (bx \dagger, \$_j, \$_{j+1} \$_{j+2}). && (6) \end{aligned}$$

The size of this system is $O(n \cdot m)$ and it is not hard to see that it can be constructed by a deterministic Turing machine in time $O(n \cdot m)$.

We will show that G decides SAT for a given input ϕ. That is, that G accepts encoding $\hat{\phi}$ if and only if ϕ is satisfiable.

For the *if*-part, consider the input string

$$\$_1\dagger C_1\dagger\$_1 \ldots \$_m\dagger C_m\dagger\$_m\$_{m+1}\dagger\dagger x_1\dagger\dagger\bar{x}_1\dagger\dagger \ldots \dagger\dagger x_n\dagger\dagger\bar{x}_n\dagger\dagger\$_{m+1}\$_{m+2}.$$

To this word we can apply the operation $\mathsf{del}_\dagger$ using contexts of (1). In fact, we apply $2n$ $\mathsf{del}_\dagger$-operations in parallel, giving

$$\$_1\dagger C_1\dagger\$_1 \ldots \$_m\dagger C_m\dagger\$_m\$_{m+1}\dagger^{2n+2}\$_{m+1}\$_{m+2}$$

as well as the circular strings $\bullet x\dagger$ for all $x \in \{x_i, \bar{x}_i \mid 1 \leq i \leq n\}$.

In the next step, these circular strings can be inserted after each encoding of a clause of ϕ using contexts (2). Again, this is done in parallel for all clauses, so with m $\mathsf{ins}_\dagger$-operations we obtain a string of the form

$$\$_1\dagger C_1\dagger z_1\dagger\$_1 \ldots \$_m\dagger C_m\dagger z_m\dagger\$_m\$_{m+1}\dagger^{2n+2}\$_{m+1}\$_{m+2}$$

with each $z_j, 1 \leq j \leq m$ in $\{x_i, \bar{x}_i \mid 1 \leq i \leq n\}$. We interpret these inserted variables as an assignment, where the variable inserted after each clause verifies this clause. It is important to note that the same variable can be inserted more than once, up to m times, into the same string, since we also have at our disposal circular strings excised from other copies of the input string. Recall that by Definition 3 we can assume that the input word is present in the multiplicity needed to generate all possible assignments of a verifying variable to a clause. If $m > 2n$, extra multiplicity is needed to provide enough variables to insert.

If the formula is satisfiable, there is at least one inserted assignment in which all inserted variables effectively verify the clause preceding it. In this case, we can apply the contexts of (3) to perform m del_x-operations. This gives m strings of the form $\bullet \vee \cdots \vee y_{j,k_j}\rangle\dagger z_i$ and a string

$$\$_1\dagger\langle y_{1,1} \vee \cdots \vee z_1\dagger\$_1 \ldots \$_m\dagger\langle y_{m,1} \vee \cdots \vee z_m\dagger\$_m\$_{m+1}\dagger^{2n+2}\$_{m+1}\$_{m+2}.$$

Recall that each clause $C_j, 1 \leq j \leq m$, is of the form $C_j = \langle y_{j,1} \vee \cdots \vee y_{j,k_j}\rangle, k_j \geq 1$, where $y_{j,k} \in \{x_i, \bar{x}_i \mid 1 \leq i \leq n\}, 1 \leq k \leq k_j$.

Again, if the formula is satisfiable, there is at least one inserted assignment which verifies all clauses and is consistent, in the sense that if a variable x_i is inserted in some place, no variable $\bar{x}_i$ is inserted after another clause. This means no context of (4) can apply to the string, since this requires the simultaneous presence of x_i and $\bar{x}_i$ in the string.

Now we can apply $\mathsf{del}_{\$_{m+1}}$ using context (5). In this and the following steps, no parallel applications are possible, so the derivation goes on sequentially. First, $\mathsf{del}_{\$_{m+1}}$ yields $\bullet\dagger^{2n+2}\$_{m+1}$ and

$$\$_1\dagger\langle y_{1,1} \vee \cdots \vee z_1\dagger\$_1 \ldots \$_m\dagger\langle y_{m,1} \vee \cdots \vee z_m\dagger\$_m\$_{m+1}\$_{m+2}.$$

From here, we apply successively $\mathsf{del}_{\$_m}$ to $\mathsf{del}_{\$_1}$ using the contexts of (6). For instance, $\mathsf{del}_{\$_m}$ gives $\bullet\dagger\langle y_{m,1} \vee \cdots \vee z_m\dagger\$_m$ and

$$\$_1\dagger\langle y_{1,1} \vee \cdots \vee z_1\dagger\$_1 \ldots \$_{m-1}\dagger\langle y_{m-1,1} \vee \cdots \vee z_{m-1}\dagger\$_{m-1}\$_m\$_{m+1}\$_{m+2},$$

after which $\mathsf{del}_{\$_{m-1}}$ can be applied. This process goes on until $\mathsf{del}_{\$_1}$ yields the axiom

$$\$_1\$_2 \ldots \$_{m+1}\$_{m+2}.$$

This means G accepts $\hat{\phi}$.

For the *only if*-part, assume that ϕ is not satisfiable. In this case, the first two steps go on exactly as described above, giving

$$\$_1\dagger C_1\dagger z_1\dagger\$_1 \ldots \$_m\dagger C_m\dagger z_m\dagger\$_m\$_{m+1}\dagger^{2n+2}\$_{m+1}\$_{m+2}.$$

However, ϕ is not satisfiable. This means that for all inserted assignments at least one of the following holds:

1. Not all clauses are verified by the variable inserted after them.
2. The inserted assignment is inconsistent.

For case 1, assume that only clause C_l is not verified by z_l. Then we cannot apply the m parallel del_x-operations as before. In fact, we can only apply $m-1$ operations giving

$$\$_1\dagger\langle y_{1,1} \vee \cdots \vee z_1\dagger\$_1 \ldots \$_l\dagger\langle \ldots \rangle\dagger z_l\dagger\$_l \ldots \$_m\dagger\langle y_{m,1} \vee \cdots \vee z_m\dagger\$_m\$_{m+1}\dagger^{2n+2}\$_{m+1}\$_{m+2}.$$

Alternatively, if $m > 2n$, z_l may not have been inserted. Also then del_{z_l} cannot be applied and we get

$$\$_1\dagger\langle y_{1,1} \vee \cdots \vee z_1\dagger\$_1 \ldots \$_l\dagger\langle \ldots \rangle\dagger\$_l \ldots \$_m\dagger\langle y_{m,1} \vee \cdots \vee z_m\dagger\$_m\$_{m+1}\dagger^{2n+2}\$_{m+1}\$_{m+2}.$$

In both of these cases, if z_{l+1} happens to verify clause l, we could apply $\mathsf{del}_{z_{l+1}}$ differently, resulting in $\bullet \vee \ldots \rangle(\dagger z_l)\dagger\$_l\$_{l+1}\dagger \ldots \rangle\dagger z_l$ and

$$\$_1\dagger\langle y_{1,1} \vee \cdots \vee z_1\dagger\$_1 \ldots \$_l\dagger\langle \ldots z_{l+1}\dagger\$_{l+1} \ldots \$_m\dagger\langle y_{m,1} \vee \cdots \vee z_m\dagger\$_m\$_{m+1}\dagger^{2n+2}\$_{m+1}\$_{m+2}.$$

A similar situation can arise if z_l happens to verify clause $l-1$. Then, instead of $\mathsf{del}_{z_{l-1}}$ we could also apply del_{z_l}, resulting in $\bullet \vee \ldots \rangle\dagger z_{l-1}\dagger\$_{l-1}\$_l\dagger \ldots \rangle\dagger z_l$ and

$$\$_1\dagger\langle y_{1,1} \vee \cdots \vee z_1\dagger\$_1 \ldots \$_{l-1}\dagger\langle \ldots z_l\dagger\$_l \ldots \$_m\dagger\langle y_{m,1} \vee \cdots \vee z_m\dagger\$_m\$_{m+1}\dagger^{2n+2}\$_{m+1}\$_{m+2}.$$

By our maximality requirement, these are the only possibilities. We suppose that no $\mathsf{del}_\dagger$ using contexts (4) can be applied (if it can, this is treated under case 2). Now, $\mathsf{del}_{\$_{m+1}}$ using context (5) is applied as before. Also $\mathsf{del}_{\$_j}$ by (6) until arriving at $\mathsf{del}_{\$_l}$ (or $\mathsf{del}_{\$_{l+1}}$). None of the strings obtained above satisfy the context of (6) at that point, so the derivation of the axiom cannot continue. No other operations can take place, so G does not accept $\hat{\phi}$ by this assignment. If more than one variables

do not satisfy their clause, the situation is the same, except that we can get more substrings of the form $\$_l\dagger\langle\ldots\rangle(\dagger z_l)\dagger\$_l$ or $\$_k\dagger\langle\ldots z_l\dagger\$_l, k<l$.

For case 2, assume all variables satisfy their clauses. In this case, m del_x-operations apply as before, giving

$$\$_1\dagger\langle y_{1,1}\vee\cdots\vee z_1\dagger\$_1\ldots\$_m\dagger\langle y_{m,1}\vee\cdots\vee z_m\dagger\$_m\$_{m+1}\dagger^{2n+2}\$_{m+1}\$_{m+2}.$$

Now suppose that z_p and z_q are inconsistent, for $p<q$. Now, in the same step as $\mathsf{del}_{\$_{m+1}}$ we also apply $\mathsf{del}_\dagger$ by context (4). This gives $\bullet\$_p\cdots\vee z_q\dagger\$_q\$_{q+1}\dagger$ and

$$\$_1\dagger\langle y_{1,1}\vee\cdots\vee z_1\dagger\$_1\cdots\vee z_p\dagger\langle\ldots\rangle z_{q+1}\dagger\$_{q+1}\ldots\ldots\$_m\dagger\langle y_{m,1}\vee\cdots\vee z_m\dagger\$_m\$_{m+1}\$_{m+2}.$$

Note that if case 1 holds for any z_l unequal to z_p and z_q, the same $\mathsf{del}_\dagger$ will still take place. As in case 1, the created string does not satisfy the context of $\mathsf{del}_{\$_q}$, so the axiom cannot be derived.

Since for all possible assignments at least one case holds, the axiom is not generated, so G does not accept $\hat{\phi}$.

Finally, since each context can induce both a del and an ins-operation, we should say a few words about the operations not mentioned before.

- $\mathsf{ins}_\dagger$ by context (1) is not possible since we never have any circular string with two consecutive symbols $\dagger$.
- $\mathsf{del}_\dagger$ by (2) cannot happen because we never have a substring $x\dagger x$ in a circular string.
- ins_x by (3) is impossible because none of the circular strings we obtain has substring $\rangle\dagger x\dagger$.
- $\mathsf{ins}_\dagger$ by (4) cannot take place since none of the circular strings contains substring $\vee x\$_j\$_{j+1}\dagger\langle$ or $\langle x\$_j\$_{j+1}\dagger\langle$.
- $\mathsf{ins}_{\$_{m+1}}$ by (5) is not possible because no circular string contains $\$_{m+1}\$_{m+2}$ (the circular strings generated by del_x (3) and $\mathsf{del}_\dagger$ (4) only contain $\$_1$ to $\$_m$).
- $\mathsf{ins}_{\$_{m+1}}$ by (6) needs a circular string containing $\$_j\$_{j+1}\$_{j+2}$, which is never created.

Our algorithm has a linear running time. Excision of the variables takes at most m steps, since we need to excise from at most m copies of the input string. If the formula is satisfiable, we obtain the axiom after $m+3$ additional steps, giving a total running time of $2m+3$ steps. Finally, the system G we constructed solves all instances of `SAT` of m clauses over n variables, thus making our solution uniform. This concludes the proof. □

5 Conclusion

In this paper we considered a model of gene rearrangement in ciliates. We showed that this model can be used as an efficient problem solving device by presenting an algorithm for solving `SAT` in time $O(n \cdot m)$. One especially interesting feature of our algorithm is that we show that using small local contexts one can perform correctness and consistency checks over arbitrarily large distances. We believe that the study of the gene rearrangement process in ciliates and its formal modelling is not only interesting from a biological point of view, but can also be beneficial for the study of computation.

Acknowledgments. The work of T.-O.I. is supported by the Center for International Mobility (CIMO) Finland, grant TM-06-4036 and by Academy of Finland, project 203667. The work of R.L. was supported by Research Grant BES-2004-6316 of the Spanish Ministry of Education and Science. The work of I.P. is supported by Academy of Finland, project 108421.

References

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

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

[3] Garey, M., Jonhson, D., *Computers and Intractability. A Guide to the Theory of NP-completeness*, Freeman, San Francisco, CA, 1979.

[4] Harju, T., Petre, I., and Rozenberg, G., Two models for gene assembly in ciliates, *Theory is forever, LNCS* 3113:89–101, 2004.

[5] Head, T., Formal Language Theory and DNA: an analysis of the generative capacity of specific recombinant behaviors. *Bull. Math. Biology* 49: 737–759, 1987.

[6] Ishdorj, T.-O, and Petre, I., Computing through gene assembly, accepted in *Int. Conf. UC'07*, 13–17, August, 2007, Kingston, Canada.

[7] Ishdorj, T.-O, Petre, I., and Rogojin, V., Computational power of intramolecular gene assembly, *International Journal of Foundations of Computer Science*, to appear, 2007.

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

[9] Kari, L., and Thierrin, G., Contextual insertion/deletions and computability. *Information and Computation* 131:47–61, 1996.

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

[11] Loos, R., Martín-Vide, C., and Mitrana, V., Solving `SAT` and `HPP` with accepting splicing systems, *PPSN IX, LNCS* 4193:771–777, 2006.

[12] Papadimitriou, Ch. P., *Computational Complexity*. Addison-Wesley, Reading, MA, 1994.

[13] Păun, Gh., Rozenberg, G., Salomaa, A., *DNA Computing - New computing paradigms*, Springer-Verlag, Berlin, 1998.

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

[15] Salomaa, A., *Formal Languages*, Academic Press, New York, 1973.

Simulating Apoptosis Using Discrete Methods: a Membrane System and a Stochastic Approach

John Jack,[1] Francisco J. Romero-Campero,[2] Mario J. Pérez-Jiménez,[2]
Oscar H. Ibarra[3] and Andrei Păun[1,4†]

[1] Department of Computer Science/IfM, Louisiana Tech University,
P.O. Box 10348, Ruston, LA 71272, USA {johnjack, apaun}@latech.edu
[2] Department of Computer Science and Artificial Intelligence, University of Sevilla,
Avda. Reina Mercedes s/n 41012, Sevilla, Spain {fran, marper}@us.es
[3] Department of Computer Science, University of California - Santa Barbara,
Santa Barbara, CA 93106, USA ibarra@cs.ucsb.edu
[4] Universidad Politécnica de Madrid - UPM, Facultad de Informática
Campus de Montegancedo S/N, Boadilla del Monte, 28660 Madrid, Spain

Abstract. Membrane Systems provide an intriguing method for modeling biological systems at a molecular level. The hierarchical structure of Membrane Systems lends itself readily to mimic the nature and behavior of cells. We have refined a technique for modeling the type I and type II FAS-induced apoptosis signalling cascade. Improvements over our previous modeling work on apoptosis include increased efficiency for storing and sorting waiting times of reactions, a nondeterministic approach for handling reactions competing over limited reactants and improvements, and refinements of the model reactions.
The modular nature of our systems provides flexibility with respect to future discoveries on the signal cascade. We provide a breakdown of our algorithms and explanations on improvements we have implemented. We also give an exhaustive comparison to an established ordinary differential equations technique. Based on the results of our simulations, we conclude that Membrane Systems are a useful simulation tool in Systems Biology that could provide new insight into the subcellular processes, and provide also the argument that Membrane Systems may outperform ordinary differential equation simulations when simulating cascades of reactions (as they are observed in cells).

1 Introduction

Many diseases and disorders are linked to the anomalous behavior of the apoptotic pathway within an organism. Specifically, understanding of FAS-induced apoptosis should be useful in combating cancers and HIV as pointed out in [10] and [11]. As new information is discovered on the stoichiometry and biochemical interactions involved in this pathway, scientists look to the computer simulations as a tool for understanding and predicting the effects of changes in molecules important to the pathway. In this paper, we describe two techniques based on discrete methods for modeling molecular signaling cascades within a cell. While the two discrete simulation strategies can be applied to any pathway in the cell, we implemented the new simulation techniques and chose to provide the simulation results for the caspase dependent apoptotic pathway. We provide the algorithms for our two refined approaches, a *Nondeterministic Waiting Time* algorithm and a *Multi-compartmental*

† To whom correspondence should be addressed. E-mail: apaun@latech.edu

Gillespie algorithm. As mentioned above, we have tested our methods on the FAS-induced apoptotic signal cascade and offer comparisons with a modeling technique based on ordinary differential equations as was used for the apoptotic pathway in [8].

The Nondeterministic Waiting Time (NWT) algorithm is a modified version of our Membrane System described in [2]. We provide two major improvements over our previous algorithm. First, we have added a nondeterministic logic to handle reactions competing over limited reactants. And second, we improved the runtime of the simulation algorithm by implementing a heap as a data structure used for the ordering of the reactions.

We argue that our nondeterministic, discrete approach has an advantage over methods based on ordinary differential equations, such as the ODE method found in [8]. Ordinary differential equations are an appropriate modeling technique when one seeks the average behavior of a system involving large numbers of elements – for example, modeling the dynamics of large cell populations. However, when modeling signal cascades within a single cell, we believe that differential equations may not be the best approach. We feel that a discrete method is better equipped to simulate processes that involve relatively low numbers of molecules/objects.

For example, assume that two reactions are competing over a single molecule. A differential equations technique allows both reactions to be applied, borrowing a fraction of the molecule to satisfy each equation. However, our discrete method maintains molecular integrity. In other words we do not allow a single molecule to be partially used in two reactions. The molecule is used for one reaction or the other. Our nondeterministic logic can affect the entire evolution of the system, changing the results of the signaling cascade in a way ODEs do not.

The second improvement over [2] is the use of a min-heap to store reactions. Initially, we create the min-heap in the standard (bottom-up) way. However, during the simulation our heap obeys two properties, which has led us to develop a non-standard approach to maintaining the min-heap property. The properties are the following:

(i) once intialized, the heap does not grow or shrink;
(ii) multiple nodes throughout the heap will be updated simultaneously.

In [3] the authors provide a nonstandard method for updating a min-heap. However, the min-heap they maintain does not obey (ii), so we have developed our own special maintenance functions. We provide a full description of our heap maintenance functions in a later section.

In this paper we demonstrate the design of our Nondeterministic Waiting Time algorithm, as well as our results from simulating the FAS-induced apoptotic signaling cascade. Section 2 provides a full description of our algorithm and pseudocode for our heap maintenance functions. Section 3 presents results from our simulation of the FAS-induced apoptotic pathway, along with comparisons to an established ordinary differential equations method and some experimental data. Section 4 describes a stochastic method, based on the ideas of the well-known Gillespie Algorithm. Section 5 is a discussion section providing ideas for future work.

We note that we will use the words rule and reaction interchangeably throughout this paper. In the context of Membrane Systems, we have rules associated to symbols, and in the context of cells we have reactions associated with proteins/molecules.

2 Membrane System For Simulating Signal Cascade

In the following we will give a brief description of the simulation technique proposed and then we will provide the actual algorithm with a discussion on its main ideas and an example for better understanding.

2.1 Explanation of Discrete Method

Our Membrane System follows the evolution of molecular multiplicities over time. We simulate individual chemical reactions which are asynchronous and occur discretely over different lengths of time. The rules of the our system obey the *Law of Mass Action*: the reaction rate depends proportionally on the concentrations of the reactants.

Every reaction r has an associated reaction rate constant k_r. For each reaction r we pre-compute a kinetic constant, $const_r$: $const_r = \frac{k_r}{V^{i-1} \times N^{i-1}}$ where V is the volume of the system, N is Avogadro's constant (6.0221415×10^{23}) and i is the number of reactants involved in the reaction. Consider a general second order reaction, $r_1 : A+B \rightarrow C$. We compute the amount of time required for a single application of the reaction: $wt_{r_1} = \frac{1}{const_{r_1} * |A| * |B|}$ where $|A|$ and $|B|$ represent the number of molecules of the two reactants. Or, consider a first order reaction, $r_2 : D \rightarrow E$. A single application of of the reaction is computed: $wt_{r_2} = \frac{1}{const_{r_2} * |D|}$. We refer to this as the calculation of the 'Waiting Time' (WT) of a reaction.

2.2 Nondeterministic Waiting Time Algorithm

We now provide the pseudocode of our algorithm:

1. *Initialization:* Calculate the waiting time for every reaction in the system, and label it 'WT' (time required for reaction to occur). Store the reactions in a min-heap (sorted by 'WT'). Set simulation time to zero ($t = 0$).
2. *Select Rule:* Select the reaction with the lowest waiting time. If there is a tie, go to step 3. If not, proceed to step 4.
3. *Handle Tie:* If there are enough reactants to satisfy all reactions in the tie, implement all reactions in step 4. If there are not enough reactants to accommodate all the reactions, use the nondeterministic logic to apply as many rules as possible.
4. *Apply Rule:* Update the multiplicities of the reactants and products involved in the reaction(s) from step 2. Aggregate the simulation time ($t = t$+WT).
5. *Update Rules:* Recalculate the waiting time for any reactions involving the reactants or products of the applied reaction(s). For each such reaction compare the new waiting time with the existing waiting time and keep the smallest of the two.
6. *Heap Maintenance:* Adjust the min-heap.

7. *Check Done:* If the desired simulation time has not been reached, go back to step 2.

We initially build our min-heap using the standard bottom-up technique. Once the top node of the heap has been selected (Step 2), applied (Step 4) and had its waiting time recalculated (Step 5), we leave it at the top of the tree. Often an applied reaction's new waiting time is close to its previous WT. Therefore, the node will most likely be located near the top of the heap once the heap is resorted. After recalculating the waiting time of all reactions involving reactants or products of the applied rule (Step 5), we are ready to reestablish our min-heap (Step 6). Next, we provide our special methods for heap maintenance:

```
1: Reheap()
2:     for each applied reaction (more than one if tie exists)
3:         for each product of the applied reaction
4:             for each reaction the product is a reactant of
5:                 CheckUp(reaction);
6:         for each reactant of the applied reaction
7:             for each reaction the reactant is a reactant of
8:                 CheckDown(reaction)
```

```
1: CheckUp(node r)
2:     If parent(r) exists
3:         If parent(r) > r
4:             Swap(r, parent(r))
5:             while parent(r) > children(parent(r))
6:                 Swap(parent(r), children(parent(r)))
7:             CheckUp(r)
```

```
1: CheckDown(node r)
2:     If children(r) exist(s)
3:         If r > children(r)
4:             Swap(smallestchild(r), r)
5:             while parent(smallestchild(r)) > smallestchild(r))
6:                 Swap(parent(smallestchild(r)), smallestchild(r))
7:             CheckDown(r)
```

Let us consider an applied rule of the form: A+B→C+D. When the rule is applied the system loses a molecule of A and one of B, and it also gains a molecule of C and one of D. Therefore, any reaction that has A, B, C, or D as a reactant requires a recalculation of its waiting time. After recalculation, the Reheap method is called, and the position of every reaction with a recalculated waiting time is checked. Since ties are usually infrequent, the **for** loop on line 2 of the Reheap method runs for very few iterations. Also, most reactions have at most two products and at most two reactants. Hence, the **for** loops on line 3 and 4 combine for a total of at most four iterations.

Our CheckUp and CheckDown methods differ from previous nonstandard heap maintenance [3]. Since we update multiple nodes in our heap simultaneously, we can have many nodes in violation of the min-heap property. We refer to Fig. 1 for an

example of our heap maintenance. In Fig. 1 we see three nodes in a heap of arbitrary size. Fig. 1(I) shows a clear violation of the min-heap property. Assume that the yellow and blue nodes have both just been recalculated (and their waiting time decreased), and also assume that green has not been recalculated. Our algorithm does not waste any clock cycles sorting the reactions with changed waiting times, so we can assume that CheckUp is first called on the blue node. Blue is compared with its parent (yellow), and since yellow has a smaller WT than blue, no changes to the heap are made and the CheckUp method for blue terminates.

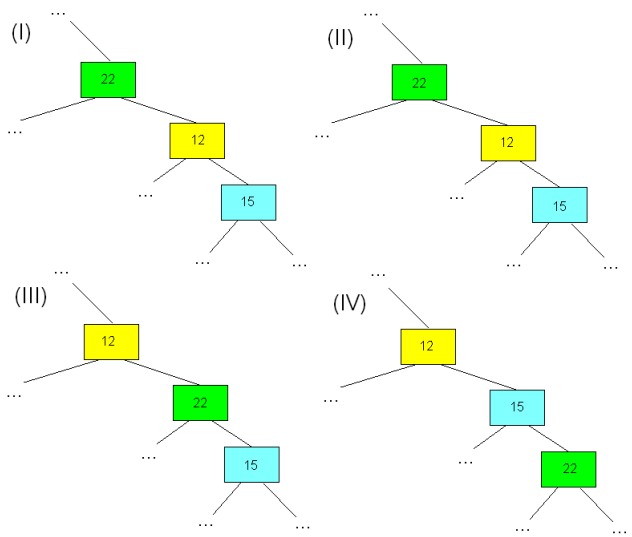


Fig. 1. (I) The waiting times for yellow and blue are recalculated, and the heap is now unsorted. (II) The method CheckUp is called for blue, but the node does not move up, since yellow has a smaller WT. (III) The method CheckUp is called for yellow. Yellow switches places with green. Before yellow checks up again, green must attempt to move down. (IV) The green node and swaps with the blue node, which satisfies the min-heap property. Green will move down as far as it can. Afterwards, CheckUp is called on the yellow node, and the process repeats until the heap is resorted.

Next, CheckUp is called for the yellow node. Yellow and green are compared, and because they violate the min-heap property, the two nodes are swapped. Since yellow and green have swapped places, the green node must be moved down the tree as far as possible in accordance with the min-heap property. Green and its children (one of which is blue) are compared, and since there is a violation of the min-heap property, green is swapped with its smallest child (assumed to be blue, but it could be the other child). Green is moved down as far as necessary, bringing up the 'tail' of yellow. Once green is in a position that does not violate the min-heap property, CheckUp is called again for yellow (line 7 of CheckUp method), and the process is repeated until yellow fails to move up. Once CheckUp and CheckDown have been called for all reactions with recalculated waiting times, the heap will be sorted.

The implementation of our heap yielded a massive performance increase over the previous algorithm from [2]. With the algorithm described in [2] we were able to complete the FAS-induced simulation ($\sim$8 cell simulated hours) in 30-40 minutes de-

pending on the particular rule set and molecular multiplicities. Our new algorithm, utilizing the efficient heap structure, takes 3-4 minutes to run the same simulations. While incorporating the heap structure we not only increased the sorting performance, we were able to eliminate an extraneous **for** loop used to put waiting times in the context of simulation times (a loop for every reaction for each applied rule). Our previous simulator had a runtime of $O(n^2 \log n)$. To be able to give the complexity of the algorithm proposed and show that in this specific case is efficient, we need to make several assumptions which are (usually) valid for the signalling cascades:
a) each reaction involves at most 5 different species of molecules
b) there are a bounded number of reactions having the same reactant (usually 3, at most 5)
c) there are not many reactions happening at the same time (due to the differences in molecule multiplicities and reaction rates).
From a), b) and c) we can now show that our new implementation has a runtime of $O(n \log n)$ with respect to the number of reactions simulated.

Next, we will describe the use of our technique in modeling the FAS-mediated signaling cascade.

3 FAS-induced apoptosis

We have chosen to simulate FAS-induced apoptosis because it has one of the most detailed descriptions/characterization in the literature (due in large part to its role in cancer and HIV research). In the interest of comparing our Membrane System with an established ordinary differential equations (ODE) technique, we have implemented 101 different rules working on 53 distinct proteins and protein complexes. The pathway begins with the stimulation of FASL and ends with the activation of the effector Caspase-3. Fei Hua et al., in [8], provide the results of an ODE simulation, as well as some experimental data (from the Jurkat cell line), which they used to fit their model.

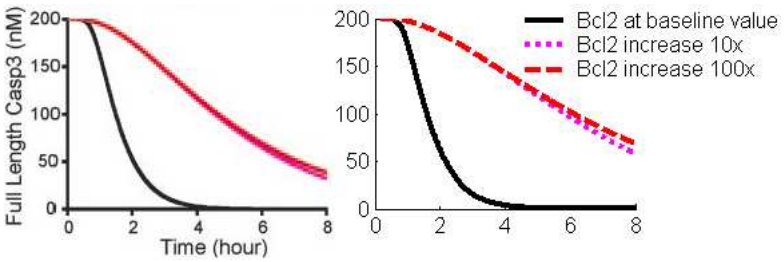


Fig. 2. On the right is the Membrane System and on the left is the ODE simulation (used with permission [8]). The two graphs show the decline over time of full length Caspase-3, which means there is an increase in active Caspase-3. In both simulators, we see that Caspase-3 nears zero after four hours for the baseline Bcl-2 concentration. However, apoptosis is inhibited as Bcl-2 levels are increased by 10- or 100-fold.

3.1 Results of Discrete Method

Similar to [8], we ran our simulation with three different initial concentrations of Bcl-2: the baseline value (75nMs), an increase by 10x (750nMs), and an increase by 100x (7500nMs). Assuming a cell volume of 10^{-12} liters, we convert the concentrations into molecular multiplicities: baseline value (45166 molecules), 10x (451660 molecules), and 100x (4516606 molecules). We expect to see a decline in Caspase-3 activation as Bcl-2 concentration is increased by 10x and 100x; we provide the results of our simulation in (Fig. 2). We also provide the results of a simulation with a decrease of 10x and 100x in comparison to the baseline Bcl-2 multiplicity (Fig. 3). Notice, the graphs based on our Membrane System simulations are comparable to the ODE results from [8].

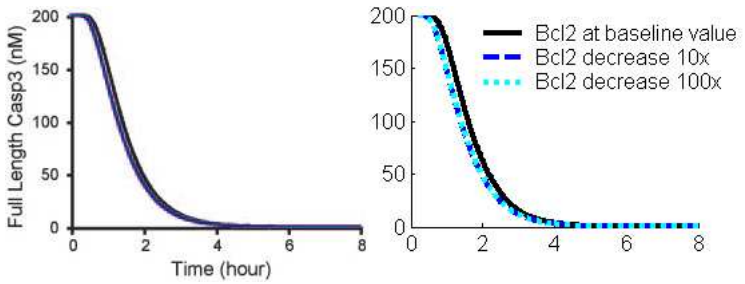


Fig. 3. On the right is the Membrane System and on the left is the ODE simulation (used with permission [8]). These results illustrate the models insensitivity to decreasing Bcl-2 concentrations by 10- and 100-fold. We see agreement between the two different simulators.

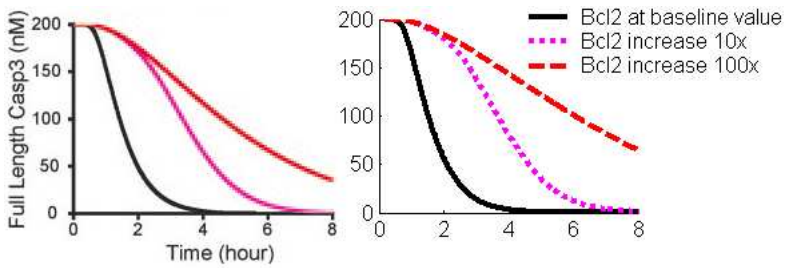


Fig. 4. On the right is the Membrane System and on the left is the ODE simulation (used with permission [8]). Here we see the effects of Bcl-2 binding to Bax only. It seems that Bcl-2 binding only with Bax is the second most effective method for blocking the apoptotic pathway.

3.2 Bcl-2's effects on the Type II pathway

We now analyze the Caspase-3 activation kinetics by considering different mechanisms through which Bcl-2 can block the type II pathway. In [24], [14], or [1] the

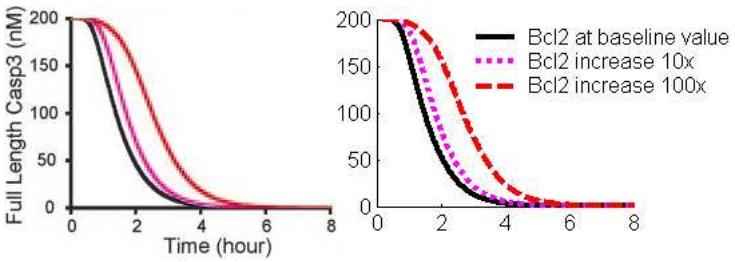


Fig. 5. On the right is the Membrane System and on the left is the ODE simulation (used with permission [8]). These two graphs show Bcl-2 binding to tBid only. We see considerably less inhibition of the pathway in this mechanism. The release of Cytochrome c is contingent on the binding of one tBid to two Bax molecules, and thus, blocking Bax would be more effective than blocking tBid.

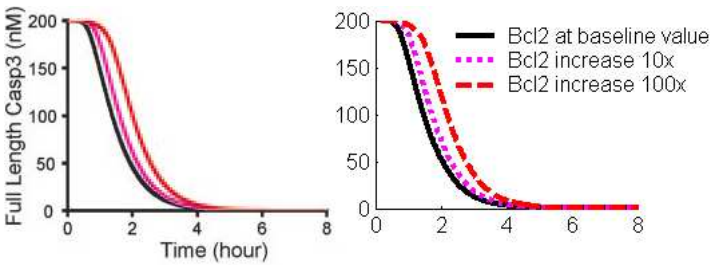


Fig. 6. On the right is the Membrane System and on the left is the ODE simulation (used with permission [8]). We see very little change in apoptotic behavior when Bcl2 is allowed to bind only with Bid. This is because Bcl-2 binding to Bid is a reversible reaction, and once Bid is truncated to tBid, it is no longer available to bind with Bcl-2.

authors suggest that Bcl-2 might bind with (a) Bax, (b) Bid, (c) tBid, or (d) bind to both Bax and tBid to block the mitochondrial pathway. We have implemented in our Membrane System four different sets of rules to test each Bcl-2 binding mechanisms. We refer the interested reader to [8] for the details of the rules. The dynamics of Caspase-3 activation are studied by varying the Bcl-2 concentration 10x and 100x the baseline value. The conclusion of [8] is that Bcl-2 binding to both Bax and tBid (d) is the most efficient mechanism for inhibiting apoptosis. Our Membrane System yields results that are in agreement with the observations from [8]. The results of (d) are illustrated in Fig. 2, and (a) - (c) can be seen in Fig. 4 - Fig. 6. A comparison of (a) - (d) at baseline Bcl-2 concentration is shown in Fig. 7.

Modeling the Behavior of the Type I Pathway

There are cells which are not sensitive to Bcl-2 over expression as described in [20]. In these cells, Caspase-3 is activated through the type I pathway, bypassing the role of the mitochondria and Bcl-2. Scaffidi et al. have suggested in [20] that the type of pathway is chosen based on the concentration of Caspase-8 generated in active form following FASL binding. High concentration of active Caspase-8 allows for direct activation of Caspase-3 (type I), but if the concentration of Caspase-8 is sufficiently low, amplification of the death signal through the mitochondria is required to induce

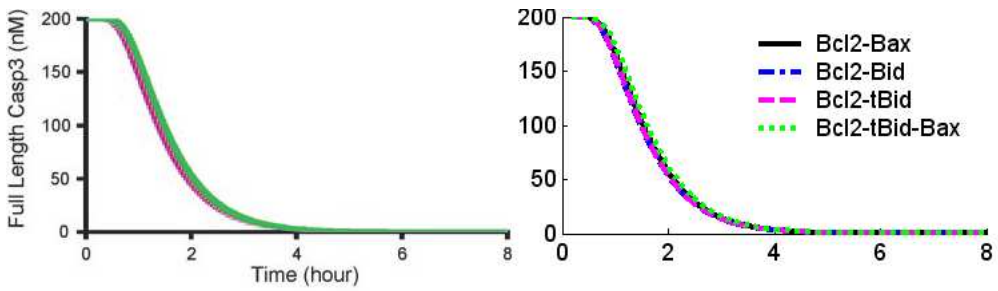


Fig. 7. On the right is the Membrane System and on the left is the ODE simulation (used with permission [8]). At baseline Bcl-2 concentration, each of the four mechanisms for Bcl-2 inhibition are similar. Both simulators agree across all rule versions: Bcl-2 binding with Bax only, Bid only, tBid only, or Bax and tBid.

the cell death (type II). We test this hypothesis by increasing the initial concentration of Caspase-8 by 20x (from 33.33nMs to 666.6nMs), which should lead to increased active Caspase-8 throughout the simulation run. In two different runs of increased Caspase-8 concentration, the baseline concentration of Bcl-2 is used and an increase of 100x, testing the sensitivity of Caspase-3 activation to Bcl-2. Fig. 8 shows the Caspase-3 activation is not sensitive to the increase in Bcl-2 concentration, which is the hallmark for type I pathway dominant behavior. N.B., the binding mechanism chosen for this simulation is Bcl-2 to Bax and tBid, which was shown above to be the most efficient mechanism for Bcl-2 inhibition of apoptosis.

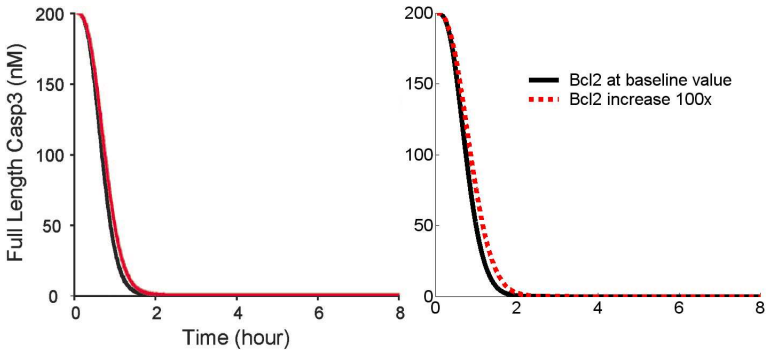


Fig. 8. On the right is the Membrane System and on the left is the ODE simulation (used with permission [8]. Unlike the type I pathway, the type II pathway is unaffected by a 100-fold Bcl-2 increase. We are pleased with these results, as Bcl-2 acts to block the release of Cytochrome c, which is an unnecessary molecule in this pathway.

We will now describe another method of simulating signal cascades, using a strategy that is based on the well known Gillespie's algorithm, but running on more than one compartment. It is called *Multi-compartmental Gillespie Algorithm.*

4 Multi-Compartmental Gillespie Algorithm

Gillespie's algorithm [4] (see also [6, 7] for some recent improvements) provides an exact method for the stochastic simulation of systems of bio-chemical reactions; the validity of the method is rigorously proved and it has already been successfully used to simulate various biochemical processes [13]. Moreover the Gillespie algorithm is used in the implementation of stochastic π-calculus [16, 22], and in its application to the modeling of biological systems [17]. The extension of the classical Gillespie algorithm, called the *Multi-compartmental Gillespie Algorithm*, is first described in [18]. Below we provide the general definition of the Membrane System.

Let $\Pi = (O, Lab, \mu, M_1, M_2, \ldots, M_n, R_1, \ldots, R_n)$ be a Membrane System with the membranes $M_i = (w_i, L_i)$ and the programs R_i, $1 \leq i \leq n$. Each set R_i of programs are active inside their corresponding membrane i. These sets contain elements of the form $(j, \pi_j, r_j, p_j, k_j)$ where:

- j is the index of a program from R_i;
- π_j is the predicate; in this section this will be always true and will be omitted;
- r_j is the boundary rule contained in the program j;
- p_j is the probability of the rule contained in the program j to be applied in the next step of evolution; this probability is computed by multiplying a stochastic constant k_j, specifically associated with program j, by the number of possible combinations of the objects present on the left side of the rules with respect to the multiset w_i (or the multiset $w_{i'}$, with $i' = upper(\mu, i)$) - the current content of membrane i (i').

First, each membrane i will be considered to be a compartment enclosing a volume, therefore the index of the next program to be used inside membrane i and its waiting time will be computed using the classical Gillespie algorithm which we recall below:

1. calculate $a_0 = \sum p_j$, for all $(j, r_j, p_j, k_j) \in R_i$;
2. generate two random numbers r_1 and r_2 uniformly distributed over the unit interval $(0, 1)$;
3. calculate the waiting time for the next reaction as $\tau_i = \frac{1}{a_0} ln(\frac{1}{r_1})$;
4. take the index j, of the program such that $\sum_{k=1}^{j-1} p_k < r_2 a_0 \leq \sum_{k=1}^{j} p_k$;
5. return the triple (τ_i, j, i).

Notice that the larger the stochastic constant of a rule and the number of occurrences of the objects placed on the left side of the rule inside a membrane are, the greater the chance that a given rule will be applied in the next step of the simulation. There is no constant time-step in the simulation. The time-step is determined in every iteration and it takes different values depending on the configuration of the system.

Next, the *Multi-compartmental Gillespie's Algorithm* is described in detail:

1. *Initialization:*

- set time of the simulation $t = 0$;
- for each membrane i in μ compute a triple (τ_i, j, i) by using the procedure described above; construct a list containing all such triples;
- sort the list of triples (τ_i, j, i) according to τ_i;

2. *Iteration:*
   - extract the first triple, (τ_m, j, m) from the list;
   - set time of the simulation $t = t + \tau_m$;
   - update the waiting time for the rest of the triples in the list by subtracting τ_m;
   - apply the rule contained in the program j only once changing the number of objects in the membranes affected by the application of the rule;
   - for each membrane m' affected by the application of the rule remove the corresponding triple $(\tau'_{m'}, j', m')$ from the list;
   - for each membrane m' affected by the application of the rule j re-run the Gillespie algorithm for the new context in m' to obtain $(\tau''_{m'}, j'', m')$, the next program j'', to be used inside membrane m' and its waiting time $\tau''_{m'}$;
   - add the new triples $(\tau''_{m'}, j'', m')$ in the list and sort this list according to each waiting time and iterate the process.
3. *Termination:*
   - Terminate simulation when time of the simulation t reaches or exceeds a preset maximal time of simulation.

Therefore, in this approach, the waiting time computed by the Gillespie algorithm is used to select the membranes which are allowed to evolve in the next step of computation. Specifically, in each step, the membranes associated to programs with the same minimal waiting time are selected to evolve by means of the corresponding rules. Moreover, since the application of a rule can affect more than one membrane at the same time (e.g., some objects may be moved from one place to another), we need to reconsider a new program for each one of these membranes by taking into account the new distribution of objects inside them. Note that in this point our approach differs from [21] where only one program is applied at each step without taking into account the rest of the programs that are waiting to be applied in the other membranes, neither it is considered the disruption that the application of one program can produce in various membranes.

We coded the model in C and simulated the FAS-induced apoptotic pathway described in Section 3. In the interest of saving space, we provide only some of the stochastic simulation results in our discussion section.

5 Discussion And Final Remarks

We have provided two discrete methods for modeling molecular signaling cascades, highlighting key changes to our previous technique. We also gave the results from the simulation of the FAS-mediated apoptotic pathway. Our Membrane System has yielded comparable results to an ordinary differential equations technique. The sixteen distinct simulations reach apoptosis at similar rates to the ODE method (as shown in Fig. 2 through Fig. 8). Although the activation of Caspase-3 is similar

between the two techniques, the molecular interactions throughout are different. We have compared the results of our two simulators with the experimental results obtained by Fei Hua et al. The Caspase-3 results are not very surprising, but the activation of Caspase-8 raises our interest. Refer to Fig. 9 for a comparison of the three simulation methods. We note the fact that in the case of the stocastic system we have depicted a single run of the simulation.

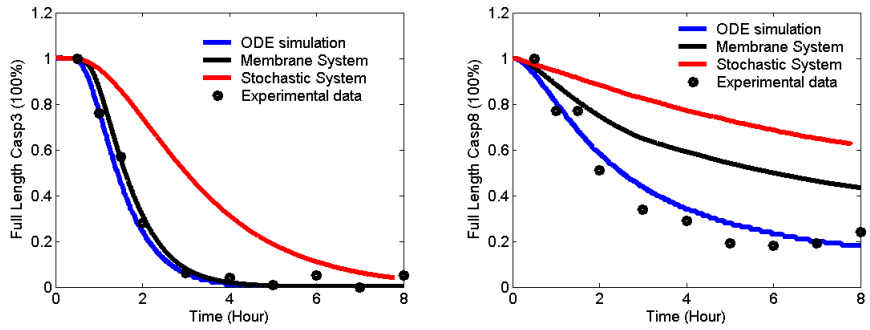


Fig. 9. The experimental data and ODE simulation results were obtained from Fei Hua et al. to be used in comparison to our Membrane System simulator. In both simulators, Bcl-2 is binding with Bax and tBid. We see that the decrease of full length Caspase-3 is similar in all three results. Interestingly, the decline of full length Caspase-8 is least prominent in the Membrane System simulator. The contrast could be a result of the discrete nature of our system. As for both results being different than the experimental data, we believe that further investigation of kinetic rates of the reactions will allow for better agreement between simulation and experimentation.

The consistency between the framework and the experimental results in the paper [8] validates our model. We have stated that our discrete methods handle low levels of molecules in a different way that ODE techniques. To further investigate the differences between discrete and ODE methods, we have chosen to focus on one rule from the FAS-mediated pathway (a transformation): $[\, CASP8_2^{P41} \,]_c \rightarrow [\, CASP8_2^{*} \,]_c$. The initial concentration of $CASP8_2^{P41}$ is assumed to be $.03nMs$ (or 18 molecules) and we use the same kinetic rate $k_5 = 0.1s^{-1}$ as in the model used in the current paper.

The ode45 method in MATLAB was used to obtain the results for the ODE technique, and is shown as the blue line in the Fig. 10. The membrane system was used to produce the discrete results depicted in the dark blue bars. Fig. 10, clearly shows a divergence between the two techniques. At the end of the simulation (second 16) we notice that the ODE has the value approximately 14 whereas the membrane simulator has the value 18. These results are obtained for the exact same constants and reaction, leading to a difference greater than 20% of the end value. We suggest to the interested reader to contemplate what would be the effects of similar differences in protein multiplicities/concentrations when considering hundreds or thousands of rules in the model being simulated. It should be obvious now why the discrete simulation techniques are producing different results than the continuous simulations. We believe that when modeling signaling cascades over a relatively

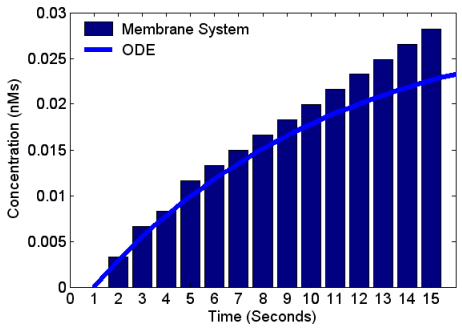


Fig. 10. The Membrane System does not allow the existence of fractional molecules, which makes it fundamentally similar to the real world. Since ODEs use concentrations for simulation, the data can become skewed as the simulation commences - i.e., fractions of molecules are interacting. If the concentration is sufficiently high, this may not affect the quality of results, but as we can see here, deviations occur when dealing with evolution of multiplicities rather than concentrations.

small number of molecules, the discrete methodology may yield better/more-realistic results than an ODE technique.

Our Nondeterministic Waiting Time algorithm shows that Membrane Systems are an intriguing alternative to ordinary differential equations methods. We have argued that the discrete nature of our technique might be better for simulating the evolution of systems involving low numbers of molecules. In the future, we would like to add a stochastic element to the Nondeterministic Waiting Time algorithm, allowing for limited stochasticity in our simulations. We will also be applying our model to signal cascades other than FAS-mediated apoptosis. Another future thrust of our group will be in the extension and refinement of the pathway discussed here. We plan to model the behavior of the caspase-based apoptotic pathway in the HIV infected cells.

Acknowledgements

J. Jack gratefully acknowledges a Ph.D. fellowship from the University of Louisiana System and support from NSF Grant CCF-0523572. The research of O. H. Ibarra was supported in part by NSF Grants NSF Grants CCF-0430945 and CCF-0524136. The research of A. Păun was supported in part by LA BoR RSC grant LEQSF (2004-07)-RD-A-23 and NSF Grants IMR-0414903 and CCF-0523572.

References

1. Cheng, E.H., Wei, M.C., Weiler, S., Flavell, R.A., Mak, T.W., Lindsten, T., Korsmeyer, S.J. (2001). BCL-2, BCL-XL sequester BH3 domain-only molecules preventing BAX- and BAK-mediated mitochondrial apoptosis. *Molecular Cell*, **8**, 705–711.
2. Cheruku, S., Păun, A., Romero-Campero, F., Pérez-Jiménez, M., Ibarra, O. (2006). Simulating FAS-Induced Apoptosis by Using P Systems. *Proceedings of Bio-inspired computing: theory and applications (BIC-TA)* September 18-22, 2006, Wuhan, China, also extended version accepted to *Progress in Natural Science*, **17**(4), 424–431.

3. Gibson, M. A., Bruck, J. (2000). Efficient Exact Stochastic Simulation of Chemical Systems with Many Species and Many Channels. *J.* Phys. Chem. **A 104**, 1876–1889.
4. Gillespie, D.T. (1976). A General Method for Numerically Simulating the Stochastic Time Evolution of Coupled Chemical Reactions. *Journal of Computational Physics*, **22**, 403–434.
5. Gillespie, D.T. (1977). Exact Stochastic Simulation of Coupled Chemical Reactions. *The Journal of Physical Chemistry*, **81**, 25, 2340–2361.
6. Gillespie, D.T. (2001). Approximate Accelerated Stochastic Simulation of Chemically Reacting Systems. *Journal of Chemical Physics*, **115**, 4, 1716–1733.
7. Gillespie, D.T. (2003). Improved Leap-size Selection for Accelerated Stochastic Simulation. *Journal of Chemical Physics*, **119**, 16, 8229–8234.
8. Hua, F., Cornejo, M., Cardone, M., Stokes, C., Lauffenburger, D. (2005). Effects of Bcl-2 Levels on FAS Signaling-Induced Caspase-3 Activation: Molecular Genetic Tests of Computational Model Predictions. *The Journal of Immunology*, **175**, 2, 985–995 and correction **175**, 9, 6235–6237.
9. Ibarra, O.H., Păun, A. (2005). Counting time in computing with cells. *Proceedings of DNA Based Computing, DNA11*, London, Ontario, 25–36 and *Lecture Notes in Computer Science*, **3892**, (2006), 112–128.
10. Krammer, P.H. (2000). CD95's deadly mission in the immune system. *Nature*, **407**, 789–795.
11. Krammer, P.H., (2000). CD95's deadly mission the in immune system. *Nature*, **407**, 789–795.
12. Manca, V., Bianco, L., Fontana, F. (2005). Evolution and Oscillation in P Systems: Applications to Biological Phenomena, *Lecture Notes in Computer Science*, **3365**, 63 – 84.
13. Meng, T.C., Somani S., Dhar, P. (2004). Modelling and Simulation of Biological Systems with Stochasticity. *In Silico Biology*, **4**, 3, 293–309.
14. Oltavi, Z.N., Milliman, C.L., Korsmeyer, S.J. (1993). Bcl-2 heterodimerizes in vivo with a conserved homolog, Bax, that accelerates programmed cell death. *Cell*, **74**, 4, 609–619.
15. Păun A., Pérez-Jiménez M., Romero-Campero F. (2006). Modelling Signal Transduction using P Systems, *Lecture Notes in Computer Science*, **4361**, 100–122.
16. Philips, A., Cardelli. L. (2004). A Correct Abstract Machine for the Stochastic Pi-calculus. *Proc. Bioconcur04*. ENTCS.
17. Priami, C., Regev, A., Shapiro, E., Silverman, W. (2001). Application of a Stochastic Name-Passing Calculus to Representation and Simulation of Molecular Processes. *Information Processing Letters*, **80**, 25–31.
18. Pérez-Jiménez, M.J., Romero-Campero, F.J. (2006) P Systems, a New Computational Modeling Tool for Systems Biology, *Transactions on Computational Systems Biology*, **4220**, 176–197.
19. Romero-Campero, F.J., Pérez-Jiménez, M.J. (2005). A Study of the Robustness of the EGFR Signalling Cascade using Continuous Membrane Systems. *Lecture Notes in Computer Science*, **3561**, 268 – 278.
20. Scaffidi, C., Fulda. S., Srinivasan, A., Friesen, C., Li, F., Tomaselli, K.J., Debatin, K.M., Krammer, P.H., Peter, M.E. (1998). Two CD95 (APO-1/Fas) signaling pathways. *The Embo Journal*, **17**, 1675–1687.
21. Stundzia, A.B., Lumsden, C.J. (1996). Stochastic Simulation of Coupled Reaction-Diffusion Processes. *Journal of Computational Physics*, **127**, 196–207.
22. The Stochastic Pi-Machine: `http://www.doc.ic.ac.uk/~anp/spim/`.
23. Van Kampen, N.G. (1992) Stochastic Processes in Physics and Chemistry. Elsevier Science B. V., Amsterdam, The Netherlands.
24. Wang, K., Yin, X.M., Chao, D.T., Milliman, C.L., Korsmeyer, S.J. (1996). BID: a novel BH3 domain-only death agonist. *Genes & Development*, **10**, 2859–2869.

Watson-Crick bordered words and their syntactic monoid

Lila Kari and Kalpana Mahalingam

University of Western Ontario,
Department of Computer Science,
London, ON, Canada N6A 5B7
lila, kalpana@csd.uwo.ca

Abstract. *DNA strands that, mathematically speaking, are finite strings over the alphabet $\{A, G, C, T\}$ are used in DNA computing to encode information. Due to the fact that A is Watson-Crick complementary to T and G to C, DNA single strands that are Watson-Crick complementary can bind to each other or to themselves in either intended or unintended ways. One of the structures that is usually undesirable for biocomputation, since it makes the affected DNA string unavailable for future interactions, is the hairpin: If some subsequences of a DNA single string are complementary to each other, the string will bind to itself forming a hairpin-like structure. This paper studies a mathematical formalization of a particular case of hairpins, the Watson-Crick bordered words. A Watson-Crick bordered word is a word with the property that it has a prefix that is Watson-Crick complementary to its suffix. We namely study algebraic properties of Watson-Crick bordered and unbordered words. We also give a complete characterization of the syntactic monoid of the language consisting of all Watson-Crick bordered words over a given alphabet. Our results hold for the more general case where the Watson-Crick complement function is replaced by an arbitrary antimorphic involution.*

1 Introduction

The subject of this paper, Watson-Crick (WK) bordered words, is motivated by the practical requirements of DNA computing experiments. DNA strands can be viewed as finite strings over the alphabet $\{A, G, C, T\}$ and are used in DNA computing to encode information. Since A is Watson-Crick complementary to T and G to C, DNA single strands that are WK complementary can bind to each other or to themselves in either intended or unintended ways. One of these undesirable DNA secondary structures, the *hairpin*, is formed when the suffix of a DNA single strand is WK complementary to the prefix of the same DNA strand. A word with this property is called Watson-Crick bordered. Experimentally, DNA strands that are Watson-Crick bordered are to be avoided when encoding data on DNA strands, since the hairpin structures they form make them unavailable for biocomputations. Theoretically, Watson-Crick bordered words generalize the classical definition of a bordered word: A bordered word is one with the property that it has a prefix that equals its suffix, [20], [18].

If in a Watson-Crick bordered word over the DNA alphabet the prefix and its WK complementary suffix do not overlap, then the strand forms a hairpin structure such as the one shown in Fig 1. If, on the other hand, the prefix of such a word and the WK complement of one of its suffixes overlap, the DNA strand could bind with another copy of itself as shown in Fig 2. Both such bindings are potentially undesirable for DNA computing experiments and this paper investigates words that could potentially interact this way. Algebraic properties of other types of languages that avoid DNA sequences undesirable for DNA based computations, such as sticky-free languages, overhang-free languages and hairpin-free languages, have

been extensively studied in [2, 3, 5, 8, 9]. The notion of Watson-Crick bordered words was formalized and its coding properties as well as relations between Watson-Crick bordered words and other types of codes have been discussed in [11]. Certain algebraic properties of involution bordered words were discussed in [11]. In this paper we study the algebraic properties of the set of all Watson-Crick bordered words through their syntactic monoid.

Fig. 1. If a word u is Watson-Crick bordered and its WK borders do not overlap, the word u may stick to itself forming a simple hairpin loop, as shown above.

The reason for our choice of method of investigation is that the syntactic monoid approach to the study of a language has proved to be very fruitful in other cases. Algebraic characterizations of many classes of codes through their syntactic monoid have been extensively studied [6, 14–16, 19]. In [6], the author formulated a general characterization method of the syntactic monoid which applies to all classes of codes that can be defined in a certain way and hence results analogous to those of [16] can be obtained for a large variety of classes of codes. For more details on codes the reader is referred to [1, 7, 18].

More recently, in [10] we have discussed the syntactic monoid properties of the set of all hairpin-free words. In this paper we use these methods to study the algebraic properties of the set of all involution-bordered words. Throughout the paper we concentrate on an antimorphic involution θ such that $\theta(a) \neq a$ for all $a \in \Sigma$. Such a function is arguably an accurate mathematical formalization of the Watson-Crick DNA strand complementarity as it features its main properties: the fact that the WK complement of a DNA strand is the reverse (antimorphism property) complement (involution property) of the original strand. (An involution is a function θ such that θ^2 equals the identity.)

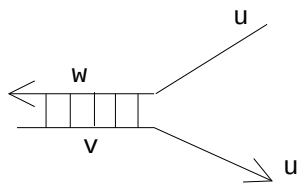


Fig. 2. If a word u is Watson-Crick bordered and its WK borders overlap, the word u may stick to another copy of itself as shown above. (Usually, in a DNA computing experiment, each DNA strand is present in hundreds or millions of copies in the solution.)

The paper is organized as follows: Section 2 reviews basic definitions. It is easy to see that, for an antimorphic involution, the set of all involution-bordered words is a proper subset of the set of all hairpins as studied in [10]. (Note that neither this

inclusion nor its reverse hold if we consider the set of general hairpins of a given length k). In [10] we showed that the elements of the syntactic monoid of the language of all hairpin-free words are idempotents and the monoid is commutative. In this paper (Section 3) we obtain a different result for involution-bordered word sets: We now show that, while all the elements of the syntactic monoid of the language of all involution-bordered words over a given alphabet are idempotents, the monoid is not commutative. We also observe that similarly to the case of the hairpin-free words, the language of all involution-bordered words is locally testable. Proposition 5 and 6 parallel results in [10] by giving a necessary and sufficient condition for a finite monoid to be the syntactic monoid of the set of all involution-bordered words over a given finite alphabet. In Section 4, we discuss the Green's relations for the set of all involution-bordered words. In contrast to the case of the set of all hairpin-free words, it turns out that the Green's relations are not trivial for the set of all involution-bordered words.

2 Definitions and basic concepts

In this section we review some basic notions. An alphabet set Σ is a finite non-empty set of symbols. A word u over Σ is a finite sequence of symbols in Σ. We denote by Σ^* the set of all words over Σ, and by Σ^+ the set of all non empty words over Σ. The empty word is denoted by λ. We note that with the concatenation operation on words, Σ^* is the free monoid and Σ^+ is the free semigroup generated by Σ. The length of a word $u = a_1...a_n$ is n for all $a_i \in \Sigma$ and is denoted by $|u|$. A language over Σ is an arbitrary subset of Σ^*. A mapping $\theta : \Sigma^* \mapsto \Sigma^*$ is called a morphism (antimorphism) of Σ^* if $\theta(uv) = \theta(u)\theta(v)$ (respectively $\theta(uv) = \theta(v)\theta(u)$) for all $u, v \in \Sigma^*$. An involution map θ is such that θ^2 equals identity.

Bordered words were initially called "overlapping words" and unbordered words were called as "non-overlapping words", [18]. For properties of bordered and unbordered words we refer the reader to [20], [18]. In [11], we extended the concept of bordered words to involution-bordered words and studied some of their algebraic properties. We now recall some definitions defined and used in [11].

Definition 1. *Let θ be either a morphic or an antimorphic involution on Σ^*.*

1. *A word $u \in \Sigma^+$ is said to be θ-bordered if there exists $v \in \Sigma^+$ such that $u = vx = y\theta(v)$ for some $x, y \in \Sigma^+$.*
2. *A non-empty word which is not θ-bordered is called θ-unbordered.*

Lemma 1 *Let θ be either morphic or an antimorphic involution.*

1. *A θ-bordered word $x \in \Sigma^+$ has length greater than or equal to 2.*
2. *For all $a \in \Sigma$, a is θ-unbordered.*
3. *For all $a \in \Sigma$ such that $a \neq \theta(a)$, a^n is θ-unbordered for all $n \geq 1$.*

In case θ is the Watson-Crick involution a θ-bordered word will be called Watson-Crick bordered, and a θ-unbordered word will be called Watson-Crick unbordered. Figures 1 and 2 illustrate some undesirable interactions that can result if a DNA string is Watson-Crick bordered.

We recall that a language or a set $X \subseteq \Sigma^*$ is said to be dense if for all $u \in \Sigma^*$, $X \cap \Sigma^* u \Sigma^* \neq \emptyset$. The following lemma was proved in [11].

Lemma 2 *Let θ be an antimorphic involution. Let L be the set of all θ-bordered words over Σ^*. Then*

1. *L is regular.*
2. *L is a dense set.*

3 The syntactic monoid of the set of all Watson-Crick bordered words

In the theory of codes, two types of syntactic monoids are usually considered, the syntactic monoid of the code itself and the syntactic monoid of the Kleene star of the code. In this section we concentrate on the characterizations of syntactic monoid of the set of all θ-bordered words, when θ is an antimorphic involution such that $\theta(a) \neq a$ for all $a \in \Sigma$. Necessary and sufficient conditions for a monoid to be the syntactic monoid of the set of all θ-bordered words are also discussed. We first review some basic concepts.

Let L be a language such that $L \subseteq \Sigma^+$. We define the context, right context and left context of a word $w \in \Sigma^*$ in L as follows:

- $C_L(w) = \{(u, v) : uwv \in L, u, v \in \Sigma^*\}$.
- $\mathcal{R}_L(w) = \{u \in \Sigma^* : wu \in L\}$.
- $\mathcal{L}_L(w) = \{u \in \Sigma^* : uw \in L\}$.

$C_L(w)$, $\mathcal{R}_L(w)$ and $\mathcal{L}_L(w)$ are called the context, right context and left context of w in L respectively. Also note that $Sub(L) = \{x : pxq \in L, p, q \in \Sigma^*\}$ is the set of all subwords of L. Recall that

Definition 2. *Let L be a language such that $L \subseteq \Sigma^+$.*

1. *The syntactic congruence of $L \subseteq \Sigma^+$ is denoted by P_L and is defined by $u \equiv v(P_L)$ iff $C_L(u) = C_L(v)$.*
2. *The syntactic monoid of L is the quotient monoid $M(L) = \Sigma^*/P_L$ with the operation $[x][y] = [xy]$, where for $x \in \Sigma^*$, $[x]$ denotes the P_L equivalence class of x.*

Let $W(L) = \{x \in \Sigma^* : C_L(x) = \emptyset\}$, i.e., $x \in W(L)$ iff $x \notin Sub(L)$. $W(L)$ is called the residue of L.

Note that if $W(L) \neq \emptyset$ then $W(L)$ represents a class for P_L and is the zero of $M(L)$.

Note that for a regular language L, $M(L)$ is the transition monoid (see [17]) of the minimal deterministic finite automaton (see [1, 17]) of L. The above definition of the syntactic congruence P_L can be defined for an arbitrary subset L of any semigroup S. If the syntactic congruence is the equality relation then we call the set L to be a disjunctive subset of S. If $L = \{x\}$ for some $x \in \Sigma^*$ and if P_L is the equality relation then we say that x is a disjunctive element of S. For more on syntactic monoid we refer the reader to [1, 12, 17].

It is a well known fact that L is a regular language if and only if $M(L)$ is finite (see [12, 17]). For any set L and its syntactic monoid $M(L)$, $\eta : \Sigma^* \to M(L)$ is the natural surjective syntactic morphism defined by $x \to [x]$. Note that for any L, L is a union of P_L classes.

We denote by $B_{\theta,\Sigma}$ the set of all θ-bordered words over Σ^*, with θ an antimorphic involution and $\theta(a) \neq a$ for all $a \in \Sigma$. In the remainder of the paper, if the alphabet Σ is clear from the context, we will denote the set of all θ-bordered words over Σ simply by B_θ.

It was shown in [11] that B_θ is regular and hence $Syn(B_\theta)$ is finite. In the following lemma we show that the residue of B_θ is the empty set.

Lemma 3 *The residue of B_θ is the empty set, i.e., $W(B_\theta) = \emptyset$.*

Proof. Follows from the fact that B_θ is dense, see Lemma 2. □

In the following proposition we show that every non zero element of $Syn(B_\theta)$ is idempotent.

Proposition 1 *For every $u \in \Sigma^*$, we have $u\ P_{B_\theta}\ u^2$.*

Proof. The congruence P_{B_θ} is equivalent to the congruence $P_{\overline{B_\theta}}$ associated to the complement $\overline{B_\theta}$ of B_θ. Hence we have to show that $u\ P_{\overline{B_\theta}}\ u^2$, i.e., $xuy \in \overline{B_\theta}$ iff $xu^2y \in \overline{B_\theta}$. Assume that $xuy \in \overline{B_\theta}$. Suppose that $xu^2y \in B_\theta$, then there exists $a \in \Sigma$ such that $xu^2y = av\theta(a)$ for some $v \in \Sigma^*$. We have the following cases:

1. If $x = ax_1$ and $y = y_1\theta(a)$ then $xuy = ax_1uy_1\theta(a)$, a contradiction since $xuy \in \overline{B_\theta}$.
2. If $x = \lambda$, the empty word, then $u^2y = av\theta(a)$ which implies $u = av_1$ and $y = y_1\theta(a)$ and hence $xuy = av_1y_1\theta(a)$, again a contradiction. The case when $y = \lambda$ is similar.
3. If both x and y are empty, i.e., $x = y = \lambda$, then $u^2 = av\theta(a)$. If $v = \lambda$, then $u = a = \theta(a)$ a contradiction to our assumption that $a \neq \theta(a)$ for all $a \in \Sigma$. Thus $v \neq \lambda$ and $u = av_1 = v_2\theta(a)$ a contradiction since $xuy = u \in \overline{B_\theta}$.

Hence $xu^2y \in \overline{B_\theta}$. Conversely, assume that $xu^2y \in \overline{B_\theta}$. Suppose $xuy \in B_\theta$, then there exists $a \in \Sigma$ such that $xuy = av\theta(a)$ for some $v \in \Sigma^*$. We have the following cases:

1. If $x = ax_1$ and $y = y_1\theta(a)$ then $xu^2y = ax_1u^2y_1\theta(a)$, a contradiction since $xu^2y \in \overline{B_\theta}$.
2. If $x = \lambda$, the empty word, then $uy = av\theta(a)$ which implies $u = av_1$ and $y = y_1\theta(a)$ and hence $xu^2y = av_1uy_1\theta(a)$, again a contradiction. The case when $y = \lambda$ is similar.
3. If both x and y are empty, i.e., $x = y = \lambda$, then $u = av\theta(a)$ which implies that $xu^2y = u^2 = av\theta(a)av\theta(a)$ again a contradiction since $xu^2y \in \overline{B_\theta}$.

Thus $xuy \in \overline{B_\theta}$ iff $xu^2y \in \overline{B_\theta}$ and hence $uP_{\overline{B_\theta}}u^2$ for all $u \in \Sigma^*$. □

Corollary 1 *The elements of the syntactic monoid of B_θ are idempotent elements.*

Proof. The fact that $uP_{\overline{B_\theta}}u^2$ for any $u \in \Sigma^*$ implies that $U = U^2$ for the class U containing u. □

If θ is a mapping of Σ^* into Σ^*, a congruence R is said to be θ-compatible if uRv implies $\theta(u)R\theta(v)$. If such is the case, then the mapping θ on Σ^* can be extended to a mapping of the quotient-monoid $S = \Sigma^*/R$ in the following way. Let U be the class mod R containing the word u. Define $\theta(U)$ to be the class of R containing $\theta(u)$. This mapping is well defined, i.e., it does not depend on the choice of the representative u of the class U. Indeed if $u' \in U$, then, R being θ-compatible, we have $\theta(u)R\theta(u')$ and hence $\theta(u') \in \theta(U)$.

Proposition 2 *The syntactic congruence P_{B_θ} is θ-compatible.*

Proof. To show that P_{B_θ} is θ-compatible, we have to show that $uP_{B_\theta}v$ implies $\theta(u)P_{B_\theta}\theta(v)$, i.e., $C_{B_\theta}(u) = C_{B_\theta}(v)$ implies $C_{B_\theta}(\theta(u)) = C_{B_\theta}(\theta(v))$. Let $uP_{B_\theta}v$ and let $(x, y) \in C_{B_\theta}(\theta(u))$, then $x\theta(u)y \in B_\theta$ which implies that $\theta(x\theta(u)y) \in \theta(B_\theta)$. Thus $\theta(y)\theta(\theta(u))\theta(x) \in \theta(B_\theta)$, .i.e., $\theta(y)u\theta(x) \in \theta(B_\theta)$. Since B_θ is θ stable, $\theta(B_\theta) \subseteq B_\theta$ and thus $\theta(y)u\theta(x) \in B_\theta$ iff $\theta(y)v\theta(x) \in B_\theta$ since $uP_{B_\theta}v$. Therefore $\theta(\theta(y)v\theta(x)) \in \theta(B_\theta) \subseteq B_\theta$ and therefore $x\theta(v)y \in B_\theta$ which implies that $(x, y) \in C_{B_\theta}(\theta(v))$. Similarly we can show that $C_{B_\theta}(\theta(v)) \subseteq C_{B_\theta}(\theta(u))$. Thus P_{B_θ} is θ-compatible. □

Recall that a semigroup in general is a set equipped with an internal associative operation which is usually written in a multiplicative form. A monoid is a semigroup with an identity element (usually denoted by e). If S is a semigroup, S^1 denotes the monoid equal to S if S has an identity element and to $S \cup \{e\}$ otherwise. In the latter case, the multiplication on S is extended by setting $s.e = e.s = s$ for all $s \in S$. Let $e \in S$ be an idempotent of S. Then the set $eSe = \{ese : s \in S\}$ is a subsemigroup of S, called the local subsemigroup associated with e. This semigroup is in fact a monoid, since e is an identity in eSe. We also recall that a semigroup S is called locally trivial if for all $s \in S$ and for all idempotents $e \in S$, we have $ese = e$. We recall the following result.

Proposition 3 *[17] Let S be a non empty semigroup. The following are equivalent.*

1. *S is locally trivial.*
2. *The set of all idempotents is the minimal ideal of S.*
3. *We have $esf = ef$ for all $s \in S$ and for all idempotents $e, f \in S$.*

Since for all $e \in Syn(B_\theta)$, e is an idempotent, we have the following observations. Let $S = Syn(B_\theta) \setminus \{1\}$, then

- S is aperiodic, i.e., for all $e \in S$, there exists n such that $e^n = e^{n+1}$.
- S is regular, i.e., for all $e \in S$, e is regular, i.e., there exists $s \in S$ such that $ese = e$.

Lemma 4 *For all $[ab] \in Syn(B_\theta)$, such that $a, b \in \Sigma$, $[ab]$ as a set is equal to the set of all words that begin with a and end with b.*

Proof. We first prove for the case when $a \neq b$. Clearly $ab \in [ab]$. Let $u \in \Sigma^*$ be such that $aub \notin [ab]$. Then there exists $x, y \in \Sigma^*$ such that $xaby \in B_\theta$ and $xauby \notin B_\theta$. Note that $xaby \in B_\theta$ implies that $xaby = cp\theta(c)$ for some $c \in \Sigma$ and $p \in \Sigma^*$. Then $xauby = cq\theta(c)$ which implies that $xauby \in B_\theta$ a contradiction. Hence $aub \in [ab]$ for all $u \in \Sigma^*$.
If $a = b$, then clearly we have $aa \in [aa]$ and for all $u \in \Sigma^*$, $aua \in [aa]$. Suppose $a \notin [aa]$ then there exists $x, y \in \Sigma^*$ such that $xaay \in B_\theta$ and $xay \notin B_\theta$. Note that $xaay \in B_\theta$ implies that $xaay = cp\theta(c)$ for some $c \in \Sigma$ and $p \in \Sigma^*$. If both x and y are non empty, then $xay = cq\theta(c)$ for some $c \in \Sigma$ and $q \in \Sigma^*$, which implies that $xay \in B_\theta$, which is a contradiction. If $x = \lambda$ and $y \in \Sigma^+$ then $aay = cp\theta(c)$ which implies $a = c$ and $y = y_1\theta(c)$ and hence $xay = ay = cy_1\theta(c)$ which implies that $xay \in B_\theta$, a contradiction. The case when $x \in \Sigma^+$ and $y = \lambda$ is similar. If $x = y = \lambda$, then $aa = cp\theta(c)$ which implies that $a = c = \theta(c)$ a contradiction to our assumption, since for all $a \in \Sigma$, $\theta(a) \neq a$. Hence $a \in [aa]$. Thus for all $a, b \in \Sigma$, and for all $[ab] \in Syn(B_\theta)$, $[ab]$ as a set is the set of all words that begin with a and end with b. □

Recall that a language L is said to be n-locally testable if whenever u and v have the same factors of length at most n and the same prefix and suffix of length $n-1$ and $u \in L$ then $v \in L$. The language L is locally testable if it is n-locally testable for some $n \in I\!N$.

We also recall a characterization of the syntactic semigroup of locally testable languages which states that (Proposition 2.1 in [13]) a recognizable subset (A language is called recognizable if there exists an algorithm that accepts a given string if and only if the string belongs to that language) L of Σ^+ is locally testable iff for all idempotents $g \in Syn(L)$, $gSyn(L)g$ is a semi lattice. We use this characterization and the above proposition to show that B_θ is locally testable.

Corollary 2 *B_θ is locally testable.*

Proof. We need to show that for all $e \in Syn(B_\theta)$, $eSyn(b)e$ is a semilattice. Note that from Lemma 4, for all $e, s \in Syn(B_\theta)$, $ese = e$ and hence $eSyn(B_\theta)e = \{e\}$. Since e is an idempotent and $\{e\}$ is commutative, $eSyn(B_\theta)e = \{e\}$ is a semilattice. Thus B_θ is locally testable.

Corollary 3 *$S = Syn(B_\theta) \setminus \{1\}$ is locally trivial.*

Proof. For all $e \in S$, e is an idempotent. We need to show that $ese = e$ for all $e, s \in S$. Let $e = [ab]$ for some $a, b \in \Sigma$ and let $s = [s_1]$ for some $s_1 \in \Sigma^+$. Then $ese = [ab][s_1][ab] = [abs_1ab] = [ab] = e$. Hence S is locally trivial. □

Corollary 4 *S is the minimal ideal of S and for all $e, s, f \in S$, $esf = ef$.*

Proof. Follows from the fact that S is locally trivial and all elements of S are idempotents and from Proposition 3. □

Corollary 5 *For all $e, f, g \in Syn(B_\theta)$, if $eg = fg$ and $ge = gf$ then $e = f$.*

Proof. Given that $eg = fg$ and $ge = gf$. Then $eg.ge = fg.gf$ which implies that $ege = fgf$ since for all $e \in Syn(B_\theta)$, e is an idempotent. Thus from Corollary 4, $ege = e^2 = e = fgf = f^2 = f$ which implies that $e = f$.

Corollary 6 *$Syn(B_\theta)$ is a simple semigroup.*

Proof. Since $\emptyset$ and $S = Syn(B_\theta)$ are the only ideals of $Syn(B_\theta)$, S is simple.

In the next proposition we show that for all $e, f \in S$, e and f are conjugates, i.e., $e = uv$ and $f = vu$ for some $u, v \in S$.

Proposition 4 *For all $e, f \in S$, e and f are conjugates.*

Proof. Let $e, f \in S$ such that $e = [ab]$ and $f = [cd]$ for some $a, b, c, d \in \Sigma$. Then $e = [ab] = [adcb] = [ad][cb]$ and $f = [cd] = [cbad] = [cb][ad]$ which implies that e and f are conjugates. □

Lemma 5 *P_{B_θ} class of 1 is trivial.*

Proof. Suppose not, let $u \equiv 1(P_{B_\theta})$ for some $u \in \Sigma^+$. Then for any $v \in B_\theta$, $uv \equiv v(P_{B_\theta})$ and $vu \equiv v(P_{B_\theta})$. Since $v \in B_\theta$, $uv, vu \in B_\theta$. Also, $v, uv, vu \in [ab]$ for some $a, b \in \Sigma$ with $\theta(a) = b$. Thus $v = axb$, $uv = ayb$ and $vu = azb$ for some $x, y, z \in \Sigma^*$. Then $u = arb$ which implies that $u \in [ab]$ and hence $1 \in [ab]$ a contradiction since $1 \notin B_\theta$. Thus P_{B_θ} class of 1 is trivial.

In the following results, using the notion of the syntactic monoid, similar to Proposition 17, 18 in [10], we establish an algebraic connection between the language B_θ of the bordered words relatively to an antimorphic involution θ over a finite alphabet Σ and a certain class of finite monoids.

Proposition 5 *Let $Syn(B_\theta)$ be the syntactic monoid of B_θ. Then:*

1. *$Syn(B_\theta)$ is a finite monoid which has no zero and every element of $Syn(B_\theta)$ is idempotent.*

2. *There exists an antimorphic involution ψ such that the set $Syn(B_\theta)$ is stable under ψ.*
3. *$Syn(B_\theta)$ has two non empty disjunctive sets D_1 and D_2 such that $Syn(B_\theta) = D_1 \cup D_2$ and $D_1 \cap D_2 = \emptyset$, where $D_1 = \{[x] \in Syn(B_\theta) \setminus \{1\} : \psi([x]) = [x]\}$.*

Proof. 1. The regularity of the language B_θ implies the finiteness of its syntactic monoid $Syn(B_\theta)$. Since B_θ is dense, $Syn(B_\theta)$ has no zero. The last part follows from Corollary 1.

2. Since the syntactic congruence P_{B_θ} is θ-compatible, an antimorphic involution ψ can be defined on $Syn(B_\theta)$ in the following way. Let U be an element of $Syn(B_\theta)$, i.e., U is a class of P_{B_θ}, and define $\psi(U)$ to be the class containing the element $\theta(u)$, where $u \in U$. This mapping is well defined because it does not depend on the choice of the representation v of the class U by virtue of θ-compatibility of P_{B_θ}. Indeed, since $uP_{B_\theta}v$, then $\theta(u)P_{B_\theta}\theta(v)$ and hence $\theta(v) \in \psi(U)$. Therefore if V is the class of P_{B_θ} containing v, then $\psi(U) = \psi(V)$. It is immediate that ψ is an antimorphism since θ is an antimorphism. To show that ψ is an involution, for all $U \in Syn(B_\theta)$, $\psi(\psi(U)) = U$. Note that $\psi(U) = [\theta(u)]$ for all $u \in U$. Thus $\psi(\psi(U)) = [\theta(\theta(u))] = [u] = U$ since θ is an involution. Thus ψ is an antimorphic involution. The last part follows from the fact that B_θ is θ-stale.
3. Let $D_1 = \{[x] \in Syn(B_\theta) : x \in B_\theta\}$ and let $D_2 = Syn(B_\theta) \setminus D_1 = \{[x] \in Syn(B_\theta) : x \in \overline{B_\theta}\}$. Let $[x] \in D_1$ which implies that $x \in B_\theta$ and thus $x = arb$ for some $a, b \in \Sigma$ and $r \in \Sigma^*$ with $\theta(a) = b$. Thus from Corollary 4, we have $\psi([x]) = \psi([arb]) = \psi([ab]) = [\theta(ab)] = [\theta(b)\theta(a)] = [ab] = [x]$. Thus for all $[x] \in D_1$, $\psi([x]) = [x]$. Now we show that D_1 is disjunctive. Suppose there exists $[x], [y] \in Syn(B_\theta)$ such that $C_{D_1}([x]) = C_{D_1}([y])$. Then $[\alpha][x][\beta] \in D_1$ iff $[\alpha][y][\beta] \in D_1$ for $[\alpha], [\beta] \in Syn(B_\theta)$ which implies that $[\alpha x \beta] \in D_1$ iff $[\alpha y \beta] \in D_1$. Thus for all $\alpha, \beta \in \Sigma^*$, $\alpha x \beta \in B_\theta$ iff $\alpha y \beta \in B_\theta$ which implies $C_{B_\theta}(x) = C_{B_\theta}(y)$ and hence $x, y \in [x] = [y]$. Hence D_1 is disjunctive. Since $P_{D_1} = P_{\overline{D_1}} = P_{D_2}$, D_2 is also disjunctive.

The next proposition is a converse of the Proposition 5.

Proposition 6 *Let M be a monoid with identity e and satisfy the following properties:*

1. *M is finite.*
2. *M has no zero.*
3. *Every element of M is an idempotent element.*
4. *There exists an antimorphic involution ψ such that M is stable under ψ.*
5. *M has two non empty disjunctive subsets D_1 and D_2 such that $D_1 = \{x \in M \setminus \{e\} : \psi(x) = x\}$ and $D_2 = M \setminus D_1$ and for all $x \in D_1$ there exists $p, q, r \in D_2$ such that $x = pq$ and either $\psi(p) = rq$ or $\psi(q) = pr$.*

Then there exists a free monoid Σ^* over a finite alphabet Σ, an antimorphic involution θ and a language B_θ in Σ^* such that,
(i) B_θ is the set of all θ-bordered words over Σ
(ii) The syntactic monoid $Syn(B_\theta) = \Sigma^*/P_{B_\theta}$ is isomorphic to M.

Proof. If $M = \{x_1, x_2, ..., x_n\}$, then take the elements of M as the letters of an alphabet $\Sigma = \{x_1, x_2, ..., x_n\}$ and let Σ^* be the free monoid generated by Σ^*. Let ϕ be the mapping of Σ^* onto M defined in the following way. If $u \in \Sigma$, then $\phi(u) = \psi(u) \in M$.

If $u = u_1u_2...u_k \in \Sigma^+$ with $u_i \in \Sigma$, then $\phi(u) = \psi(u) = \psi(u_k)...\psi(u_1)$. If $u = \lambda$, then $\phi(u) = e$, the identity of M. It is clear that ϕ is an antimorphism on Σ^* onto M. The relation ρ defined as $u\rho v$, $u, v \in \Sigma^*$ iff $\phi(u) = \phi(v)$ is a congruence of Σ^* and the quotient monoid Σ^*/ρ is isomorphic to M.
Let $B_\theta = \{x \in \Sigma^+ : \phi(x) = x\}$ and let P_{B_θ} be the syntactic congruence of B_θ. We need to show that $P_{B_\theta} = \rho$. We first show that $\rho \subseteq P_{B_\theta}$. Let $u\rho v$ then $\phi(u) = \phi(v)$. We need to show that $uP_{B_\theta}v$. Let $\alpha u\beta \in B_\theta$ which implies that $\phi(\alpha u\beta) = \alpha u\beta = \phi(\beta)\phi(u)\phi(\alpha) = \phi(\beta)\phi(v)\phi(\alpha)$ since $u\rho v$. Thus $\phi(\alpha u\beta) = \phi(\alpha v\beta) = \phi(\alpha u\beta)$ which implies $\alpha v\beta = \alpha u\beta$, since ϕ is an involution, it is bijective. Thus $\phi(\alpha v\beta) = \alpha v\beta$ which implies that $\alpha v\beta \in B_\theta$. Similarly we can show that $\alpha v\beta \in B_\theta$ and hence $\alpha u\beta \in B_\theta$. Thus $uP_{B_\theta}v$.
Conversely, we need to show that $P_{B_\theta} \subseteq \rho$. Let $uP_{B_\theta}v$. If u is not equivalent to v modulo ρ then $\phi(u) \neq \phi(v)$. M has a disjunctive D_1. Then syntactic congruence P_{D_1} is the equality relation and we have $C_{D_1}(\phi(u)) \neq C_{D_1}(\phi(v))$. This implies the existence if $\alpha, \beta \in M$ such that $\alpha\phi(u)\beta \in D_1$ and $\alpha\phi(v)\beta \notin D_1$ or $\alpha\phi(u)\beta \notin D_1$ and $\alpha\phi(v)\beta \in D_1$. Suppose that we have the first case, $\alpha\phi(u)\beta \in D_1$ and $\alpha\phi(v)\beta \notin D_1$, and since ϕ is bijective there exists $r, s \in \Sigma^*$ such that $\alpha = \phi(r)$, and $\beta = \phi(s)$. Thus $\alpha\phi(x)\beta = \phi(r)\phi(u)\phi(s) = \phi(sur) \in D_1$ and $\phi(svr) \notin D_1$, i.e., $\phi(sur) = sur$ and $\phi(svr) \neq svr$ which implies that $sur \in B_\theta$ and $svr \notin B_\theta$ a contradiction since $C_{B_\theta}(u) = C_{B_\theta}(v)$. Hence it follows that $P_{B_\theta} \subseteq \rho$.
We define the requested antimorphism θ of Σ^* by taking the corresponding permutation of the alphabet Σ and extending it to Σ^* in the usual way. If $u \in \Sigma^+$, $u = x_1x_2...x_n$ for $x_1, x_2, ..., x_n \in \Sigma$, then $\theta(u) = \theta(x_1x_2...x_n) = \theta(x_n)...\theta(x_1)$ and $\theta(\lambda) = \lambda$. It is immediate that θ is bijective antimorphism. Let us show now that conditions (i) and (ii) are satisfied.
For (i), let $u \in B_\theta$ and suppose that u is θ-unbordered. If $u \in B_\theta$ then $u = u_1u_2...u_k$ for some $u_i \in \Sigma$. Then if a word u is Watson-Crick bordered and its WK borders overlap, the word u may stick to another copy of itself as shown above. $\phi(u) = \phi(u_k)...\phi(u_1) = u_1...u_k$ which implies $\phi(u_k)\phi(u_1) = u_1u_k$ by Corollary 4. Thus $u_1 = u_k$. Hence $\phi(u) = \phi(u_1u_k) = \phi(u_1u_1) = \phi(u_1) = u = u_1u_1 = u_1$ since for all $f \in M$, f is an idempotent. Thus for all $u \in B_\theta$, $u = u_1$ for some $u_1 \in D_1$. Hence there exists $p, q, r \in D_2$ such that $u = pq$ with $\psi(p) = rq$ or $\psi(q) = pr$. Thus $u = pq$ implies $\psi(u) = \psi(q)\psi(p) = pr\psi(p)$ or $\psi(u) = \psi(q)rq$ which implies that u is θ-bordered. Suppose there exists a $u \in \Sigma^*$ such that u is θ-bordered and $u \notin B_\theta$, then $u = axb$ with $\theta(a) = b$ and $a, b \in \Sigma$. Thus $\psi(u) = \psi(axb) = \psi(ax\theta(a)) = \psi(\theta(a))\psi(x)\psi(a)$ $= \psi(\theta(a))\psi(a) = \psi(b)\psi(a) = ab = axb = u$. Thus $\psi(u) = u$ implies that $\phi(u) = u$ and $u \in B_\theta$.
Condition (ii) follows by construction. □

4 Green's relations for the set of all Watson-Crick bordered words

We recall here the definition of Green's relations and some well known facts about some of the relations. For extensive treatments of Green's relations and the related varieties of finite monoids, we refer the reader to [4, 12, 17]. In [10], it was shown that Green's relations are trivial for the language of all hairpin-free words. In contrast, this is not the case for the language of all involution-bordered words. Namely, in this section we show that $S = Syn(B_\theta) \setminus \{1\}$ is $\mathcal{H}$-trivial and S is not $\mathcal{K}$-trivial for all $\mathcal{K} \in \{\mathcal{D}, \mathcal{R}, \mathcal{L}, \mathcal{J}\}$.

Definition 3. (Green's Relations:) *Let S be a semigroup. We define on S four equivalence relations $\mathcal{R}$, $\mathcal{L}$, $\mathcal{H}$ and $\mathcal{J}$ called Green's relations:*

$$a\mathcal{R}b \Leftrightarrow aS^1 = bS^1$$
$$a\mathcal{L}b \Leftrightarrow S^1a = S^1b$$
$$a\mathcal{J}b \Leftrightarrow S^1aS^1 = S^1bS^1$$
$$a\mathcal{H}b \Leftrightarrow a\mathcal{R}b \text{ and } a\mathcal{L}b$$

Note that the relations $\mathcal{R}$ and $\mathcal{L}$ commute, i.e., $\mathcal{RL} = \mathcal{LR}$ and $\mathcal{D} = \mathcal{RL}$. In a finite semigroup $\mathcal{D} = \mathcal{J}$. A semigroup S is $\mathcal{K}$-trivial iff $e\mathcal{K}f$ implies $e = f$ for $\mathcal{K} \in \{\mathcal{D}, \mathcal{R}, \mathcal{L}, \mathcal{J}, \mathcal{H}\}$. A semigroup S is aperiodic if for all $x \in S$ there exists n such that $x^n = x^{n+1}$. Note that $S = Syn(B_\theta) \setminus \{1\}$ is aperiodic since all elements of S are idempotents.

We use the following propositions from [17] to show that $S = Syn(B_\theta) \setminus \{1\}$ is $\mathcal{H}$-trivial and the $\mathcal{D}$ class of S is equal to S.

Proposition 7 *[17] Let S be a semigroup and let g and f be idempotents of S. Then $g\mathcal{D}f$ if and only if g and f are conjugates, i.e., there exists $u, v \in S$ such that $g = uv$ and $f = vu$.*

Proposition 8 *[17] Let S be a finite semigroup. The following conditions are equivalent.*
1. S is aperiodic (for every $x \in S$ there exists n such that $x^n = x^{n+1}$).
2. There exists $m > 0$ such that for every $x \in S$, $x^m = x^{m+1}$.
3. S is $\mathcal{H}$-trivial.

Proposition 9 *The $\mathcal{D}$ class and $\mathcal{J}$ class of S is equal to S.*

Proof. Follows from the fact that S is simple and finite.

Proposition 10 *$S = Syn(B_\theta) \setminus \{1\}$ is $\mathcal{H}$-trivial.*

Proof. Since S is aperiodic, by Proposition 8, S is $\mathcal{H}$-trivial. □

Proposition 11 *Let $\Sigma = \{a_1, a_2, ..., a_n\}$ and let $[ab] \in S = Syn(B_\theta) \setminus \{1\}$ for some $a, b \in \Sigma$. Then the $\mathcal{R}$ class of $[ab]$ is $\{[aa_i] : a_i \in \Sigma\}$. and $\mathcal{L}$ class of $[ab]$ is $\{[a_ib] : a_i \in \Sigma\}$.*

Proof. Let $e\mathcal{R}f$ where $e = [ab]$ for some $a, b \in \Sigma$. $e = [ab]$ is the set of all words that begin with a and end with b. Then for all $f \in [ab]S^1$, f is the set of all words that begin with a. Thus $\mathcal{R}$ class of $[ab]$ is $\{[aa_i] : a_i \in \Sigma\}$. Similarly we can show that the $\mathcal{L}$ class of $[ab]$ is $\{[a_ib] : a_i \in \Sigma\}$. □

Corollary 7 *For all $e, f \in S$, $\mathcal{R}_e \cap \mathcal{L}_f = \{ef\}$.*

Proof. For some $e, f \in S$, $e = [ae_1]$ and $f = [f_1b]$ for some $a, b \in \Sigma$ and $e_1, f_1 \in \Sigma^*$. Note that $ef = [ae_1].[f_1b] = [ae_1f_1b] = [ab]$. Then from Proposition 11, $\mathcal{R}_{[ae_1]} = \{[aa_i] : a_i \in \Sigma\}$ and $\mathcal{L}_{[f_1b]} = \{[a_ib] : a_i \in \Sigma\}$. Thus $\mathcal{R}_e \cap \mathcal{L}_f = \{[ab]\} = \{ef\}$.

Example 1. Let $\Delta = \{A, C, G, T\}$ and let θ be an antimorphic involution that maps $A \mapsto T$ and $C \mapsto G$. Then $B_\theta = \{aub : a, b \in \Delta, \theta(a) = b, u \in \Delta^*\}$ is the set of all θ-bordered words over Δ^*. Then $Syn(B_\theta) = \{[1], [A], [C], [G], [T], [AC], [AG], [AT], [CA], [CG], [CT], [GA], [GC], [GT], [TA], [TG], [TC]\}$. Note that for all $a, b \in \Delta$, $[ab]$ is the set of all words that begin with a and end with b and $[a]$ represents the class that contains all words that begin and end with a. We now compute both the $\mathcal{R}$ and $\mathcal{L}$ class for elements of $Syn(B_\theta)$.

- $\mathcal{L}_{[A]} = \{\ [A], [CA], [GA], [TA]\}$
- $\mathcal{L}_{[C]} = \{\ [C], [AC], [GC], [TC]\}$
- $\mathcal{L}_{[G]} = \{\ [G], [AG], [CG], [TG]\}$
- $\mathcal{L}_{[T]} = \{\ [T], [CT], [GT], [AT]\}$
- $\mathcal{R}_{[A]} = \{\ [A], [AC], [AG], [AT]\}$
- $\mathcal{R}_{[C]} = \{\ [C], [CA], [CG], [CT]\}$
- $\mathcal{R}_{[G]} = \{\ [G], [GA], [GC], [GT]\}$
- $\mathcal{R}_{[T]} = \{\ [T], [TA], [TG], [TC]\}$

Also note that since $\mathcal{H} = \mathcal{R} \cap \mathcal{L}$ for all $e \in Syn(B_\theta)$, $\mathcal{H}_e = \{e\}$

5 Conclusion

The DNA secondary structure called "hairpin" has been a topic of constant interest in experimental as well as theoretical biomolecular computing, as it is usually undesirable in DNA-based computing experiments. This paper investigates a mathematical formalization of a particular case of hairpins, the Watson-Crick bordered words, whereby the "sticky borders" that cause a DNA single strand to form a hairpin are situated at the extremities of the strand. Cases where these "sticky borders" are situated in the interior of the strand have been addressed, e.g., in [9], [10]. The main results of this paper are algebraic properties of Watson-Crick bordered and unbordered words, and a complete characterization of the syntactic monoid of the language consisting of all Watson-Crick bordered words over a given alphabet.

Directions for future work are two-fold. On one hand we intend to investigate other generalizations of classical notions in combinatorics of words motivated by DNA computing, such as Watson-Crick conjugate words and Watson-Crick commutative words. On the other hand, we intend to formalize other DNA secondary structures such as DNA pseudo-knots and study their properties.

Acknowledgment Research supported by NSERC and Canada Research Chair grants for Lila Kari.

References

1. J.Berstel and D.Perrin, *Theory of Codes,* Academic Press, Inc. Orlando Florida, (1985).
2. M.Domaratzki. *Hairpin structures defined by DNA trajectories*, Proc. of DNA Computing 12, C.Mao, T.Yokomori, Editors, LNCS 4287 (2006), 182-194.
3. M.Garzon, V.Phan, S.Roy and A.Neel. *In search of optimal codes for DNA computing*, Proc. of DNA Computing 12, C.Mao, T.Yokomori, Editors, LNCS 4287 (2006), 143-156.
4. J.M.Howie, *Fundamentals of Semigroup Theory*, Oxford Science Publications, (1995).
5. N.Jonoska, K.Mahalingam and J.Chen, *Involution codes: with application to DNA coded languages*, Natural Computing, Vol 4-2 (2005), 141-162.
6. H.Jürgensen, *Syntactic monoid of codes*, Acta Cybernetica 14 (1999), 117-133.
7. H.Jürgensen and S.Konstantinidis, *Codes*, Handbook of Formal Languages, Vol 1, Chapter 8, G.Rozenberg, A.Salomaa, Editors, (1997), 511-608.
8. L.Kari, S.Konstantinidis, E.Losseva and G.Wozniak, *Sticky-free and overhang-free DNA languages*, Acta Informatica 40 (2003), 119-157.
9. L.Kari, S.Konstantinidis, E.Losseva, P.Sosik and G.Thierrin, *Hairpin structures in DNA words*, Proceedings of DNA Computing 11, A.Carbone, N.Pierce, Editors, LNCS 3892 (2005), 158-170.
10. L.Kari, K.Mahalingam and G.Thierrin, *The syntactic monoid of hairpin-free languages*, Accepted, Acta Informatica (2007).
    Available online: http://www.springerlink.com/content/2r264425831k6283/
11. L.Kari and K.Mahalingam, *Involution bordered words*, Accepted, IJFCS, (2007). Available online: http://www.csd.uwo.ca/~ lila/invbor.pdf

12. G.Lallement, *Semigroups and Combinatorial Dynamics*, Wiley/Interscience, New York (1995).
13. A.De Luca and A.Restivo, *A characterization of strictly locally testable languages and its application to subsemigroups of a free semigroup*, Information and Control 44 (1980), 300-319.
14. M.Petrich and C.M.Reis, *The syntactic monoid of the semigroup generated by a comma-free code*, Proceedings of the Royal Society of Edinburgh, 125A (1995), 165-179.
15. M.Petrich, C.M. Reis and G.Thierrin, *The syntactic monoid of the semigroup generated by a maximal prefix code*, Proceedings of the American Mathematical Society, 124-3 March (1996), 655-663.
16. M.Petrich and G.Thierrin, *The syntactic monoid of an infix code*, Proceedings of the American Mathematical Society 109-4 (1990), 865-873.
17. J.E.Pin, *Varieties of Formal Languages*, Plenum Press (1986).
18. H.J.Shyr, *Free Monoids and Languages*, Hon Min Book Company (2001).
19. G.Thierrin, *The syntactic monoid of a hypercode*, Semigroup Forum 6 (1973), 227-231.
20. S.S.Yu, *d-Minimal Languages*, Discrete Applied Mathematics 89 (1998), 243-262.