Acta Informatica 40, 119–157 (2003)
Digital Object Identifier (DOI) 10.1007/s00236-003-0118-7

Sticky-free and overhang-free DNA languages

Lila Kari[1], **Stavros Konstantinidis**[2], **Elena Losseva**[1], **Geoff Wozniak**[1]

[1] Deptartment of Computer Science, University of Western Ontario, London, Ontario, N6A 5B7 CANADA (e-mail: {lila;elena;wozniak}@csd.uwo.ca)
[2] Deptartment of Mathematics and Computing Science, Saint Mary's University, Halifax, Nova Scotia, B3H 3C3 CANADA (e-mail: s.konstantinidis@stmarys.ca)

Received: 6 February 2003
Published online: 2 September 2003 –

Abstract. An essential step of any DNA computation is encoding the input data on single or double DNA strands. Due to the biochemical properties of DNA, complementary single strands can bind to one another forming double-stranded DNA. Consequently, data-encoding DNA strands can sometimes interact in undesirable ways when used in computations. It is crucial thus to analyze properties that guard against such phenomena and study sets of sequences that ensure that no unwanted bindings occur during any computation. This paper formalizes and investigates properties of DNA languages that guarantee their robusteness during computations. After defining and investigating several types of DNA languages possessing good encoding properties, such as sticky-free and overhang-free languages, we give algorithms for deciding whether regular DNA languages are invariant under bio-operations. We also give a method for constructing DNA languages that, in addition to being invariant and sticky-free, possess error-detecting properties. Finally, we present the results of running tests that check whether several known gene languages (the set of genes of a given organism) as well as the input DNA languages used in Adleman's DNA computing experiment, have the defined properties.

Key words: Theoretical DNA computing, DNA encodings, codes, formal languages.

Research partially supported by Grants R2824A01 and R220259 of the Natural Sciences and Engineering Research Council of Canada.

1 Introduction

Deoxyribonucleic acid (DNA, Fig.1) is the molecule found in all cellular organisms that holds their genetic information. It is composed primarily of *nucleotides* (Figure 1 (b)) with a sugar-phosphate *backbone* (Figure 1 (a)). The nucleotides attach to the backbone to form a structure that resembles a clothesline if one were to use the backbone as the line and the nucleotides as the clothes. This is known as *single-stranded DNA*. A single strand of DNA has an orientation and its two ends are physically different. By convention, one end of the backbone is called the $5'$ end and the other one is called the $3'$ end.

Single-stranded DNA molecules can bind to each other to form *double-stranded DNA* molecules. This happens due to the fact that the nucleotides are pairwise Watson-Crick complementary: A is complementary to T and C to G. When two complementary single DNA strands with opposite orientation meet under favourable conditions, they bind to each other to form a double-stranded DNA molecule in a process called base-pairing, hybridization or annealing. The reverse process, of a double-stranded DNA molecule breaking apart into its single-stranded components is called melting or denaturation.

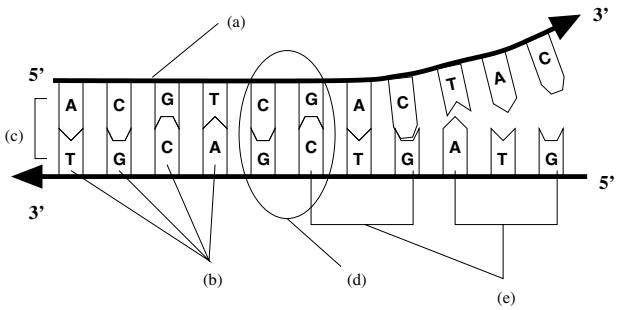

Fig. 1. A representation of a segment of a double-stranded DNA molecule. It is composed of a sugar-phosphate backbone (a) with a $5' \rightarrow 3'$ orientation, designated by the arrows, and a collection of nucleotides (b). The nucleotides form bindings (c) based on the Watson-Crick complementarity, such as those in (d). The two single-strands of DNA bind in an anti-parallel fashion, that is, when lined up, one strand is oriented $5' \rightarrow 3'$ and the other is oriented $3' \rightarrow 5'$. Non-overlapping sequences of three nucleotides (e) define codons, which eventually define the amino acid sequences of proteins

DNA computing is based on the fact that information (numbers, letters, special characters) can be encoded over the four-letter alphabet $\Delta = \{A, C, G, T\}$ and therefore represented physically by DNA strands. Moreover, molecular biology techniques can be used to manipulate those strands

and thus perform arithmetic and logic operations. The techniques that have so far been used for computations, called in the sequel *bio-operations* [17], include synthesis of desired strands, hybridization, denaturation, separation of strands by length, extraction from a heterogeneous solution of those strands that contain a given pattern as a subsequence, cut and paste DNA strands at desired locations, insert and delete DNA strands into other strands, make copies of DNA strands, detect and read out the sequence of letters composing a DNA strand, etc.

After the initial human intervention, consisting for example of mixing the appropriate components of a solution, most bio-operations consist either of DNA strands self-assembling, or of active molecules (like enzymes) acting upon DNA strands. A fundamental difference between an electronic computer and a DNA based-computer is that in the former data interaction is fully controlled by the programmer: one bit from a memory location will not affect another bit at another location, unless explicitely instructed to do so. In contrast, in a test-tube DNA-computer, data-encoding DNA strands can affect each other in undesired ways. Take for example the bio-operation hybridization based on Watson-Crick complementarity. Adleman's DNA algorithm, [1], for finding a Directed Hamiltonian Path in a given graph consisted of encoding the nodes and edges on single DNA strands in such a way that legal paths through the graph were formed by self-assembly: $5' \to 3'$ nodes were brought together by $3' \to 5'$ edges encoded especially to bind both the incoming and outgoing nodes. As seen in Adleman's experiment, hybridization is fundamental to DNA computing. However, if the input data is not carefully encoded, some data-encoding DNA single strands can bind to others rendering them useless for subsequent computation. This points out to yet another difference between DNA computing and electronic computing. In electronic computing an operand is not "consumed" by an operation, i.e. performing the addition 1 + 2 = 3 will not decrease the number of "1"s available for other additions. However, in DNA computing, a bio-operation usually consumes both operands. This means that if one of the operands is involved in an illegal binding, it may be unavailable for the desired computation and thus affect the correctness of the result.

In most proposed DNA-based algorithms, the initial DNA solution encoding the input to the problem will contain some DNA strands which represent single *codewords*, and some which represent strings of catenated codewords. Several attempts have been made to address the issue of "good encodings" by trying to find sets of codewords which are unlikely to form undesired bonds with each other by hybridization [5], [9], [10]. For example genetic and evolutionary algorithms have been developed which select for sets of DNA sequences that are less likely to form undesirable bonds [4], [6]. [7] has developed a program to create DNA sequences to meet logical

and physical parameters such as uniqueness, melting temperatures and G/C ratio as required by the user. [8] has addressed the issue of finding an optimal word design for DNA computing on surfaces. [12] has designed a software for constraint-based nucleotide selection. [11] has investigated encodings for DNA computing in virtual test tubes. [22] used combinatorial methods to calculate bounds on the size of a set of uniform code words (as a function of codeword length) which are less likely to mis-hybridize.

This paper continues the approach in [18], [14] by formalizing and investigating properties of languages that guarantee that no unwanted partial bindings will occur between the words of the language. The paper is organized as follows. Section 2 contains basic definitions, notation and examples of the notions used or defined in this paper, such as sticky-free DNA languages or overhang-free DNA languages. Section 3 investigates properties of such languages, for example, what are necessary and sufficient conditions for the catenation of two languages to have one of the desired properties. Section 4 gives algorithms for deciding whether a given regular language is invariant under bio-operations. By applying these results to the computation language of a DNA-based system (the set of all possible words that can be obtained during any bio-computation) we can decide whether "good" encoding properties of the initial input language are preserved during a bio-computation. Section 5 gives a method for constructing languages that, in addition to being invariant, nonoverlapping and sticky-free, possess error-detecting capabilities. Finally, Section 6 presents the results of running tests checking whether several known gene languages (the set of genes of a given organism), as well as the input DNA language used in Adleman's first DNA computing experiment, have the properties we have defined.

2 Definitions and Examples

For a finite set S, we denote by $|S|$ the cardinality of S, that is, the number of elements in S. The set of non-negative integers is denoted by $\mathbf{N}$. Let X^* be the free monoid generated by the finite alphabet X under the catenation operation, where 1 denotes the empty word. X^+ equals $X^* \setminus \{1\}$. A word w over X is a string $w = a_1 a_2 \dots a_n$ where $a_i \in X$. The length of the word w is denoted by $|w|$ and is the number of its letters, including repetitions, $|w| = n$. The length of the empty word is zero. A language L is a subset of X^*. The catenation of two languages $L_1, L_2 \subseteq X^*$ is defined as $L_1 L_2 = \{uv |\ u \in L_1, v \in L_2\}$. A mapping $\alpha : X^* \to X^*$ is called a *morphism (anti-morphism)* of X^* if $\alpha(uv) = \alpha(u)\alpha(v)$ (respectively $\alpha(uv) = \alpha(v)\alpha(u)$) for all $u, v \in X^*$. A bijective morphism (anti-morphism) is called an *isomorphism (anti-isomorphism)* of X^*. Note

that both a morphism and an anti-morphism of X^* are completely defined if we define their values on the letters of X.

An *involution* $\theta : S \to S$ of S is a mapping such that θ^2 is equal to the identity mapping, i.e., $\theta(\theta(x)) = x$ for all $x \in S$. It follows then that an involution θ is bijective and $\theta = \theta^{-1}$. The identity mapping is a trivial example of an involution. In general, if $f : X \to X$ is an involution, then X can be partitioned into $X = A \cup A' \cup B$ where $|A| = |A'|$ and, for every $a \in A$ we have $f(a) = a'$, $f(a') = a$, $a' \in A'$, while $f(b) = b$ for all $b \in B$. If $A = A' = \emptyset$ then f is the identity on X, while if $B = \emptyset$ f is a sort of complement function on X which maps every element of A into an element of A' and vice versa.

An involution of X can be extended to either a morphism or an anti-morphism of X^*. For example, if the identity of X is extended to a morphism of X^*, we obtain the identity involution of X^*. However, if we extend the identity of X to an anti-morphism of X^* we obtain instead the mirror-image involution of X^* that maps each word u into v where

$$u = a_1 a_2 \dots a_k, \quad v = a_k \dots a_2 a_1, a_i \in X, 1 \leq i \leq k.$$

If Δ^* is the free monoid generated by the DNA-alphabet Δ then the mapping $\tau : \Delta \to \Delta$ defined by $\tau(A) = T, \tau(T) = A, \tau(C) = G, \tau(G) = C$ can be extended in the usual way to an anti-morphism of Δ^* that is also an involution of Δ^*. This involution formalizes the notion of Watson-Crick complement of a DNA sequence and will therefore be called the *DNA involution*, [18]. By convention, a word $w = a_1 a_2 \dots a_n$ in Δ^* will signify the DNA single strand $5' - a_1 a_2 \dots a_n - 3'$.

We conclude the list of definitions needed with some coding theory notions. A code K is a subset of X^+ satisfying the property that, for every word w in K^+, there is a unique sequence $(v_1, v_2, \dots, v_n)$ of words in K such that $w = v_1 v_2 \cdots v_n$. An infix code, K, has the property that no word of K is properly contained in another word of K, that is, $K \cap (X^+ K X^* \cup X^* K X^+) = \emptyset$. A comma-free code K is a language with the property $K^2 \cap X^+ K X^+ = \emptyset$. Every comma-free code is an infix code.

Let us return now to the DNA computation set-up. In a DNA algorithm the input data consists of a set of "codewords" represented by DNA strands. In our terminology, this is a language over Δ^+. We are interested in defining languages in such a way that no two codewords can bind to each other. [18], [14] have defined and analyzed several types of unwanted hybridizations. For example, a language $L \subseteq \Delta^+$ where no codeword is the exact Watson-Crick complement of another codeword is called τ-nonoverlapping. A language where a codeword has the property that its Watson-Crick complement never is a subword of (and thus never binds to a segment of) another one is called DNA compliant. A language where the Watson-Crick complement of

a codeword never is a subword of the catenation of two other codewords is called τ-free. In [18], [14] properties of languages with "good" coding features such as τ-nonoverlapping, DNA compliant and τ-free, were defined and investigated.

Note that until now we eliminated the cases where one word was completely complementary to a segment of another, or to a segment of the catenation of two others. However, the biological reality points to other possible undesired hybridizations: two strands can stick to each other even when both of them have only segments of themselves that are fully complementary. For example if two codewords have enough of an overlap, they will bind producing a strand that is partially double-stranded and with single-stranded overhangs ("sticky ends") on the $5'$ end or the $3'$ end (see Fig.2, third row, 1st, 3rd and 4th box). This paper continues the study in [18], [14] by addressing types of unwanted hybridizations that involve such partial bindings of codewords.

A summary of all the desirable properties of languages is given below, where θ is an arbitrary morphic or anti-morphic involution of X^* and $L \subseteq X^+$. In the particular case where θ is the DNA involution and $X = \Delta$, they depict the good encoding properties that languages consisting of DNA strands should have if they are to be used for computations. Definitions (A)-(E) of θ-nonoverlapping and θ-compliant languages were given in [18]; definition (F) of a θ-free language was given in [14]. Definitions (G)-(J) of θ-sticky-free and θ-overhang-free languages are new notions introduced here.

(A) θ**-nonoverlapping**: $L \cap \theta(L) = \emptyset$.
(B) θ**-compliant**: $\forall w \in L,\ x, y \in X^*,\ w, x\theta(w)y \in L \Rightarrow xy = 1$.
(C) θ-p**-compliant**: $\forall w \in L,\ y \in X^*,\ w, \theta(w)y \in L \Rightarrow y = 1$.
(D) θ-s**-compliant**: $\forall w \in L,\ y \in X^*,\ w, y\theta(w) \in L \Rightarrow y = 1$.
(E) **strictly** θ**-compliant**: $\forall w \in L,\ x, y \in X^*,\ w, x\theta(w)y \in L \Rightarrow xy = 1$ and $w \neq \theta(w)$.
(F) θ**-free**: $L^2 \cap X^+\theta(L)X^+ = \emptyset$.
(G) θ**-sticky-free**: $\forall w \in X^+,\ x, y \in X^*,\ wx, y\theta(w) \in L \Rightarrow xy = 1$.
(H) θ-$3'$**-overhang-free**: $\forall w \in X^+,\ x, y \in X^*,\ wx, \theta(w)y \in L \Rightarrow xy = 1$.
(I) θ-$5'$**-overhang-free**: $\forall w \in X^+,\ x, y \in X^*,\ xw, y\theta(w) \in L \Rightarrow xy = 1$.
(J) θ**-overhang-free**: both θ-$3'$-overhang-free and θ-$5'$-overhang-free.

For convenience, we agree to say that a language L containing the empty word has one of the above properties if $L \setminus \{1\}$ has that property.

Some of the situations these definitions are meant to depict are graphically presented in Figure 2: Each box in the figure represents the situation forbidden in a class of languages having a certain property. The arrows

signify proper inclusion. For example, if a DNA language is θ-3′-overhang-free then it is also θ-p-compliant. The names of the corresponding language properties are given in brackets. For example, a θ-p-compliant language is a language where no two words can bind to form a structure like that in the bottom left box of Figure 2. θ-compliance, θ-p-compliance and θ-s-compliance (p stands for prefix and s stands for suffix) have been defined and studied in [18], [14], while θ-sticky-freeness and θ-overhang-freeness, θ-3′-overhang-freeness, θ-5′-overhang-freeness are investigated in this paper. The properties depicted by the top three boxes are the topic of future study.

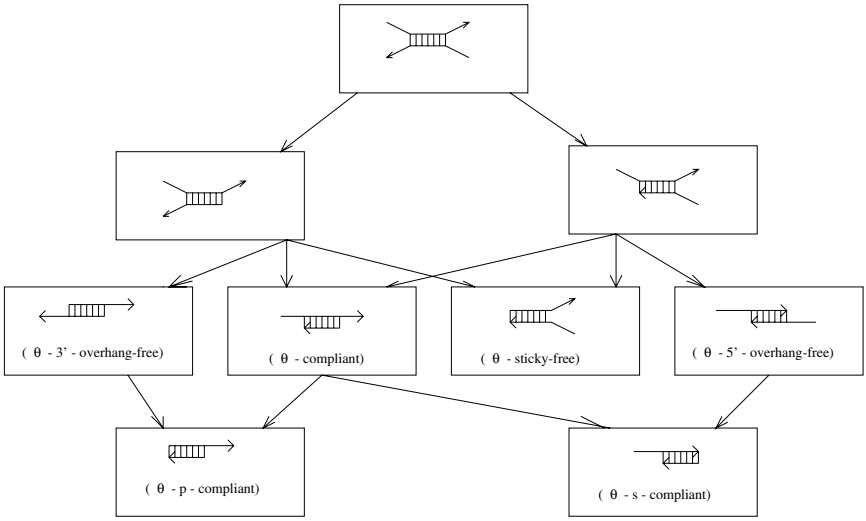

Fig. 2. Language hierarchy

To further clarify the notions, we give in the following examples of languages that have or do not have the defined properties.

Example 1. If we consider $X = \{a, b\}$ and f an anti-morphic involution of X^* defined by $f(a) = b$, $f(b) = a$, then the language $L = \{a^n b^n | \ n \geq 1\}$ is not f-compliant. Indeed, for any $k \geq 1$ $f(a^k b^k) = f(b^k)f(a^k) = a^k b^k$ which is a subword of infinitely many words in L. The language L is not f-nonoverlapping as $L \cap f(L) = L \neq \emptyset$.

However, L is f-p-compliant as $v, f(v)x \in L$ imply $v = a^i b^i, a^i b^i x \in L$ which implies $x = 1$. A similar argument shows that L is f-s-compliant.

If instead of the involution f we consider the function $g(a) = b, g(b) = a$ extended to a morphic involution, then L is g-compliant. Indeed, any word $w = a^n b^n \in L$ cannot have as a subword a word of the type $g(u) = b^k a^k$, where $u = a^k b^k \in L$. L is also g-nonoverlapping as $L \cap g(L) = \emptyset$.

Observe that, for an involution θ, if a language is θ-compliant then it is both θ-p-compliant and θ-s-compliant but the reverse does not hold as shown by the previous example.

Note that the notions of θ-compliant, θ-p-compliant and θ-s-compliant language become, in the particular case of θ being the identity function on X, extended to a morphic involution, the well-known notions of infix code, prefix code and respectively suffix code.

Example 2. If $X = \{a, b\}$ and g is the morphic involution $g(a) = b, g(b) = a$, the language $L = \{a^n b^n | \ n \geq 1\}$ is not g-free. Indeed, we can find $u = a^i b^i$, $v = a^j b^j$ and $w = a^k b^k$ with $k < \min\{i, j\}$ such that

$$uv = a^i b^i a^j b^j = a^i b^{i-k} b^k a^k a^{j-k} b^j = xg(w)y,$$

where $x = a^i b^{i-k}$, $y = a^{j-k} b^j$ and therefore $xy \neq 1$.

On the other hand, for $Y = \{a, b, c\}$ and h extended to a morphic involution from $h(a) = c$, $h(c) = a$, $h(b) = b$, we have that the language $L' = \{a^n b^n c^n | \ n \geq 1\}$ is h-free. Indeed, words $uv \in L'^2$ are of the form $uv = a^i b^i c^i a^j b^j c^j$, $i, j > 0$, while, for any $w = a^k b^k c^k \in L'$, $h(w) = c^k b^k a^k$ and therefore $h(w)$ cannot be a subword of any $uv \in L'^2$.

Note that in the particular case when θ is the identity on an alphabet X extended to a morphic involution of X^*, the notion of a θ-free language becomes the well-known notion of a comma-free code.

Example 3. If $X = \{a, b\}$ and the function $e(a) = a$, $e(b) = b$ is extended to an anti-morphic involution then the language $L = \{a^n b^n | \ n > 0\}$ is e-sticky-free. This follows as any word $wx \in L$ is of the form $a^n b^n$, $n > 0$ and therefore $w = a^i b^j$, $i > 0$, $j \geq 0$. However then we have $e(w) = b^j a^i$ which cannot be a suffix of any word in L.

On the other hand, if Y, L' and h are like in Example 2, then L' is not h-sticky-free. We can have, for example, $u = v = a^i b^i c^i$, $i > 0$ in L' and therefore $u = wx$, $v = yh(w)$ for $x = b^i c^i$, $y = a^i b^i$, $w = a^i$.

Example 4. If $Y = \{a, b, c\}$, $L' = \{a^n b^n c^n | \ n > 0\}$, and t is defined as $t(a) = a$, $t(b) = c$, $t(c) = b$ and is extended to a morphic involution, then L' is not t-3'-overhang-free. Indeed, we can find $wx = a^i b^i c^i$, $i > 0$, $w = a^j$, $x = a^{i-j} b^i c^i$ such that $t(w)y = a^j b^j c^j \in L'$. The language L' is t-5'-overhang-free as $xw \in L'$ means $xw = a^i b^i c^i$ and therefore $t(w) = ub^k, k > 0$. However, no word in L' ends in b and therefore $t(w)$ cannot be the suffix of any word in L'. As it is t-5'-overhang-free but not t-3'-overhang-free, it follows that L' is not t-overhang-free.

An example of a language which is not g-5'-overhang-free where g is defined as in Example 2 is $L'' = \{w \in \{a, b\}^* | \ |w|_a = |w|_b\}$. We can find indeed words $x = y = b^3 a^3$, $w = ba$ such that both xw and $yg(w)$ are in L''.

3 Sticky-free and overhang-free languages

We have defined several properties that are desirable for DNA languages to have. The practical implication is that one can write algorithms (see Section 6) that can check whether or not a given DNA language has these properties. Ideally a minimal number of such checks would be optimal, therefore we are interested in finding interconnections between these properties which would reduce the number of checks needed. For example, the result that a language which is $3'$-overhang-free is p-compliant has the practical implication that we only need to check the property of $3'$-overhang-freeness. The first part of this section investigates such relations between some of the properties, complementing thus the results obtained in [14]. The second part of the section addresses the problem of catenation of DNA languages. This is a practical problem that might arise when combining two computations, which means taking the union of, but also possibly catenating, their input DNA languages. The question we address is under which conditions the catenation of two sticky-free or overhang-free languages has the same property.

The following proposition makes a connection between the notions of θ-sticky-free and θ-compliant languages. (If $L \subseteq X^+$ is a language then L_p denotes the set of all its proper and nonempty prefixes and L_s the set of all its proper and nonempty suffixes.)

Proposition 1. *For every language $L \subseteq X^+$ and for every given morphic or anti-morphic involution $\theta : X^+ \to X^+$, the following are equivalent:*

(1) L is θ-sticky-free;
(2) $\theta(L)$ is θ-sticky-free;
(3) $L_p \cap \theta(L_s) = \emptyset$ and L is both θ-p-compliant and θ-s-compliant.

Proof. (1)⇒(2)

Suppose $\theta(L)$ is not θ-sticky-free, i.e., there exist $x, y \in X^*, w \in X^+$ such that $wx \in \theta(L)$, $y\theta(w) \in \theta(L)$ and $xy \neq 1$. If θ is morphic, then $\theta(w)\theta(x) \in L$ and $\theta(y)w \in L$. Since L is sticky-free, we have $1 = \theta(x)\theta(y) = \theta(xy)$ and therefore $xy = 1$ – a contradiction. If θ is anti-morphic, then $\theta(x)\theta(w) \in L$ and $w\theta(y) \in L$. Again, since L is sticky-free, $1 = \theta(x)\theta(y) = \theta(yx)$ and therefore $yx = 1$ – a contradiction. Hence $\theta(L)$ must be θ-sticky-free.

(2)⇒(3)

Suppose that $L_p \cap \theta(L_s) \neq \emptyset$. Let u be an element of $L_p \cap \theta(L_s)$. $u \in L_p$ implies that $ux \in L$ for some $x \in X^+$. The fact that $u \in \theta(L_s)$ implies that $\theta(u) \in L_s$ which, in turn, implies that $y\theta(u) \in L$ for some $y \in X^+$. If θ is morphic, then $\theta(u)\theta(x) \in \theta(L)$ and $\theta(y)u \in \theta(L)$. Since $\theta(L)$ is θ-sticky-free we have $1 = \theta(x)\theta(y) = \theta(xy)$ and so $xy = 1$ – a contradiction. If θ is anti-morphic, we have $\theta(x)\theta(u) \in \theta(L)$ and also

$u\theta(y) \in \theta(L)$. Hence $1 = \theta(y)\theta(x) = \theta(xy)$ and $xy = 1$ – a contradiction. Hence $L_p \cap \theta(L_s) = \emptyset$. To show that L is θ-p-compliant, suppose that $ux \in L$ and $\theta(u) \in L$ with $u \in X^+$. Then if θ is morphic, $\theta(u)\theta(x) \in \theta(L)$ and $u \in \theta(L)$. Because $\theta(L)$ is sticky-free, $\theta(x) = 1$, and hence $x = 1$. If θ is anti-morphic, then $\theta(x)\theta(u) \in \theta(L)$ and $u \in \theta(L)$ imply $x = 1$ again. Hence L is θ-p-compliant and θ-s-compliance can be similarly shown.

(3)⇒(1) Suppose that L is not θ-sticky-free. Then there exist $wx \in L, w \in X^+, y\theta(w) \in L, x, y \in X^*$ with $xy \neq 1$. If $x \neq 1, y \neq 1$, then $w \in L_p, \theta(w) \in L_s$ which implies $w \in \theta(L_s)$, and therefore $w \in L_p \cap \theta(L_s)$ – a contradiction. If $x \neq 1$ and $y = 1$, then L is not θ-s-compliant – a contradiction. Conversely, $x = 1, y \neq 1$ imply L is not θ-p-compliant – a contradiction. □

The following proposition shows a connection between the θ-sticky-free, θ-compliant, θ-overhang-free and θ-free languages.

Proposition 2. *Let X be an alphabet, θ an involution, and L, $\emptyset \neq L \subseteq X^+$ be a language. In case θ is morphic, then if L is θ-compliant and θ-sticky-free then L is θ-free. In case θ is anti-morphic, then if L is θ-compliant and either θ-3′-overhang-free or θ-5'-overhang-free then L is θ-free.*

Proof. Suppose L is not θ-free, i.e., $L^2 \cap X^+\theta(L)X^+ \neq \emptyset$. Then there exist $u, v, w \in L$ and $x, y \in X^+$ such that $uv = x\theta(w)y$.

Consider the case when $\theta(w)$ is a subword of u. Let $u = u_1u_2u_3 \in L$ such that $\theta(w) = u_2$. Observe that $u_1u_3 \neq 1$ since $x \in X^+$. Moreover, since L is θ-compliant, $\theta(L)$ is also θ-compliant, [18].

If θ is morphic, then $\theta(u) = \theta(u_1)\theta(u_2)\theta(u_3) \in \theta(L)$ and $u_2 \in \theta(L)$ imply $\theta(u_1)\theta(u_3) = 1$ by the θ-compliance of $\theta(L)$. Hence $\theta(u_1u_3) = 1$ and so $u_1u_3 = 1$ – a contradiction.

If θ is anti-morphic, then $\theta(u) = \theta(u_3)\theta(u_2)\theta(u_1) \in \theta(L)$, $u_2 \in \theta(L)$ imply $\theta(u_3)\theta(u_1) = 1$, by the θ-compliance of $\theta(L)$. Hence $\theta(u_1u_3) = 1$ and so $u_1u_3 = 1$ – a contradiction.

If $\theta(w)$ is a subword of v, we can reason similarly to the above case to get a contradiction.

Otherwise, we have $u = xu_2, v = v_1y$ and $\theta(w) = u_2v_1$ for some $u_2, v_1 \in X^+$.

If θ is morphic, then $w = \theta(u_2v_1) = \theta(u_2)\theta(v_1) \in L$, $v_1y \in L$ imply $\theta(u_2)y = 1$, since L is θ-sticky-free. But we have $y \in X^+$, i.e. $y \neq 1$ – a contradiction.

If θ is anti-morphic and L is θ-5'-overhang-free then $w = \theta(u_2v_1) = \theta(v_1)\theta(u_2) \in L$, $xu_2 \in L$ imply $\theta(v_1)x = 1$. If L is θ-3′-overhang-free then we use the fact that $v_1y \in L$ implies $\theta(u_2)y = 1$. Both cases contradict the fact that $x, y \in X^+$, i.e., $x \neq 1, y \neq 1$. □

In the remainder of this section we study the problem of when, given two languages L_1 and L_2 with property $\mathcal{P}$, their catenation also has property $\mathcal{P}$. [18] studied conditions under which the catenation of θ-compliant languages is also θ-compliant and [14] found conditions under which the catenation of θ-free languages is still θ-free. Here we are considering situations where the properties of θ-sticky-freeness and θ-overhang-freeness are preserved under catenation.

Note first that the catenation of two sticky-free languages is not always sticky-free.

Example 5. Take $Y = \{a, b, c\}$, $L_1 = \{a^n b^n |\ n > 0\}$, $L_2 = \{b^n c^n |\ n > 0\}$, and t the morphic involution on Y defined as $t(a) = c$, $t(c) = a$, $t(b) = b$. The language L_1 is t-sticky-free. Indeed, any word $wx \in L_1$ starts with $w = a^i b^j$, $i > 0$, $j \geq 0$. This implies $t(w) = c^i b^j$ which cannot be a suffix of any word in L_1. Also L_2 is t-sticky-free as $wx \in L_2$ implies $w = b^i c^j$, $i > 0$, $j \geq 0$, therefore $t(w) = b^i a^j$ which cannot be a suffix of any word in L_2.

However, $L_1 L_2 = \{a^n b^{n+p} c^p |\ n, p > 0\}$ is not t-sticky-free. Indeed, we can find words $w = a^i$, $x = b^{i+j} c^j$, $y = a^k b^{k+i}$ such that $wx = a^i b^{i+j} c^j \in L_1 L_2$ and also $yt(w) = a^k b^{k+i} c^i \in L_1 L_2$.

Example 6. Let $Y = \{a, b, c\}$ and let e be the identity function on Y extended to an anti-morphic involution. Let $L_1 = \{a^n b^n c^n |\ n > 0\}$ and $L_2 = \{c^n b^n a^n |\ n > 0\}$. The language L_1 is e-sticky-free as the images through e of nontrivial prefixes of words L_1 are not suffixes of words in L_1. The same argument shows that L_2 is e-sticky-free. However, $L_1 L_2 = \{a^n b^n c^{n+m} b^m a^m |\ n, m > 0\}$ is not e-sticky-free as we have images of prefixes of $L_1 L_2$ which are at the same time suffixes of $L_1 L_2$.

The following proposition gives a sufficient condition for the catenation of two languages to be θ-sticky-free.

Lemma 1. *Let θ be an anti-morphic involution and let L be a θ-sticky-free language. If two words $w_1 \in L$ and $w_2 \in \theta(L)$ have a common and nonempty prefix then $w_1 = w_2$.*

Proof. Suppose $w_1 = px_1$ and $w_2 = px_2$, for some words x_1, x_2, p with p nonempty. Then $\theta(x_2)\theta(p) \in L$, which implies that $x_1 = x_2 = 1$ using the fact that L is θ-sticky-free. Hence, $w_1 = w_2$. □

Proposition 3. *Let X be a finite alphabet, let $\theta : X^* \to X^*$ be an involution, and let L_1, L_2 be nonempty subsets of X^+ such that $L_1 \cup L_2$ is θ-sticky-free and $L_1 \cap \theta(L_2) = \emptyset$. Then the following statements hold true.*

1 *L_1L_2 is θ-sticky-free.*
2 *If L_1 is θ-nonoverlapping then $L_1 \cup L_1L_2$ is θ-nonoverlapping and θ-sticky-free.*
3 *If L_2 is θ-nonoverlapping then $L_2 \cup L_1L_2$ is θ-nonoverlapping and θ-sticky-free.*
4 *If L_1 and L_2 are θ-nonoverlapping then $L_1 \cup L_2 \cup L_1L_2$ is θ-nonoverlapping and θ-sticky-free.*

Proof. For the first part, assume that L_1L_2 is not θ-sticky-free. Then there are two words of the form wx and $y\theta(w)$ in L_1L_2 such that w is nonempty and not both x and y are empty. We show that this assumption leads to a contradiction. Consider words u_1, v_1 in L_1 and u_2, v_2 in L_2 such that $wx = u_1u_2$ and $y\theta(w) = v_1v_2$.

First consider the case where θ is anti-morphic. As $wx = u_1u_2$ and $w\theta(y) = \theta(v_2)\theta(v_1)$, the words u_1 and $\theta(v_2)$ have a common and nonempty prefix. By Lemma 1, this implies that $u_1 = \theta(v_2)$, which contradicts $L_1 \cap \theta(L_2) = \emptyset$.

Now consider the case where θ is morphic. We distinguish four subcases.

(a) $|w| \leq |u_1|$ and $|\theta(w)| \leq |v_2|$.
(b) $|w| \leq |u_1|$ and $|\theta(w)| > |v_2|$.
(c) $|w| > |u_1|$ and $|\theta(w)| \leq |v_2|$.
(d) $|w| > |u_1|$ and $|\theta(w)| > |v_2|$.

Subcase (a) implies that both x and y are nonempty, and $u_1 = wx_1$ and $v_2 = y_2\theta(w)$, for some words x_1 and y_2. This leads to a contradiction using the fact that $L_1 \cup L_2$ is θ-sticky-free and $L_1 \cap \theta(L_2) = \emptyset$. Subcase (b) implies that x is nonempty and $\theta(w) = s_1v_2$, $u_1 = wx_1$ and $v_1 = ys_1$, where s_1 is a nonempty suffix of v_1 and x_1 is a proper prefix of x. Then we have that $u_1 = \theta(s_1)\theta(v_2)x_1$ and $v_1 = y\theta(\theta(s_1))$, which contradicts the fact that $L_1 \cup L_2$ is θ-sticky-free. Subcase (c) also leads to a contradiction using the arguments of subcase (b).

Finally, subcase (d) implies that $w = u_1p_2$, $\theta(w) = s_1v_2$, $u_2 = p_2x$ and $v_1 = ys_1$, where p_2 is a nonempty prefix of u_2 and s_1 is a nonempty suffix of v_1. Then we have that $u_1p_2 = \theta(s_1)\theta(v_2)$. If $|\theta(v_2)| < |p_2|$ then $p_2 = z\theta(v_2)$ and $\theta(s_1) = u_1z$, for some proper and nonempty prefix z of p_2. Moreover, it follows that $v_1 = y\theta(u_1)\theta(z)$ and $u_2 = z\theta(v_2)x$, which contradicts the fact that $L_1 \cup L_2$ is θ-sticky-free. If $|\theta(v_2)| \geq |p_2|$ then $\theta(v_2) = zp_2$ and $u_1 = \theta(s_1)z$, for some proper suffix z of u_1. As $u_2 = p_2x$ and $v_1 = y\theta(\theta(s_1))$, we have that $x = z = 1$ and $y = z = 1$, which contradicts the assumption that x and y are nonempty.

For the second part it is sufficient to show that the following claim is false: there are words $x, y \in X^*$, $xy \in X^+$, and $w \in X^+$ such that $wx, y\theta(w) \in L_1 \cup L_1L_2$. We distinguish four cases:

(a) $wx \in L_1$ and $y\theta(w) \in L_1$.
(b) $wx \in L_1$ and $y\theta(w) \in L_1L_2$.
(c) $wx \in L_1L_2$ and $y\theta(w) \in L_1$.
(d) $wx \in L_1L_2$ and $y\theta(w) \in L_1L_2$.

Case (a) is not possible as L_1 is θ-nonoverlapping and θ-sticky-free. In case (b), there are words $w_1 \in L_1$ and $w_2 \in L_2$ such that $y\theta(w) = w_1w_2$. Then, as $wxw_2 \in L_1L_2$ and L_1L_2 is θ-sticky-free, we have $y = xw_2 = 1$ which is impossible. In case (c), we have $wx = w_1w_2$ for some words $w_1 \in L_1$ and $w_2 \in L_2$. If $|w| \leq |w_1|$ then there is $x_1 \in X^*$ such that $wx_1 = w_1 \in L_1$. This is impossible, however, as L_1 is θ-nonoverlapping and θ-sticky-free. Now if $|w| > |w_1|$ then $w = w_1s$, $w_2 = sx$, and $y\theta(w_1s) \in L_1$, for some $s \in X^+$. If θ is anti-morphic then $y\theta(s)\theta(w_1) \in L_1$ which is impossible as L_1 is θ-nonoverlapping and θ-sticky-free. If θ is morphic then $y\theta(w_1)\theta(s) \in L_1$ which is impossible as $sx \in L_2$ and $L_1 \cup L_2$ is θ-sticky-free. In case (d), we have $x = y = 1$ using the first part of the proposition. Then, $w = w_1w_2$ and $\theta(w) = u_1u_2$ for some words $w_1, u_1 \in L_1$ and $w_2, u_2 \in L_2$. If θ is anti-morphic then $w = \theta(u_2)\theta(u_1)$ which is impossible as L_1L_2 is θ-sticky-free. Now if θ is morphic then $w_1w_2 = \theta(u_1)\theta(u_2)$. But then, using a case distinction on the relation between $|w_1|$ and $|\theta(u_1)|$, one can verify that the assumption of L_1 being θ-nonoverlapping and θ-sticky-free is contradicted.

The third part of the proposition can be proved using similar arguments. Finally, the fourth part follows from the previous ones. □

The following corollary shows that, if we have at our disposal a θ-nonoverlapping and θ-sticky-free language K, then the language obtained by taking arbitrary catenations of words from K remains θ-nonoverlapping and θ-sticky-free. The result is relevant, as catenation of codewords is one of the most common ways of combining information encoded on DNA strands.

Corollary 1. *Let θ be an involution of X^* and let $K \subseteq X^+$ be a language. If K is θ-nonoverlapping and θ-sticky-free then K^+ also is θ-nonoverlapping and θ-sticky-free.*

Proof. Assume that K is θ-nonoverlapping and θ-sticky free. We use induction on n to show that $\bigcup_{i=1}^{n} K^i$ is θ-nonoverlapping and θ-sticky-free. The case of $n = 1$ is trivial. Now suppose the claim holds for $\bigcup_{i=1}^{n} K^i$. Then, $K \cup (\bigcup_{i=1}^{n} K^i)$ is θ-sticky-free and, as $K \cap \theta(\bigcup_{i=1}^{n} K^i) = \emptyset$, one has that $K \cup K(\bigcup_{i=1}^{n} K^i)$ is θ-sticky-free. Hence, $\bigcup_{i=1}^{n+1} K^i$ is θ-nonoverlapping and θ-sticky-free and the induction is complete. □

Note that Proposition 3, (1), gives a sufficient condition (in two parts) for the catenation of two languages L_1L_2 to be θ-sticky-free for a given involution θ. To see whether the conditions are also necessary, we analyze situations where the catenation of two languages is not θ-free and see whether this coincides with a violation of condition (1) in Proposition 3.

One such example is that of the languages in Example 5. In that case, the languages satisfy condition $L_1 \cap t(L_2) = \emptyset$ but fail to satisfy the condition $L_1 \cup L_2$ being t-sticky-free. Indeed, $L_1 \cup L_2 = \{a^n b^n, b^m c^m | \ n, m > 0\}$ is not t-sticky-free.

The languages in Example 6 fail to satisfy both conditions. Indeed, $L_1 \cap e(L_2) = L_1 \neq \emptyset$ and $L_1 \cup L_2 = \{a^n b^n c^n, c^m b^m a^m | \ n, m > 0\}$ which is not e-sticky-free.

Consider a third example, $L_1 = \{a\}$, $L_2 = \{b\}$ over $X = \{a, b\}$, and $f(a) = b, f(b) = a$ extended to an anti-morphic involution. These languages satisfy the condition $L_1 \cup L_2$ being f-sticky-free but $L_1 \cap f(L_2) = \{a\} \neq \emptyset$. The catenation $L_1 L_2 = \{ab\}$ is not f-sticky-free as we have $ab = af(a) \in L_1 L_2$.

The above examples all depict situations in which the languages involved are θ-sticky-free and where failure to satisfy one or both parts of condition (1) in Proposition 3 coincides with $L_1 L_2$ being not θ-sticky-free. This suggests that, in case both languages are θ-sticky-free, condition (1) in Proposition 3 for their catenation to be θ-sticky- free is also necessary. The following result holds.

Proposition 4. *Let θ be an involution of X^* and let $L_1, L_2 \subseteq X^+$ be two nonempty θ-sticky-free languages. Then $L_1 L_2$ and $L_2 L_1$ are θ-sticky-free iff $L_1 \cup L_2$ is θ-sticky-free and $L_1 \cap \theta(L_2) = \emptyset$.*

Proof. One of the implications follows from 1. of Proposition 3. For the converse implication, assume that θ is a morphic or anti-morphic involution of X^* and L_1, L_2 are two θ-sticky-free languages such that $L_1 \cap \theta(L_2) \neq \emptyset$. Let $w \in L_1 \cap \theta(L_2)$. Then there exists $u \in L_2$ such that $\theta(u) = w, w \in L_1$. Then $wu \in L_1 L_2$ and $wu = w\theta(w)$. We found a word that is prefix of a word in $L_1 L_2$ and its image under θ is a suffix of a word in $L_1 L_2$, which means $L_1 L_2$ is not θ-sticky-free.

Assume now that $L_1 \cup L_2$ is not θ-sticky-free. As L_1, L_2 are θ-sticky-free, the only possibilities are that there exist $wx \in L_1$ with $y\theta(w) \in L_2$ or $wx \in L_2$ with $y\theta(w) \in L_1$. Consequently, $wxy\theta(w)$ is in $L_1 L_2$ or $L_2 L_1$ contradicting the fact that $L_1 L_2$ and $L_2 L_1$ are θ-sticky-free. □

Proposition 4 does not hold if we replace "$L_1 L_2$ and $L_2 L_1$ are θ-sticky-free" by "$L_1 L_2$ is θ-sticky-free", as shown by the following example. Consider the DNA alphabet, the DNA involution τ and the languages $L_1 = \{ACTG, GGAA\}$, $L_2 = \{TTCA, CAGG\}$. Then we have that $L_1 L_2$ is τ-sticky-free, but $L_1 \cup L_2$ is not τ-sticky-free. Note that in this case $L_2 L_1$ fails to be τ-sticky-free.

The rest of this section will address the question of when the catenation of two languages is θ-overhang-free for some involution θ of X^*. Recall that DNA strands are "directed" molecules with two distinct ends called the $3'$

end and 5' end. Because of this polarity distinction, the overhang structure (see Fig.2) has two cases. In one case, it is the $3'$ ends of the two strands that hang over, in the other case it is the 5' ends. We still consider general results about morphic and anti-morphic involutions. However, because the $3'$-overhang and 5'-overhang structures are similar, there is some symmetry in the four propositions that follow.

In case the involution θ is morphic, a sufficient condition for L_1L_2 to be θ-$3'$-overhang-free is very weak. It namely suffices for L_1 to be θ-$3'$-overhang-free in order for L_1L_2 to have the same property, regardless of the properties of L_2.

Proposition 5. *Let L_1, L_2 be nonempty subsets of X^+ and let θ be a morphic involution. If L_1 is θ-nonoverlapping and θ-$3'$-overhang-free, then $L_1 \cup L_1L_2$ and, therefore, L_1L_2 are θ-$3'$-overhang-free and θ-nonoverlapping.*

Proof. First, suppose it is not the case that L_1L_2 is θ-$3'$-overhang-free and θ-nonoverlapping. Then there are $u, v \in L_1L_2$ such that $u = wx$ and $v = \theta(w)y$ for some $w \in X^+$ and $x, y \in X^*$.

For wx the choices are:

(1) $w = w_1w_2; w_1 \in L_1, w_2x \in L_2; w_1, w_2 \in X^+$,
(2) $w \in L_1, x \in L_2$,
(3) $x = x_1x_2, wx_1 \in L_1, x_2 \in L_2$ with $x_1, x_2 \in X^+$.

For $\theta(w)y$ the choices are:

(a) $w = w'w''; \theta(w') \in L_1, \theta(w'')y \in L_2$ with $w', w'' \in X^+$.
(b) $\theta(w) \in L_1, y \in L_2$.
(c) $y = y_1y_2; \theta(w)y_1 \in L_1, y_2 \in L_2$ with $y_1, y_2 \in X^+$.

Cases (1b), (1c), (2a), (2c), and (3) cannot occur because they all contradict L_1 being θ-$3'$-overhang-free. Case (2b) cannot occur because it would imply that $w, \theta(w) \in L_1$, but we have that L_1 is θ-nonoverlapping.

The remaining case is (1a). We have $w = w_1w_2 = w'w''$, $w_1, \theta(w') \in L_1$, $w_2x, \theta(w'')y \in L_2$. If $|w_1| = |w'|$, then $w_1 = w', \theta(w') = \theta(w_1)$ and so both $w_1, \theta(w_1) \in L_1$ – a contradiction. If $|w_1| > |w'|$, then $w_1 = w'r$, for some nonempty r. We have $w_1 = w'r \in L_1, \theta(w') \in L_1$. This contradicticts the hypothesis that L_1 is θ-$3'$-overhang-free. If $|w_1| < |w'|$, we get the same contradiction.

Now we show that $L_1 \cup L_1L_2$ is θ-$3'$-overhang-free and θ-nonoverlapping by contradiction. Suppose $wx \in L_1 \cup L_1L_2$ and $\theta(w)y \in L_1 \cup L_1L_2$ for some $w \in X^+$ and $x, y \in X^*$. By the above, it is sufficient to consider the following two cases: (a) $wx \in L_1$ and $\theta(w)y \in L_1L_2$, and (b) $\theta(w)y \in L_1$ and $wx \in L_1L_2$.

In the case (a), there are words $u_1 \in L_1$ and $u_2 \in L_2$ such that $\theta(w)y = u_1u_2$. Then, using a case distinction on the relation between $|\theta(w)|$ and

$|u_1|$, one can verify that the assumption of L_1 being θ-nonoverlapping and θ-3′-overhang-free is contradicted. For case (b), one can work analogously. □

The above proposition can be used to construct large classes of languages which are θ-3′-overhang-free and θ-nonoverlapping. For example, if the alphabet is $\Delta = \{A, C, G, T\}$ and θ is such that $\theta(A) = T$ and $\theta(C) = G$, then the language αL is θ-nonoverlapping and θ-3′-overhang-free for every symbol $\alpha \in \Delta$ and for every nonempty language $L \subseteq \Delta^+$.

The following corollary shows that, if θ is a morphic involution, given a θ-3′-overhang-free language L, the language obtained by taking arbitrary catenations of words from L is also θ-3′-overhang-free.

Corollary 2. *Let L be a nonempty subset of X^+ and let θ be a morphic involution. If L is θ-nonoverlapping and θ-3′-overhang-free, then also L^+ is θ-nonoverlapping and θ-3′-overhang-free.*

Proof. By Proposition 5, $L \cup LL^+$ is θ-nonoverlapping and θ-3′-overhang-free. Then, the claim follows by the fact that $L^+ = L \cup LL^+$. □

Proposition 6. *Let L_1, L_2 be nonempty subsets of X^+ and let θ be a morphic involution. If L_2 is θ-nonoverlapping and θ-5'overhang-free, then L_1L_2 is θ-5'overhang-free.*

Proof. Similar to that of the previous proposition. □

Corollary 3. *Let K be a nonempty subset of X^+ and let θ be a morphic involution. If K is θ-nonoverlapping and θ-5'overhang-free, then also K^+ is θ-nonoverlapping and θ-5'overhang-free.*

Proof. Similar to that of the previous corollary. □

In case the involution we deal with is anti-morphic, as is the case of the DNA involution, there are notably more requirements on languages in order to have their concatenation being overhang-free.

Proposition 7. *Let L_1, L_2 be nonempty subsets of X^+ and let θ be an anti-morphic involution. If L_1 is θ-compliant, θ-overhang-free, θ-nonoverlapping, L_2 is θ-3′-overhang-free, and $L_1 \cup L_2$ is θ-p-compliant, then L_1L_2 is θ-3′-overhang-free.*

Proof. Suppose L_1L_2 is not θ-3′-overhang-free. Then there exist $u, v \in L_1L_2$ such that $u = wx, v = \theta(w)y, w \in X^+$ so that $xy \neq 1$.

For wx the choices are:

(1) $w = w_1w_2; w_1 \in L_1, w_2x \in L_2; w_1, w_2 \in X^+$
(2) $w \in L_1, x \in L_2$

(3) $x = x_1x_2, wx_1 \in L_1, x_2 \in L_2$ with $x_1, x_2 \in X^+$

For $\theta(w)y$ the choices are:

(a) $y = y_1y_2; \theta(w)y_1 \in L_1, y_2 \in L_2$ with $y_1, y_2 \in X^+$
(b) $\theta(w) \in L_1, y \in L_2$
(c) $w = w'w''; \theta(w'') \in L_1, \theta(w')y \in L_2$ with $w', w'' \in X^+$

The cases (1ab), (2ac), (3bc) cannot occur because they contradict the hypothesis that L_1 is θ-compliant and θ-nonoverlapping. Case (3a) cannot occur because L_1 is θ-overhang-free. The case (2b) cannot occur since it contradicts L_1 being θ-nonoverlapping. The remaining case is (1c), where $w = w_1w_2 = w'w''$, $w_1 \in L_1, w_2x \in L_2$ and $\theta(w'') \in L_1, \theta(w')y \in L_2$.

If $|w_1| = |w'|$, then $w_1 = w'$ and $w_2 = w''$. We have $xy \neq 1$. Hence either $x \neq 1$ or $y \neq 1$. If $y \neq 1$, then $\theta(w')y, w' \in L_1 \cup L_2$ gives a contradiction, since $L_1 \cup L_2$ is θ-p-compliant. If $x \neq 1$, then $w_2x, \theta(w_2) \in L_1 \cup L_2$ also gives a contradiction.

If $|w_1| > |w'|$, then there exists $r \neq 1$ such that $w_1 = w'r$ and $w'' = rw_2$. Hence $w_1 = w'r \in L_1$, $\theta(w'') = \theta(rw_2) = \theta(w_2)\theta(r) \in L_1$. This contradicts L_1 being θ-overhang-free.

If $|w_1| < |w'|$, then there exists $r \neq 1$ such that $w' = w_1r$ and $w_2 = rw''$. Hence $\theta(w')y = \theta(w_1r)y = \theta(r)\theta(w_1)y \in L_2$, $w_2x = rw''x \in L_2$. This contradicts L_2 being θ-3′-overhang-free. □

Proposition 8. *Let L_1, L_2 be nonempty languages in X^+ and let θ be an anti-morphic involution. If L_2 is θ-compliant, θ-overhang-free, $L_2 \cap \theta(L_2) = \emptyset$, L_1 is θ-5'overhang-free, and $L_1 \cup L_2$ is θ-s-compliant, then L_1L_2 is θ-5'overhang-free.*

Proof. Similar to that of the above proposition. □

Let L_1 and L_2 be two DNA languages over Δ and consider τ, the DNA involution. The following result gives sufficient conditions for the concatenation of L_1 and L_2 to be overhang-free.

Corollary 4. *Let $L_1 \subseteq \Delta^+$ and $L_2 \subseteq \Delta^+$ be two nonoverlapping, overhang-free, DNA compliant languages with respect to τ, the DNA involution. If $L_1 \cup L_2$ is both p- and s-compliant, then L_1L_2 is overhang-free.*

4 Languages invariant under bio-operations

Until now we have considered static properties of languages, i.e. conditions ensuring that the initial DNA language encoding the input to a problem has good encoding properties. A necessary next step is to determine how to ensure that the computational steps performed on the language of initial

encodings do not alter these properties. In this section we investigate several bio-operations and study their effect on properties of DNA languages.

We model such bio-operations using multiset notation. The reason for using multisets is as follows. Besides the differences mentioned in Section 2, there is another important difference between electronic and DNA-based computations. While in electronic computing the computation process is deterministic and it involves individual elements, in DNA computing we deal with populations of DNA strands (each DNA strand is present in thousands or millions of copies) and the results of a bio-operation are obtained stochastically. This means that, in the case of DNA computing, the number of available copies of each strand is very important. In mathematical terms this translates in using, instead of sets, the notion of multiset.

A multiset M (over the alphabet X) is a mapping of X^* into $\mathbf{N} \cup \{\infty\}$. Intuitively, for a word w, $M(w)$ is the number of copies of w in M. The multiset is finite if $\sum_{w \in X^*} M(w)$ is finite. The language of all the (distinct) words in M is called the *support* of M and is defined as $\mathrm{supp}(M) = \{w \in X^* \mid M(w) > 0\}$. Let M_1 and M_2 be two multisets. We write $M_1 \subseteq M_2$ if $M_1(w) \leq M_2(w)$ for every word w. We use the notation $M_1 \cup M_2$ for the multiset defined by $(M_1 \cup M_2)(w) = M_1(w) + M_2(w)$ for all words w. Similarly, the multiset $M_1 \setminus M_2$ is defined by $(M_1 \setminus M_2)(w) = \max\{0, M_1(w) - M_2(w)\}$ for all words w. The finite multiset that consists of the words $w_1, \ldots, w_k$, where k is a non-negative integer, is denoted by $\langle w_1, \ldots, w_k \rangle$. Note that these words are not necessarily distinct. The set of all finite multisets is denoted by $\mathcal{M}_{\mathrm{fin}}$.

Having defined a multiset, we proceed now to define the bio-operations we will consider in this paper: cut, paste, splice, contextual deletion and contextual insertion. Cutting and pasting of DNA double-strands is accomplished by restriction enzymes [17]: each enzyme can cut only at a specific subsequence called restriction site, and it cuts the strand in a predetermined way, either in two blunt-ended pieces or in two pieces with sticky-ends. Another enzyme, called ligase, can accomplish the reverse bio-operation of pasting together two double DNA strands with compatible ends. Splicing is a combination of cut and paste: two DNA strands can be cut, and the prefix of one recombines with the suffix of the other and vice-versa. Contextual insertion and deletion of a DNA strand at a specific location in another DNA strand can be accomplished using a technique called site-specific oligonucleotide mutagenesis.

Note that these bio-operations act on either single strands or double strands, while the properties we have studied for the initial DNA language of a problem deal only with the case of single DNA strands. The reason why this does not restrict the generality of our results is that, even if the initial data is encoded in double strands, most likely during the computation

the double-strands will be denatured and may interact with one another as single-strands.

This section studies the questions of whether we can find algorithms to decide if a given DNA language L is invariant under some bio-operations, where invariance means that the results of a bio-operation acting on words in L belong to the same language L. These notions can be applied as follows. Define, like in [14], the computation language of a DNA-based system as the (multi)set of all words that can appear at any time during a computation. If we prove that a given computation language is invariant under some bio-operations then we know that the good initial encoding properties will be preserved under any computation involving those bio-operations.

The results obtained in this section give algorithms for deciding invariance under various bio-operations and calculate their space and time complexities. The problems turn out to be undecidable for context-free languages. The last proposition of the section gives a method of obtaining invariant and sticky-free languages of the form K^*, where K is a comma-free code. Such languages have the property that they are sticky-free and, moreover, this property is preserved under any of the bio-operations considered.

Definition 1. *A word operation is a partial mapping* $op : \mathcal{M}_{\text{fin}} \to 2^{\mathcal{M}_{\text{fin}}}$*; that is,* $op(M)$ *is a set of finite multisets, for every* M *in* $\operatorname{dom} op$*.*

Intuitively, a word operation op takes a multiset of words, say M, alters some (or all) of these words and results in a new multiset N in $op(M)$. In general the operation is nondeterministic: it is possible to obtain a multiset N' in $op(M)$ which is different from N. Word operations are usually denoted by expressions that are called *rules*. If r is a rule then op_r is the operation represented by r. Here we consider the following five types of rules for an alphabet X that does not contain the symbols \$ and $\#$.

- A *splicing rule* r is an expression of the form $x_1\#y_1\$x_2\#y_2$, where x_1, y_1, x_2 and y_2 are in X^*. Then, $\operatorname{dom} op_r = \{\langle p_1x_1y_1s_1, p_2x_2y_2s_2\rangle \mid p_1, s_1, p_2, s_2 \in X^*\}$ and $op_r\langle w_1, w_2\rangle = \{\langle p_1x_1y_2s_2, p_2x_2y_1s_1\rangle \mid p_1, s_1, p_2, s_2 \in X^*, w_1 = p_1x_1y_1s_1, w_2 = p_2x_2y_2s_2\}$ – see [13] and [17] for details on DNA systems based on splicing operations.
- A *paste rule* r is an expression of the form $x\$y$, where x and y are in X^*. Then, $\operatorname{dom} op_r = \{\langle px, ys\rangle \mid p, s \in X^*\}$ and $op_r\langle w_1, w_2\rangle = \{\langle pxys\rangle \mid p, s \in X^*, w_1 = px, w_2 = ys\}$.
- A *cut rule* r is an expression of the form $x\#y$, where x and y are in X^*. Then, $\operatorname{dom} op_r = \{\langle pxys\rangle \mid p, s \in X^*\}$ and $op_r\langle w\rangle = \{\langle px, ys\rangle \mid p, s \in X^*, w = pxys\}$.

- A *contextual deletion rule* r is an expression of the form $x\#v\#y$, where x, v and y are in X^*. Then, $\mathrm{dom}\, op_r = \{\langle pxvys\rangle \mid p, s \in X^*\}$ and $op_r\langle w\rangle = \{\langle pxys, v\rangle \mid p, s \in X^*, w = pxvys\}$.
- A *contextual insertion rule* r is an expression of the form $x\$v\$y$, where x, v and y are in X^*. Then, $\mathrm{dom}\, op_r = \{\langle pxys, v\rangle \mid p, s \in X^*\}$ and $op_r\langle w, v\rangle = \{\langle pxvys\rangle \mid p, s \in X^*, w = pxys\}$. The significance of contextual insertion/deletion operations in terms of computability is explored in [19].

The above operations can be extended to larger domains when many rules are available. More specifically, if R is a nonempty set of rules then the operation op_R is defined by $\mathrm{dom}\, op_R = \bigcup_{r\in R} \mathrm{dom}\, op_r$ and $op_R(M) = \bigcup_{r\in R} op_r(M)$, for every multiset M in $\mathrm{dom}\, op_R$.

If op is a word operation and L is a language, then $\overline{op}(L)$ is the set of all words that are obtained when op is applied to any multiset of words in L. More formally,

$$\overline{op}(L) = \{w \mid \exists\, M \in \mathrm{dom}\, op\,, \exists\, N \in op(M) : \mathrm{supp}(M) \subseteq L,\ w \in \mathrm{supp}(N)\}.$$

For example, if $L = \{b, aaa, abaa\}$ and r is the contextual insertion rule $a\$b\$a$ then $\overline{op}_r(L) = \{abaa, aaba, ababa\}$, but $\overline{op}_r(L \setminus \{b\}) = \emptyset$. Now consider a word operation op and a language L. For every nonnegative integer n, define $\overline{op}^n(L)$ to be L if $n = 0$, or $\overline{op}(\overline{op}^{n-1}(L))$ if $n > 0$. Then, we write $\overline{op}^+(L)$ for $\bigcup_{n=1}^{\infty} \overline{op}^n(L)$ and $\overline{op}^*(L)$ for $\bigcup_{n=0}^{\infty} \overline{op}^n(L)$. With this notation the following statement can be proved easily.

Remark 1. Let L be a language and let op be a word operation. If $\overline{op}(L) \subseteq L$ then $\overline{op}^+(L) \subseteq \overline{op}(L)$ and $\overline{op}^*(L) \subseteq L$. Also, if R is a set of rules then $\overline{op}_R(L) = \bigcup_{r\in R} \overline{op}_r(L)$.

If M and N are multisets and r is a rule, we write $M \Longrightarrow_r N$ if there are M_1 in $\mathrm{dom}\, op_r$ and N_1 in $op_r(M_1)$ such that $M_1 \subseteq M$ and $N = (M \setminus M_1) \cup N_1$; that is, N obtains from M by applying the operation op_r on some words in M. For example, for $r = a\$b\$a$, both of the following hold true: $\langle b, aaa, abaa\rangle \Longrightarrow_r \langle abaa, abaa\rangle$ and $\langle b, aaa, abaa\rangle \Longrightarrow_r \langle aaa, ababa\rangle$.

Now let us define a multiset system (without output). (In this section we are only interested in the reliability of DNA computations rather than the results of these computations; therefore we omit the mechanism for identifying the output (or terminal) words. The reader is refered to [23] for various DNA computing models). A multiset system is a triple $\mu = (X, A, R)$, where X is an alphabet, R is a set of rules and A is a multiset, called the initial multiset of μ. Then, for a multiset B, we write $A \Longrightarrow_\mu^* B$ if there is a non-negative integer n and multisets $M_0, \ldots, M_n$ such that

$A = M_0$, $B = M_n$, and there is a rule r_i in R such that $M_i \Longrightarrow_{r_i} M_{i+1}$, for all $i = 0, \ldots, n-1$. The *computation language* of a multiset system μ is the language $\{w \mid w \in \mathrm{supp}(B)$ and $A \Longrightarrow_{\mu}^{*} B$, for some multiset $B\}$. To prevent failures in computations, such as undesirable bonds between words, we need to ensure that the computation language satisfies certain 'good' combinatorial properties (θ-free, θ-sticky-free, etc.). More specifically, if L is a language with 'good' properties, we wish L to be closed under every operation op_r, where r is any rule of μ. In this case, assuming $\mathrm{supp}(A) \subseteq L$, it follows that $\mathrm{supp}(B) \subseteq L$ for every multiset B such that $A \Longrightarrow_{\mu}^{*} B$.

Definition 2. *Let op be a word operation. A language L is called invariant under op, or simply op-invariant, if* $\overline{op}(L) \subseteq L$.

Next we consider the problem of deciding whether a given language is invariant under a given word operation. When the given language is regular, the problem is decidable in time polynomial with respect to the input size. In this case, the input consists of a finite automaton A and a rule r. The size of a rule r is simply the length of the word r – see below for the size of the automaton. On the other hand, when the given language is context-free the problem is undecidable.

Proposition 9. *The following problem is decidable in polynomial time.*

- *Input: A rule r and a complete deterministic finite automaton A.*
- *Output: YES/NO depending on whether* $L(A)$ *is invariant under the operation* op_r.

In particular, if r is a paste rule the problem is decidable in time $O(|A|^2|r|)$. *If r is a splicing rule, it is decidable in time* $O(|A|^4|r|)$. *If r is a cut, contextual insertion, or contextual deletion rule then the problem is decidable in time* $O(|A|^2|r|)$ *or* $O(|A|^3|r|)$, *depending on the algorithm used.*

The proof of the statement uses, repeatedly, the following basic concepts and results which involve constructions of automata. We use [28] as reference. Although most statements about the sizes of automata are not specified explicitly in [28], they follow easily from the given constructions.

1 A (nondeterministic) finite automaton A is a quintuple (X, Q, δ, s, F), where X is the input alphabet, Q is the state alphabet, s is the start state, F is the set of final states, and δ is the (finite) set of transitions. A transition is a word of the form pvq, where p and q are state symbols and $v \in X \cup \{1\}$. Such a transition says that at state p the automaton can enter state q if the current input symbol is $v \in X$, or unconditionally if $v = 1$. The size of the transition pvq is equal to $|pvq| = 2 + |v|$. The size $|A|$ of a finite automaton A is the sum of the sizes of its transitions. We assume throughout that in every finite automaton all states are reachable

from the start state and, therefore, the number of states of an automaton A is smaller than $|A|$. The automaton A is deterministic if for every transition pvq one has $v \in X$ and there is no transition pvq' with $q \neq q'$. Such an automaton is *complete* if for every state p and for every $a \in X$ there is a transition paq in δ.

2 Given a complete deterministic finite automaton A, one can construct, in time $O(|A|)$, a complete deterministic finite automaton $\bar{A}$ such that $L(\bar{A}) = X^* \setminus L(A)$ and $|\bar{A}| = |A|$.

3 Given two finite automata A_1 and A_2, one can construct, in time $O(|A_1| + |A_2|)$, a finite automaton A_1A_2 such that $L(A_1A_2) = L(A_1)L(A_2)$ and $|A_1A_2| = O(|A_1| + |A_2|)$.

4 Given two finite automata A_1 and A_2, one can construct, in time $O(|A_1| + |A_2|)$, a finite automaton $A_1 \cup A_2$ such that $L(A_1 \cup A_2) = L(A_1) \cup L(A_2)$ and $|A_1 \cup A_2| = O(|A_1| + |A_2|)$.

5 Given a finite automaton A_1 and a deterministic finite automaton A_2, one can construct, in time $O(|A_1||A_2|)$, a finite automaton $A_1 \cap A_2$ such that $L(A_1 \cap A_2) = L(A_1) \cap L(A_2)$ and $|A_1 \cap A_2| = O(|A_1||A_2|)$. This can be achieved using a Cartesian product construction.

6 Given an automaton A, a state q of A, and a set P of states of A, one can construct, in time $O(|A|)$, an automaton $A_{q,P}$ such that $|A_{q,P}| = |A|$ and $L(A_{q,P})$ consists of all the words that can be accepted by A when q is used as the start state and P is used as the set of final states. Note that, for every words x and y, $xy \in L(A)$ if and only if there is a state q of A such that $x \in L(A_{s,q})$ and $y \in L(A_{q,F})$, where s is the start state of A and F is the set of final states of A.

7 A finite transducer T is a sixtuple $(X, \Gamma, Q, \delta, s, F)$, where X, Q, s and F are as in finite automata, Γ is the output alphabet, and δ is the (finite) set of transitions – in this paper we only consider transducers with $X = \Gamma$. A transition is a word of the form px/yq, where p and q are states, x is an input word, and y is an output word. Such a transition says that at state p the transducer can enter state q and output y if the current input is $x \in X^+$, or unconditionally if $x = 1$. The size of a transducer T is the sum of the sizes of its transitions, where the size of a transition px/yq is $3 + |xy|$. Given a finite transducer T and a complete deterministic finite automaton A, one can construct, in time $O(|T||A|)$, a finite automaton TA such that $|TA| = O(|T||A|)$ and $L(TA) = T(L(A))$ – see [28]; that is, a word v is in $L(TA)$ if and only if there is a word $w \in L(A)$ such that, on input w, the transducer T can reach a final state and output v.

8 Given a finite automaton A one can decide in time $O(|A|)$ whether or not $L(A)$ is empty. This problem is equivalent to deciding whether there is a path from the start state to a final state of the graph (automaton) A.

This task can be performed in time $O(|A|)$ using a depth-first search algorithm that would begin with the start state of the graph – see [21], for instance.

Proof. The main steps of the algorithm are as follows.

(a) Construct a finite automaton A_r such that $L(A_r) = \overline{op}_r(L(A))$.
(b) Construct the automaton $A_r \cap \bar{A}$.
(c) Output YES/NO depending on whether $L(A_r \cap \bar{A})$ is empty.

First we consider steps (b) and (c). Deciding whether $\overline{op}_r(L(A)) \subseteq L(A)$ is equivalent to deciding whether $L(A_r) \cap L(\bar{A}) = \emptyset$ which can be performed in time $O(|A_r \cap \bar{A}|)$. As the size of the automaton $A_r \cap \bar{A}$ is $|A_r||A|$, the time complexity of the algorithm is $O(|A_r||A|)$. It remains now to find the size of the automaton A_r – the time to compute $A_r \cap \bar{A}$ is $O(|A_r||A|)$. For the case where r is a paste rule, we show that $|A_r| = O(|A||r|)$ and, therefore, the algorithm runs in time $O(|A|^2|r|)$. For the cases where r is a cut, a contextual insertion, or a contextual deletion rule, we show two possible constructions of A_r, one based on transducers and one based on automata of the form $A_{q,P}$. The transducer-based construction produces an automaton A_r of size $O(|A||r|)$. In this case, the time complexity of the algorithm is $O(|A|^2|r|)$. The second type of construction gives an automaton A_r the size of which is $O(|A|^2|r|)$ in the worst case and, therefore, the algorithm would run in time $O(|A|^3|r|)$. Finally, when r is a splicing rule, we can offer only a construction of the second type which produces an automaton A_r of size $O(|A|^3|r|)$ in the worst case. In this case, the time complexity is $O(|A|^4|r|)$. In each case, for the correctness of the construction, one needs to show $L(A_r) = \overline{op}_r(L(A))$. We only do this for the case where r is a splicing rule – this is the most involved case.

The constructions will involve the following automata, for $z \in X^*$. The automaton B_z of size $O(|z|)$ such that $L(B_z) = X^*z$; the automaton C_z of size $O(|z|)$ such that $L(C_z) = zX^*$; the automaton D_z of size $O(|z|)$ such that $L(D_z) = \{z\}$; and the automaton $A_\emptyset$ of size $O(1)$ such that $L(A_\emptyset) = \emptyset$. For the input automaton A we write s for its start state and F for the set of final states.

Splicing rule: $r = x_1\#y_1\$x_2\#y_2$. For each state q, construct the automata $A_{s,q} \cap B_{x_1}$, $A_{q,F} \cap C_{y_1}$, $A_{s,q} \cap B_{x_2}$, $A_{q,F} \cap C_{y_2}$. Let P_1 be the set of states q for which both $L(A_{s,q} \cap B_{x_1})$ and $L(A_{q,F} \cap C_{y_1})$ are non-empty. Let P_2 be the set of states q for which both $L(A_{s,q} \cap B_{x_2})$ and $L(A_{q,F} \cap C_{y_2})$ are non-empty. Then, construct the automaton

$$A_r = \bigcup_{q_1 \in P_1, q_2 \in P_2} ((A_{s,q_1} \cap B_{x_1})(A_{q_2,F} \cap C_{y_2}) \cup (A_{s,q_2} \cap B_{x_2})(A_{q_1,F} \cap C_{y_1})).$$

The size of A_r is $O(|P_1||P_2|(|A||x_1| + |A||y_2| + |A||x_2| + |A||y_1|))$. Moreover, as the sizes of P_1 and P_2 are bounded by $|A|$, it follows that $|A_r| =$

$O(|A|^3|r|)$ in the worst case. The total time of the construction is $O(|A|^3|r|)$: $O(|A|(|A||x_1| + |A||y_1| + |A||x_2| + |A||y_2|))$ for computing P_1 and P_2 plus $O(|A_r|)$ for constructing A_r. We now show that $L(A_r) = \overline{op}_r(L(A))$. First, assume $w \in \overline{op}_r(L(A))$. Then, there are words in $L(A)$ of the form $p_1x_1y_1s_1$ and $p_2x_2y_2s_2$ such that $w = p_1x_1y_2s_2$ or $w = p_2x_2y_1s_1$. Without loss of generality assume $w = p_1x_1y_2s_2$. As $p_1x_1y_1s_1$ is in $L(A)$ there is a state q_1 of A such that $p_1x_1 \in L(A_{s,q_1})$ and $y_1s_1 \in L(A_{q_1,F})$. Hence, there is a state q_1 such that both $L(A_{s,q_1} \cap B_{x_1})$ and $L(A_{q_1,F} \cap C_{y_1})$ are non-empty which implies $q_1 \in P_1$. Similarly, there is a state $q_2 \in P_2$ such that $y_2s_2 \in L(A_{q_2,F} \cap C_{y_2})$. But, as $w = p_1x_1y_2s_2$, it follows that $w \in L((A_{s,q_1} \cap B_{x_1})(A_{q_2,F} \cap C_{y_2}))$ and, therefore, $w \in L(A_r)$. Now assume $w \in L(A_r)$. Then, there are states $q_1 \in P_1$ and $q_2 \in P_2$ such that $w \in L((A_{s,q_1} \cap B_{x_1})(A_{q_2,F} \cap C_{y_2}))$ or $w \in L((A_{s,q_2} \cap B_{x_2})(A_{q_1,F} \cap C_{y_1}))$. Without loss of generality we only consider the first case which implies $w = p_1x_1y_2s_2$, for some $p_1, s_2 \in X^*$, and $p_1x_1 \in L(A_{s,q_1} \cap B_{x_1})$ and $y_2s_2 \in L(A_{q_2,F} \cap C_{y_2})$. As $q_1 \in P_1$, $L(A_{q_1,F} \cap C_{y_1}) \neq \emptyset$ which, together with $p_1x_1 \in L(A_{s,q_1})$, implies $p_1x_1y_1s_1 \in L(A)$, for some $s_1 \in X^*$. Similarly, $p_2x_2y_2s_2 \in L(A)$, for some $p_2 \in X^*$. Then, as $\langle p_1x_1y_1s_1, p_2x_2y_2s_2\rangle \in \operatorname{dom} op_r$, one has $w \in \overline{op}_r(L(A))$ as required.

Paste rule: $r = x\$y$. First, construct the automata B_x and C_y. Then, construct the automaton $A_r = (A \cap B_x)(A \cap C_y)$ whose size is $O(|A||x| + |A||y|)$ or, equivalently, $O(|A||r|)$. Clearly the time required to construct A_r is also $O(|A_r|)$. It follows now that $L(A_r) = \overline{op}_r(L(A))$ as required.

Cut rule: $r = x\#y$. The transducer-based construction works as follows. First, construct transducers T_1 and T_2 whose input language is X^*xyX^* such that $|T_1| = O(|r|)$ and $|T_2| = O(|r|)$ and, for every u, w in X^*, $u \in T_1(w)$ if and only if $w = pxys$ and $u = px$, and $u \in T_2(w)$ if and only if $w = pxys$ and $u = ys$. Then, construct the automata T_1A and T_2A, each of size $O(|A||r|)$. The required automaton A_r is $T_1A \cup T_2A$ and its size is $O(|T_1A| + |T_2A|)$ or, equivalently, $O(|A||r|)$. Moreover, the time required for the construction is $O(|A||r|)$. The second construction works as follows. First, construct the automata B_x and C_y. Then, for each state q of the automaton A, construct the automata $A_{s,q} \cap B_x$ and $A_{q,F} \cap C_y$. Obviously, the total size of those automata is $O(|A||r|)$. For each state q, test whether both $L(A_{s,q} \cap B_x)$ and $L(A_{q,F} \cap C_y)$ are non-empty and, in this case, add q to an initially empty set P. Note that $|P|$ is no greater than the number of states of A and, therefore, $|P| = O(|A|)$. Also, P can be computed in time $O(|A|(|A||x| + |A||y|))$ which is $O(|A|^2|r|)$. Then, the required automaton A_r is

$$\bigcup_{q \in P} ((A_{s,q} \cap B_x) \cup (A_{q,F} \cap C_y)).$$

The time complexity of the construction is $O(|A|^2|r|)$ in the worst case.

Contextual insertion rule: $r = x\$v\$y$. First, test whether v is in $L(A)$. If not, the rule cannot be applied on the words of $L(A)$ and A_r is simply $A_\emptyset$. If v is in $L(A)$ we offer again two constructions for A_r. The transducer-based construction is as follows. First, construct a transducer T of size $O(|r|)$ whose input language is X^*xyX^* such that, for every words u and w, one has $u \in T(w)$ if and only if $u = pxvys$ and $w = pxys$ for some $p, s \in X^*$. Then, $A_r = TA$ and the time for the construction is $O(|A||r|)$. The second construction is as follows. First, as in the case of the cut rule, construct the set P of all states q such that both $L(A_{s,q} \cap B_x)$ and $L(A_{q,F} \cap C_y)$ are non-empty. Then,

$$A_r = \bigcup_{q \in P} ((A_{s,q} \cap B_x)D_v(A_{q,F} \cap C_y)).$$

The time complexity of the construction is $O(|A|^2|r|)$ in the worst case.

Contextual deletion rule: $r = x\#v\#y$. The transducer based construction is as follows. First, construct a transducer T whose input language is X^*xvyX^* such that $|T| = O(|r|)$ and, for every $u, w \in X^*$, $u \in T(w)$ if and only if $w = pxvys$ and $u = pxys$, for some $p, s \in X^*$. Then, construct the transducer TA. If $L(TA)$ is not empty let $A_r = TA \cup D_v$; else, A_r is equal to $A_\emptyset$ (in this case, the operation op_r is not applicable to any word in $L(A)$). Hence, $|A_r| = O(|T||A| + |v|)$ or, equivalently, $|A_r| = O(|A||r|)$. This construction requires time $O(|A||r|)$. The second type of construction is as follows. For each state q of A, compute the state q' at which A arrives when it runs on input v starting at state q. Let P be the set of pairs (q, q') thus computed. One has that $|P| = O(|A|)$ and the construction of P requires time $O(|P||v|)$, or $O(|A||v|)$ in the worst case. Now, for each pair (q, q') in P, construct the automata $A_{s,q} \cap B_x$ and $A_{q',F} \cap C_y$ and add the pair (q, q') in an initially empty set S if both $L(A_{s,q} \cap B_x)$ and $L(A_{q',F} \cap C_y)$ are non-empty. It should be clear that $S \subseteq P$ and, therefore, $|S| = O(|A|)$. Now if S is empty then the rule $x\#v\#y$ is not applicable to any word in $L(A)$ and the automaton A_r is equal to $A_\emptyset$. On the other hand, if S is not empty then A_r is the union of the automata D_v and $\bigcup_{(q,q') \in S}(A_{s,q} \cap B_x)(A_{q',F} \cap C_y)$. In any case, $|A_r| = O(|v| + |S|(|A||x| + |A||y|))$ which implies $|A_r| = O(|A|^2|r|)$. The total time of the construction is $O(|A||v|)$ for computing P plus $O(|P|(|A||x| + |A||y|))$ for computing S, plus $O(|v| + |A|^2|xy|)$ for constructing A_r. Thus, the total time is bounded by $O(|A|^2|r|)$. □

Proposition 10. *The following problem is undecidable.*

- *Input: A cut rule r and a context-free grammar G.*
- *Output: YES/NO depending on whether the language $L(G)$ is invariant under the operation op_r.*

Proof. The proof involves a few tools from CF grammars (context-free grammars) – see, for instance, [27] for details on CF grammars. Let us call the problem in the proposition Π. We shall reduce the following problem, call it Π_1, to Π:

- *Input:* A CF grammar G with $L(G) \cap X^*aX^* \neq \emptyset$ for all $a \in X$.
- *Output:* YES/NO depending on whether $X^* = L(G)$.

Note that Π_1 is a restricted version of the following undecidable problem, call it Π_1': Decide whether $X^* = L(G)$, for given CF grammar G. We argue first that Π_1 is indeed undecidable by reducing Π_1' to Π_1. The argument consists of two parts:

1. One can decide, for given CF grammar G and given $a \in X$, whether $L(G) \cap X^*aX^* \neq \emptyset$. This follows from the fact that the language $L(G) \cap X^*aX^*$ is context-free and the emptiness problem for context-free languages is decidable.

2. Now assume there is an algorithm deciding Π_1. We obtain a contradiction by deciding Π_1' as follows. First test whether there is a symbol a such that $L(G) \cap X^*aX^* = \emptyset$. If there is, output NO. Else, use the hypothetical algorithm for Π_1 to decide whether $X^* = L(G)$.

Now we proceed to showing that Π, the problem in question, is undecidable by reducing Π_1 to Π. So let G be a CF grammar with $L(G) \cap X^*aX^* \neq \emptyset$ for all $a \in X$. Consider the set of rules $R = \{a\#1, 1\#a, 1\$a \mid a \in X\}$. We show that $L(G) = X^*$ if and only if $\overline{op}_r(L(G)) \subseteq L(G)$ for all rules $r \in R$, or using Remark 1, if and only if $\overline{op}_R(L(G)) \subseteq L(G)$. So first assume $L(G) = X^*$. Obviously then $\overline{op}_R(L(G)) \subseteq L(G)$. Now assume $\overline{op}_R(L(G)) \subseteq L(G)$. Then, $\overline{op}_R^*(L(G)) \subseteq L(G)$. It suffices to prove that X^* is a subset of $\overline{op}_R^*(L(G))$. So let $w \in X^*$. We use induction on $|w|$ to show $w \in \overline{op}_R^*(L(G))$. First suppose $|w| = 0$ and consider any word u in $L(G)$. If $w = u$ we are done; else, $u = u_1a$ for some $a \in X$ and $u_1 \in X^*$. In this case, by applying the rule $a\#1$ on the multiset $\langle u_1a \rangle$, one has $u_1a, 1 \in \overline{op}_R(L(G))$; therefore, $w \in \overline{op}_R^*(L(G))$. Now suppose $u \in \overline{op}_R^*(L(G))$ for all words u of length at most n, and consider the word w of length $n+1$. Then, $w = ua$ for some $u \in X^n$ and $a \in X$. Also, as $L(G) \cap X^*aX^* \neq \emptyset$, there is a word of the form u_1au_2 in $L(G)$. By applying the rule $a\#1$ on $\langle u_1au_2 \rangle$ and then the rule $1\#a$ on $\langle u_1a \rangle$ it follows that $a \in \overline{op}_R^*(L(G))$. Finally, as $u \in \overline{op}_R^*(L(G))$, the rule $1\$a$ can be applied on $\langle u, a \rangle$ to obtain $w \in \overline{op}_R^*(L(G))$ as required. □

We consider now a method of obtaining invariant and sticky-free languages of the form K^*, where $K \subseteq X^+$ is a comma-free code. This is a modification of a method used in [14] to obtain involution-free and splicing-invariant languages of the form K^* for some code K. The idea is as follows: When an operation cuts the word $w \in K^*$, the resulting words are also in

K^* if the cut point in w is at the beginning or at the end of a subword v of w with $v \in K$. This condition is satisfied when we permit only operations represented by K-delimited rules as defined below.

Let us define a *context* to be a pair of words, say $(x, y) \in X^* \times X^*$. The context is called *K-delimited* if $x \in X^*K$ or $y \in KX^*$. A rule r is called K-delimited if one of the following holds

- $r = x_1\#y_1\$x_2\#y_2$ and the contexts (x_1,y_1) and (x_2,y_2) are K-delimited.
- $r = x\#y$ and the context (x, y) is K-delimited.
- $r = x\$y$
- $r = x\#v\#y$ and $v \in K^+$.
- $r = x\$v\$y$ and $v \in K^+$ and the context (x, y) is K-delimited.

Proposition 11. *Let θ be an involution, let $K \subseteq X^+$ be a code, and let R be a set of K-delimited rules. If K is comma-free, θ-nonoverlapping and θ-sticky-free (or θ-free), then the language K^* is op_R-invariant, θ-nonoverlapping, and θ-sticky-free (or θ-free, respectively).*

Proof. If K is θ-nonoverlapping and θ-free then also K^* is θ-nonoverlapping and θ-free – see [14]. If K is θ-nonoverlapping and θ-sticky-free then also K^* is θ-nonoverlapping and θ-sticky-free – see Corollary 1. Now assume K is a comma-free code. Then, for every $v \in K$ and $x, y \in X^*$, $xvy \in K^*$ implies $x, y \in K^*$ – see [3]. Consider a rule $r \in R$ and a multiset M of words from K^+ such that $M \in \operatorname{dom} op_r$. If r is a cut rule of the form $x\#y$, then, as r is K-delimited, there are words $x_1, y_1 \in X^*$ and $v \in K$ such that $xy = x_1vy_1$. Moreover M is of the form $\langle px_1vy_1s\rangle$. As K is comma-free, both px_1 and y_1s are in K^*. Hence, $\overline{op}_r(K^*) \subseteq K^*$ as required. Similarly, one verifies that if r is a rule of another type then again $\overline{op}_r(K^*) \subseteq K^*$. □

There are probably other ways to achieve invariance of K^*, but the method of K-delimited rules allows us to use tools from the theory of codes. Hence, the primary motivation for using K-delimited rules is of a technical nature. For the interpretation on the other hand, consider a restriction enzyme with recognition site $x\#y$. If the strands are encoded using a code K that contains a word v such that $x \in X^*v$ or $y \in vX^*$, then the parts of the strand that will be cut by $x\#y$ are also in K^*. One only possible limitation of this argument is that one needs to find a code K with short enough codewords.

5 Invariant and sticky-free languages with error-detecting capabilities

In this section we build on Proposition 11 to obtain a method for constructing languages that, in addition to being invariant, nonoverlapping and

sticky-free, possess certain error-detecting capabilities. The language property of error-detection can be defined with respect to a particular channel, or (information) medium, which is capable of introducing errors in the transmitted/stored words of the language – see [20]. A channel, or medium, say γ, is a binary relation on X^*. If $(w, z) \in \gamma$, then we say that z can be obtained from w through γ, or that γ can transform w to z. When $(w, z) \in \gamma$ and $w \neq z$, we say that z can be obtained from w with errors. As most errors encountered when dealing with DNA strands are insertions, deletions or substitutions of individual bases, we will restrict our discussion to these types of errors. The main result of the section, Proposition 13, effectively constructs a solid code that is τ-nonoverlapping, τ-sticky-free, γ-error detecting for some channel γ, and asymptotically optimal regarding its rate of encoding information.

Here we shall consider the following channel, for any positive integer ℓ.

- $\mathbf{sid}(1, \ell)$: a pair (w, z) is in the channel if and only if w can be transformed to z using at most one symbol substitution, insertion, or deletion in every ℓ consecutive symbols of w. For example, both $(aaaaa, baaa)$ and $(aaaaa, baaaaa)$ are in $\mathbf{sid}(1, 3)$, but $(aaaaa, baab) \notin \mathbf{sid}(1, 3)$. An informal way for testing whether (w, z) is in the channel $\mathbf{sid}(1, \ell)$ is the following – see [20] for formal details: Insert special symbols in w to mark the positions of the errors in w to obtain z. If every two of these special symbols are separated by at least ℓ symbols of w then the errors are valid and, therefore, (w, z) is in the channel. For example, to test that $(aaaaa, baaa)$ is in $\mathbf{sid}(1, 3)$, one can use either of the expressions $\sigma_b aaa\delta aa$ and $\sigma_b aaaa\delta a$, where $\sigma_b a$ indicates substitution of a with b and δa indicates deletion of a.

A language L is called *error-detecting for* a channel γ if, for all words w and z in $L \cup \{1\}$, $(w, z) \in \gamma$ implies $w = z$. This property ensures that the channel cannot transform a word of the language to *another* word of the language. Thus, when a word z is retrieved from γ and z is in L then z must be correct. Here we are interested in languages of the form K^*, where K is a code. For certain channels γ and codes K, to ensure that K^* is error-detecting for γ it is sufficient to show that K is $(\gamma, 1)$*-detecting:* for all $v \in K^*$ and $z \in K$, if $(v, z) \in \gamma$ then $v = z$. We note that if K is $(\gamma, 1)$-detecting then K is error-detecting for γ.

Our method for constructing languages of the form K^* that possess the properties of invariance, sticky-freeness, and error-detection relies on the following statement which uses the notion of *solid code*. A code K is solid if it is an infix code and no proper and nonempty prefix of a word in K is also a suffix of a word in K.

Proposition 12. *Let ℓ be a positive integer, let γ be the channel $\mathbf{sid}(1,\ell)$, let θ be an involution, let K be a finite code the maximum word length of which is at most ℓ, and let R be a set of K-delimited rules. If K is solid, θ-nonoverlapping and θ-sticky-free (or θ-free) and $(\gamma,1)$-detecting then the language K^* is op_R-invariant, θ-nonoverlapping and θ-sticky-free (or θ-free, respectively), and error-detecting for γ.*

Proof. That K^* is op_R-invariant, θ-nonoverlapping and θ-sticky-free follows from Proposition 11 when we note that every solid code is a comma-free code. On the other hand, if a solid code K of maximum word length at most ℓ is $(\gamma,1)$-detecting then K^* is error-detecting for γ – see [20]. □

We continue with a construction of codes over Δ satisfying the premises of the above proposition. For a word w in Δ^* we define the parity symbols $\mathrm{p}_C(w)$ and $\mathrm{p}_G(w)$ as follows:

$$\mathrm{p}_C(w) = \begin{cases} A, & \text{if } |w|_A + |w|_C \text{ is odd;} \\ T, & \text{if } |w|_A + |w|_C \text{ is even.} \end{cases}$$

and

$$\mathrm{p}_G(w) = \begin{cases} A, & \text{if } |w|_A + |w|_G \text{ is odd;} \\ T, & \text{if } |w|_A + |w|_G \text{ is even.} \end{cases}$$

Lemma 2. *For every positive integer n and for every words x and s in Δ^* the code*

$$P_{x,s}(n) = \{xw\mathrm{p}_C(w)\mathrm{p}_G(w)s \mid w \in \Delta^n\}$$

is $(\gamma,1)$-detecting, where $\gamma = \mathbf{sid}(1, n + |xs| + 2)$.

Proof. Consider words $v \in P_{x,s}(n)^k$ and $z \in P_{x,s}(n)$, where k is a non-negative integer, such that $(v,z) \in \gamma$. We need to show that $v = z$. Clearly, $k \geq 1$. First we show that $k = 1$ and, therefore, $v \in P_{x,s}(n)$. Indeed, assume $k \geq 2$. As the number of deletions in v can be at most k, it follows that $|z| \geq |v| - k$. On the other hand, as all the words of $P_{x,s}(n)$ have the same length, one has $|v| = k|z|$ and, therefore, $|z| \geq k(|z| - 1)$. Then, the assumption $k \geq 2$ implies $|z| \leq 2$ which is impossible. Hence, $v \in P_{x,s}(n)$ and, therefore, $|z| = |v|$ which implies that no insertion or deletion error can be used to transform v to z. Now assume there is one substitution error in v to obtain z. By the definition of $P_{x,s}(n)$, there are words $w, y \in \Delta^n$ such that $v = xw\mathrm{p}_C(w)\mathrm{p}_G(w)s$ and $z = xy\mathrm{p}_C(y)\mathrm{p}_G(y)s$. Obviously the error in v cannot be in the subwords x and s. If the error is in $\mathrm{p}_C(w)$ then $\mathrm{p}_C(y) \neq \mathrm{p}_C(w)$. In this case, however, there is no error in w; therefore, y must be equal to w which implies $\mathrm{p}_C(y) = \mathrm{p}_C(w)$ – a contradiction. Hence, $\mathrm{p}_C(w) = \mathrm{p}_C(y)$ and similarly $\mathrm{p}_G(w) = \mathrm{p}_G(y)$. Now if the error is in the subword w of v, then $w = u_1\alpha u_2$ and $y = u_1\beta u_2$ for some words u_1, u_2 and symbols α, β with $\alpha \neq \beta$. This implies $|y|_\alpha = |w|_\alpha - 1$ and $|y|_\beta = |w|_\beta + 1$.

By considering all the possible values of α and β, one can verify that the above equations together with $\mathrm{p}_C(w) = \mathrm{p}_C(y)$ and $\mathrm{p}_G(w) = \mathrm{p}_G(y)$ lead to a contradiction. Hence, the only possibility remaining is that z is obtained from v with no errors; that is, $v = z$ as required. □

Proposition 13. *Let $x = (x_n)$ be a sequence of words in $C\{C, G\}^*G$ such that $\lim(1/|x_n|) = \lim(|x_n|/n) = 0$. For every positive integer n the code*

$$K_x(n) = \{x_nCw\mathrm{p}_C(w)\mathrm{p}_G(w)T \mid w \in \Delta^n, x_nCw \notin \Delta^+x_n\Delta^*\}$$

has the following properties.

1. *It is a solid code.*
2. *It is τ-nonoverlapping and τ-sticky-free, where τ is the DNA involution.*
3. *It is $\bar{\tau}$-nonoverlapping and $\bar{\tau}$-overhang-free, where $\bar{\tau}$ is the complementarity involution.*
4. *It is $(\gamma, 1)$-detecting, where $\gamma = \mathbf{sid}(1, |x_n| + n + 4)$.*
5. *The information rate of $K_x(n)$ tends to 1 as $n \to \infty$.*
6. *If $x_n = \tau(x_n)$ then $K_x(n)$ is τ-free.*

Proof. First assume there is a nonempty word u which is a proper prefix and suffix of the code. If $|u| \leq |x_nC|$ then $u \in \{C, G\}^+ \cap \Delta^*\{T\}$ which is impossible. If $|u| > |x_nC|$ then u starts with x_n and is a proper suffix of some codeword $x_nCw\mathrm{p}_C(w)\mathrm{p}_G(w)T$. This implies $x_nCw \in \Delta^+x_n\Delta^*$ – a contradiction. Hence, $K_x(n)$ is a solid code. Using a similar argument one can verify that the code is also τ-sticky-free. Moreover, as $\tau(K_x(n)) \subseteq \{A\}\Delta^+\{G\}$ and $\bar{\tau}(K_x(n)) \subseteq \{G\}\Delta^+\{A\}$, it follows that the code $K_x(n)$ is τ-nonoverlapping and $\bar{\tau}$-nonoverlapping as well. That the code is $\bar{\tau}$-overhang-free follows easily from the fact that every codeword starts with C and ends with T. Now, as $K_x(n) \subseteq P_{x_nC,T}(n)$, Lemma 2 implies that $K_x(n)$ is $(\gamma, 1)$-detecting.

For the second last part, consider the code $L_x(n)$ that consists of all the words w in Δ^n such that the symbols of w at positions $1, |x_n|, 2|x_n|, \ldots$ are not in $\{C, G\}$; that is, $L_x(n) = (\{A, T\}\Delta^{|x_n|-1})^{q_n}\{A, T\}\Delta^{r_n}$, where q_n and r_n are the unique integers satisfying $n - 1 = q_n|x_n| + r_n$. Then it follows that $L_x(n)$ is a subset of the set $\{w \mid w \in \Delta^n,\ x_nCw \notin \Delta^+x_n\Delta^*\}$ whose size is equal to $|K_x(n)|$. Hence, $|K_x(n)| \geq |L_x(n)|$ which implies

$$|K_x(n)| \geq (2 \cdot 4^{|x_n|-1})^{q_n} \cdot 2 \cdot 4^{r_n} = 4^{n-(1+q_n)/2}.$$

From the above and the assumptions about $|x_n|$, it follows that $\log_4 |K_x(n)|/(|x_n| + n + 4)$, the information rate of $K_x(n)$, tends to 1, as $n \to \infty$.

For the last part, assume there are words $w_1, w_2, w \in \Delta^n$ such that x_nCw_1, x_nCw_2, $x_nCw \notin \Delta^+x_n\Delta^*$ and

$$x_nCw_1\mathrm{p}_C(w_1)\mathrm{p}_G(w_1)Tx_nCw_2\mathrm{p}_C(w_2)\mathrm{p}_G(w_2)T \in \Delta^+z\Delta^+,$$

where $z = \tau(x_n C w \mathrm{p}_C(w) \mathrm{p}_G(w) T) = A\tau(\mathrm{p}_G(w))\tau(\mathrm{p}_C(w))\tau(w)Gx_n$. As z starts with the symbol A and ends with the symbol G, there must be a suffix s_1 of w_1 and a prefix p_2 of w_2 such that $|s_1| + |p_2| = n$ and $z = s_1 \mathrm{p}_C(w_1)\mathrm{p}_G(w_1) T x_n C p_2$. Then, there exists a suffix s of length $|p_2|$ of $\tau(w)$ such that $x_n C p_2 = sGx_n$ which implies that $x_n C w_2 \in \Delta^+ x_n \Delta^*$ – a contradiction. Hence, the code $K_x(n)$ is τ-free. □

6 Empirical tests and future directions

We conclude this paper with some empirical tests for checking whether certain DNA languages possess the good encoding properties defined in Section 2: *(a)* the sets of genes of various organisms, and *(b)* the set of encodings used in Adleman's experiment [1].

A gene is a section of DNA from a genome that contains information for the construction of a protein: it consists of coding regions (exons) alternating with introns (DNA sequences that do not code for proteins and will eventually be deleted). Consider a collection of genes of an organism (herein a gene will consist only of its coding regions, excluding the introns) to be a language over Δ. Call such a language a *gene language*. In this section we investigate whether several known gene languages satisfy the conditions (A)-(J) defined in Section 2 for data-encoding DNA languages. Gene languages as such do not exist in reality. The genes as present on the DNA sequence of a genome contain, besides their coding regions, also regions that do not code for any proteins. Moreover, the genome contains other sequences besides both the above-mentioned ones, for example promoters and enhancers of gene activity. The context of DNA sequences in a living cell is different from the DNA computing set-up in that all the genomic sequences are strung along the genome and are not separate sequences that may interact. However, as gene languages are DNA languages provided by nature, they seem a natural test-bed for our properties. Moreover, DNA computations may use DNA sequences that are parts of natural DNA and thus, sets of subwords of gene languages can become subsets of a DNA language used for bio-computations. Consequently, the results of these tests might give us an indication of the degree to which our formalizations are compatible with the biological reality.

In this section we will refer to conditions (A)-(F) in Section 2 as the τ-full conditions (stemming from a fact that a full match of some word is needed), (G)-(J) as the τ-partial conditions (only a partial match of some word is needed) and the entire collection of conditions as the τ-total conditions. Recall that τ is the formalization of the Watson-Crick DNA involution.

Before testing, we can advance conjectures whether gene languages which are, after all, different from and serve different purposes from DNA

languages used for computations, should have the defined "good" properties. There is a good chance that a gene language will satisfy the τ-full conditions, simply because of the length of the words in a gene language. For example, a gene language L_G should almost certainly be τ-nonoverlapping. Indeed, for a language not to be τ-nonoverlapping, we need to find two words in the language that are perfect Watson-Crick complements of each other. This implies, however, that the words are of the same length, which is unlikely to happen for L_G since it is unlikely that two genes have the exact same length. It is still possible, however, that L_G could fail to satisfy the τ-compliance condition since it is not unusual for one gene to be significantly shorter than another and therefore, at least in theory, its Watson-Crick complement could be a subword of another gene.

Regarding the τ-partial conditions, at first sight it would seem that any gene language would fail to satisfy these conditions. Indeed, such a conjecture seems supported by the fact that a partial binding of just a single nucleotide between the two genes would cause the whole language to fail the test. A closer look at the definition of a gene, however, reveals some additional facts that undermine that conjecture. A region of a DNA sequence that codes for a gene almost always starts with ATG and tends to end with either TAA, TAG or TGA (the STOP codons). Consequently, any pair of such sequences will successfully pass the τ-sticky-free test. This is true as most genes will have the form ATGu and thus, another gene, which is of one of the forms vTAA, vTAG or vTGA, would not bind to it in the fashion required.

Finally, note that the kind of argument that takes into account the start and end sequences of a gene fails to indicate whether or not a gene language will have the τ-3′-overhang, τ-5′overhang or τ-overhang-free properties. Indeed, for a pair of genes to fail the τ-3′-overhang-free test, for example, the beginning of a gene should start to bind with the middle of another and we cannot draw any conclusions as we have no information, in general, about the sequences inside of a gene.

Let us take a closer look at the conditions (A)-(J) which are intended to formalize undesirable bindings that occur between DNA sequences encoding information. There are at least two points of view from which these conditions could be refined to provide a more accurate reflection of biological reality.

Firstly, note that the τ-full conditions forbid certain DNA sequences to perfectly bind with others. In reality, however, two DNA single strands can bind to each other even if their overlapping regions are not perfect complements of each other. Instead of identical sequences, or perfect Watson-Crick complements, we usually deal with the more relaxed notion of *homology*. Informally, two sequences are called *homologous* if they are "almost identical", i.e., they differ by relatively few letters comparatively to their length.

To formalize this concept, we make use of the notion of Hamming distance.

Definition 3. *Given two words of equal length* $u, v \in \Delta^+$, $u = a_1 a_2 \dots a_n$, $v = b_1 b_2 \dots b_n$ *such that* $a_{i_1} \neq b_{i_1}, a_{i_2} \neq b_{i_2}, \dots, a_{i_k} \neq b_{i_k}$, $1 \leq i_j \leq n$, $1 \leq j \leq k$ *and* $a_j = b_j$ *for* $j \notin \{i_1, i_2, \dots, i_k\}$, *the Hamming Distance between* u *and* v *is* $H(u, v) = k$.

In other words, the Hamming Distance between two words of equal length is the number of positions where the two words differ. For example, $H(\texttt{ACC}, \texttt{GCC}) = 1$ and $H(\texttt{ACGT}, \texttt{TCGA}) = 2$.

We are now ready to define the notion of r-homologous words.

Definition 4. *Let* r *be a number with* $0 < r \leq 1$. *Two words of equal length* $u, v \in \Delta^+$ *are called* r*-homologous if* $\frac{H(u,v)}{|u|} \leq (1 - r)$.

Informally, two words of equal length are r-homologous if the number of nucleotides where they differ is at most $(1 - r)$ of their length. Note that if two words are 1-homologous, this means that they are identical. (In the analysis that follows, we used $r = 1$ and $r = 0.85$.) We can modify the definitions (A)-(F) to take homology into account. For example, condition (B) becomes:

> If $0 < r \leq 1$, a language L is τ-compliant with homology r iff $w, xuy \in L$, $|w| = |u|$, $\frac{H(u,\tau(w))}{|u|} \leq (1 - r)$ implies $xy = 1$.

Secondly, regarding conditions (G)-(J), the τ-partial conditions, note that according to the current definitions even a match of one letter would make them theoretically "stick" to each other and thus, fail the test. However, given that a gene language consists of relatively long sequences, a binding between subsequences of an empirically suggested length of at least $n = 9$ is required for the sequences to stick to each other in any reliable fashion.

We can now modify accordingly definitions (G)-(J) to take this observation into account. For example, condition (G) becomes:

> Let $n \geq 0$ be a natural number. A language L is τ-sticky-free of degree n iff $wx, y\tau(w) \in L$, $|w| \geq n$ implies $xy = 1$.

We may denote a τ-partial condition of degree n with a subscript to signify the degree. Thus, τ-sticky-free of degree n becomes τ_n-sticky-free.

Each genome we tested came with a list of *coding regions* that were extracted, and their union was taken to be the gene language defined by the genome. These coding regions are considered to be regions that code for some protein, although some coding regions are more well defined than others. For ease of testing, any coding region listed in the genome was taken to be a word in the corresponding gene language.

The test data for the programs was extracted from the genomes listed below.

- *Human Papilloma Virus* (HPV)
  HPV is a relatively small genome, but with the 63 different variations that have been mapped, it provides a large number of test cases that can be verified rapidly. Of the 63 genomes tested, there were an average of 7 coding regions per genome. The smallest coding region (of the entire collection) was 117 base pairs long, whereas the longest was 3132. The average length of a coding region was 1040 base pairs.
- Various *mitochondria*
  Mitochondria are small, specialized organisms (organelles) that live inside the cells of eukaryotes (organisms whose cells have a membrane enclosed nucleus, such as mammals). Thirteen different genomes were tested. On average, each genome contained 20 coding regions. Overall, the shortest coding region was 64 nucleotides in length, with the longest being 5223 nucleotides. The average number of nucleotides in a coding region was 786.
- *Drosophila*
  Drosophila is a type of fruit fly. The genome is relatively small but is much larger than that of HPV or mitchondrial genomes. It consists of 13912 coding regions. The coding regions (i.e., words) ranged from a length of 78 base pairs to 26415 base pairs. On average, the length of a coding region was 1503 base pairs.
- *Arabidopsis*
  Arabidopsis is a type of mustard plant. The genome of this organism is approximately three times larger than that of drosophila, making it a rather large genome. It consists of 25570 coding regions, ranging in length from 60 base pairs to 15417 base pairs. The average length of a coding region is 1289 base pairs.

For the drosophila genome, some coding regions were listed as having very short lengths. It is suspected that these short lengths were due to erroneous data (or erroneous interpretation of the data) in the genome file. When the tests were performed, all regions for all genomes with length less than 60 base pairs were omitted. The rationale for this is that a coding region (with no introns) of length 60 defines 20 amino acids. This does not form a very large protein. Most proteins are rather complicated, so even a limit of 20 amino acids for a protein seems unlikely.

All tests were performed on a single CPU system, running at 1 GHz. The τ-full conditions with homology r (denoted also by r-τ-conditions) were run with $r = 100\%$ and $r = 85\%$. For the τ_n-partial conditions, tests were only run with $n = 9$, because a binding of two DNA strands of length less than 9 is considered to be weak and therefore unlikely. Not all tests were run, because of prohibitively high running times. This is indicated either by the absence of the respective columns in the tables or by an ∞ sign. Future work

includes optimizing the algorithms used to achieve substantial reductions in running times.

Table 1. Gene language tests: Results for τ-full conditions. The numbers in parentheses indicate the length of the subword that is able to bind completely with another word

	τ-nonoverlapping	τ-compliant	τ-free
HPV	pass	pass	pass
Mitochondria	pass	pass	pass
Drosophila	pass	fail (294)	fail
Arabidopsis	pass	fail (132)	fail

Table 2. Gene language tests: Results for τ-full conditions with homology 85%

	85%-τ-nonoverlapping	85%-τ-p-compliant	85%-τ-s-compliant
HPV	pass	pass	pass
Mitochondria	pass	pass	pass
Drosophila	pass	∞	∞
Arabidopsis	pass	∞	∞

Table 3. Gene language tests: Results for τ_n-partial conditions with $n = 9$ (two strands will stick if they have matching ends of length at least 9). The numbers in parentheses indicate the lengths of the matching sequences

	τ_9-sticky-free	τ_9-$3'$-overhang-free	τ_9-$5'$overhang-free
HPV	pass	pass	1 fail
Mitochondria	pass	1 fail (9)	3 fail (9, 9, 341)
Drosophila	∞	∞	∞
Arabidopsis	∞	∞	∞

The failures with partial matches of length 9 for mitochondria in Table 3 are not nearly as significant as the match of length 341. This very long match was somewhat surprising considering the probability of such a match occuring. It would be interesting to investigate which coding regions matched up so well and what they represent, as this may provide some insight into why the match is so long.

It should be noted that the failures in Table 1 may be a result of the oversimplified approach of considering each coding region listed in the genome

as a word in the language. The coding region may be labelled in the original data as having special properties or may not in fact be a coding region. For future tests, these details will be investigated. This applies to all genomes being considered.

We now present the results of our tests on the DNA encodings used in Adleman's experiment, [1]. As expected, as such an in vitro DNA computation is exactly the situation where we would want the defined encoding properties to apply, we find that all the sequences used pass our refined tests.

The input to the Hamiltonian Path Problem is a directed graph with a designated input node and a designated output node. The question is whether the graph possesses a Hamiltonian Path, i.e., a path starting at the input node, ending at the output node, and passing through all the nodes exactly once. Adleman's DNA solution to the problem was to encode each node i in a DNA sequence $u_i v_i$, where $|u_i| = |v_i| = 10$ and encode each edge in a 20-letter long DNA sequence as follows. The (oriented) edge connecting nodes $u_i v_i$ and $u_j v_j$ was encoded as $\tau(v_i)\tau(u_j)$. Consequently, when putting together the edges and the nodes, all possible legal paths through the graph were formed. The DNA algorithm proceeded afterwards by a sequence of bio-operations intended to weed out the paths that were not Hamiltonian.

Table 4 shows the results of testing whether the set of nodes and the set of edges, taken separately and together, have the good encoding properties we have defined. As expected, the set of edges and nodes taken together fails all the tests except the τ-nonoverlapping one. This happens as, in this particular DNA algorithm, the very mechanism that solves the problem is based on the interaction by hybridization of nodes with edges. However, we would expect that, as nodes should not hybridize with nodes, neither edges with edges, the sets taken separately should pass the tests. The obtained results are presented in Table 4.

Table 4. Results of tests on sequences used in Adleman's DNA computing experiment [1]

	Edges	Nodes	Both
τ-nonoverlapping	pass	pass	pass
τ-compliant	pass	pass	fail
τ-free	pass	pass	fail
τ-sticky-free	fail	fail	fail
τ-3′-overhang-free	fail	fail	fail
τ-5′overhang-free	fail	fail	fail

We see that both the set of nodes and the set of edges fail the τ-sticky-free, τ-3′-overhang-free and τ-5'-overhang free tests. The reason is that the original definition causes a DNA language to fail a τ-partial test even in the

Table 5. Results of tests on Adleman's sequences with $n = 0$ to 5. A $-$ indicates a fail and $\surd$ indicates a pass. The tuple at a position i represents the result with $n = i$. For example, the set of edges when tested for τ_3-3′-overhang-free is a fail, but τ_4-3′-overhang-free is a pass

	Edges	Nodes	Both
τ_i-sticky–free	$(-,-,-,\surd,\surd,\surd)$	$(-,-,-,\surd,\surd,\surd)$	$(-,-,-,\surd,\surd,\surd)$
τ_i-3′-overhang-free	$(-,-,-,\surd,\surd,\surd)$	$(-,-,-,\surd,\surd,\surd)$	$(-,-,-,-,-,-)$
τ_i-5′-overhang-free	$(-,-,-,-,\surd,\surd)$	$(-,-,-,\surd,\surd,\surd)$	$(-,-,-,-,-,-)$

Table 6. Results of tests on the Adleman sequences with $r = 85\%$ for the τ-full conditions

	Edges	Nodes	Both
$85\% - \tau$-nonoverlapping	pass	pass	pass
$85\% - \tau$-compliant	pass	pass	pass
$85\% - \tau$-free	pass	pass	fail

case when two strands stick to each other by a single nucleotide. This is not a realistic assumption and Table 5 lists the results of testing Adleman's sequences with $n = 0$ up to $n = 5$. Note that $n = 4$ is the first value where the sets of DNA encodings pass all the tests, which is reasonable as overlaps of at most 3 nucleotides do not usually lead to undesired bindings of 20-nucleotide-long strands.

Table 6 contains the results of the τ-full tests for Adleman's sequences where the condition of "bind only if perfectly complementary" has been replaced by "bind if at most 15% of nucleotides differ". Note that the sets pass also this more realistic condition.

Overall, the results confirm that the choice of node and edge encodings was one that ensured that the computation in Adleman's experiment would proceed as planned, without unwanted hybridizations.

To conclude, the results of these empirical tests suggest that our definitions of good encoding properties, and especially their refinements, are an adequate model of the biological reality of computing with data encoded on DNA strands. Directions of further research include a detailed investigation of sticky-free and overhang-free languages with specified length of overlap, and of DNA compliant and free languages that take homology into account. This includes the design of more efficient algorithms to check the conditions when large amounts of data are involved.

Acknowledgements. We thank Len Adleman for suggestions and providing the DNA sequences that were used in his 1994 experiment, and Richard Lipton for comments.

References

1. Adleman, L.: Molecular computation of solutions to combinatorial problems. *Science* **266**, 1021–1024 (1994)
2. Baum, E. B.: DNA sequences useful for computation. In: Landweber, L. F., Baum, E. (eds.) *Proceedings of DNA-based computers II*. Princeton. *AMS DIMACS Series* **44** 235–241 (1998)
3. Berstel, J., Perrin, D.: *Theory of codes*. Orlando: Academic Press 1985
4. Deaton, R., Garzon, M., Murphy, R., Franceschetti, D. R., Stevens, S. E.: Genetic search of reliable encodings for DNA based computation. In: *Proceedings of First Conference on Genetic Programming*, pp. 9–15. Stanford, Stanford University 1996
5. Deaton, R., Murphy, R., Garzon, M., Franceschetti, D. R., Stevens, S. E.: Good encodings for DNA-based solutions to combinatorial problems. In: Landweber, L. F., Baum, E. (eds.) *Proceedings of DNA-Based Computers II*, Princeton. *AMS DIMACS Series* **44** 247–258 (1998)
6. Deaton, R., Murphy, R. E., Rose, J. A., Garzon, M., Franceschetti, D. R:, Stevens, S. E., Jr.: A DNA based implementation of an evolutionary search for good encodings for DNA computation. In: *Proceedings IEEE Conference on Evolutionary Computation, ICEC-97*, 267–271 (1997)
7. Feldkamp, U., Saghafi, S., Rauhe, H.: DNASequenceGenerator – A program for the construction of DNA sequences. In: Jonoska, N., Seeman, N. C. (eds.), *Proceedings of DNA-Based Computers VII*, pp. 179–189. Tampa, Florida 2001
8. Frutos, A. G., Liu, Q., Thiel, A. J., Sanner, A. M. W., Condon, A. E., Smith, L. M., Corn, R. M.: Demonstration of a word design strategy for DNA computing on surfaces. *Nucleic Acids Research* **25** (23), 4748–4757 (1997)
9. Garzon, M., Deaton, R., Nino, L. F., Stevens, S. E., Jr., Wittner, M.: Genome encoding for DNA computing. In: *Proceedings of 3rd Genetic Programming Conference*, Madison, WI, pp. 684–690 (1998)
10. Garzon, M., Neathery, P., Deaton, R., Murphy, R. C., Franceschetti, D. R., Stevens, S. E., Jr.: A new metric for DNA computing. In: Koza, J. R., Deb, K., Dorigo, M., Vogel, D. B., Garzon, M., Iba, H., Riolo, R. L. (eds.) *Proceedings of 2nd Annual Genetic Programming Conference*, pp. 472–478. Stanford, Stanford University 1997
11. Garzon, M., Oehmen, C.: Biomolecular computation in virtual test tubes. In: Jonoska, N., Seeman, N. C. (eds.), *Proceedings of DNA-Based Computers VII*, pp. 75–83. Tampa, Florida 2001
12. Hartemink, A. J., Gifford, D. K., Khodor, J.: Automatic constraint-based nucleotide sequence selection for DNA computations. In: Kari, L., Rubin, H., Wood, D. H. (eds.) *Proceedings of DNA-Based Computers IV, Philadelphia. Biosystems* **52** (1–3), 227–235 (1999)
13. Head, T.: Formal language theory and DNA: an analysis of the generative capacity of specific recombinant behaviors. *Bull. Math. Biol.* **49**, 737–759 (1987)
14. Hussini, S., Kari, L., Konstantinidis, S.: Coding properties of DNA languages. In: Jonoska, N., Seeman, N. C. (eds.) *Proceedings of DNA-Based Computers VII*, pp. 107–118. Tampa, Florida 2001
15. Jonoska, N., Seeman, N. C. (eds.): *Proceedings of DNA-Based Computers VII*, Tampa, Florida 2001
16. Jürgensen, H., Konstantinidis, S.: Codes. In: Rozenberg, G., Salomaa, A. (eds.) *Handbook of Formal Languages*, Vol. 1, pp. 511–600. Berlin, Springer 1997
17. Kari, L.: DNA computing: arrival of biological mathematics. *The Mathematical Intelligencer* **19** (2), 9–22 (1997)

18. Kari, L., Kitto, R., Thierrin, G.: Codes, involutions and DNA encoding. In: Brauer, W., Ehrig, H., Karhumaki, J., Salomaa, A. (eds.) *Lecture Notes in Computer Science* **2300**, pp. 376–393. Berlin Heidelberg New York, Springer 2002
19. Kari, L., Thierrin, G.: Contextual insertions/deletions and computability. *Information and Computation* **131**, 47–61 (1996)
20. Konstantinidis, S., O'Hearn, A.: Error-detecting properties of languages. *Theoretical Computer Science* **276**, 355–375 (2003)
21. Manber, U.: *Introduction to algorithms: A creative approach.* Addison-Wesley Publishing Company 1989
22. Marathe, A., Condon, A., Corn, R.: On combinatorial DNA word design. In: Winfree, E., Gifford, D. (eds.) *Proceeedings of DNA-Based Computers V*, 75–89 (1999)
23. Paun, G., Rozenberg, G., Salomaa, A. (eds.) *DNA Computing: New Computing Paradigms*. Berlin, Heidelberg New York, Springer 1998
24. Reif, J. H., LaBean, T. H., Pirrung, M., Rana, V. S., Guo, B., Kingsford, C., Wickham, G. S.: Experimental construction of very large scale DNA databases with associative search capability. In: Jonoska, N., Seeman, N. C. (eds.): *Proceedings of DNA-Based Computers VII*, pp. 241–250. Tampa, Florida 2001
25. Rozenberg, G., Salomaa, A. (eds.) *Handbook of formal languages*. Berlin Heidelberg New York, Springer 1997
26. Shyr, H. J.: *Free Monoids and Languages*. Taichung, Taiwan: Hon Min Book Company 1991
27. Wood, D.: *Theory of computation*. New York: Harper and Row 1987
28. Yu, S.: *Regular languages*. In: Rozenberg, G., Salomaa, A. (eds.) *Handbook of formal languages*, Vol. 1, pp. 41–110. Berlin Heidelberg New York, Springer 1997