Fundamenta Informaticae 138 (2015) 179–192
DOI 10.3233/FI-2015-1205
IOS Press

A Formal Language Model of DNA Polymerase Enzymatic Activity

Srujan Kumar Enaganti, Lila Kari*, Steffen Kopecki
Department of Computer Science
The University of Western Ontario
London, ON N6A 5B7, Canada
senagant@uwo.ca;lila.kari@uwo.ca;steffen@csd.uwo.ca

Abstract. We propose and investigate a formal language operation inspired by the naturally occurring phenomenon of DNA primer extension by a DNA-template-directed DNA Polymerase enzyme. Given two DNA strings u and v, where the shorter string v (called *primer*) is Watson-Crick complementary and can thus bind to a substring of the longer string u (called *template*) the result of the primer extension is a DNA string that is complementary to a suffix of the template which starts at the binding position of the primer. The operation of DNA primer extension can be abstracted as a binary operation on two formal languages: a template language L_1 and a primer language L_2. We call this language operation L_1*-directed extension of* L_2 and study the closure properties of various language classes, including the classes in the Chomsky hierarchy, under directed extension. Furthermore, we answer the question under what conditions can a given language of target strings be generated from a given template language when the primer language is unknown. We use the canonic inverse of directed extension in order to obtain the optimal solution (the minimal primer language) to this question.

1. Introduction

Computational models inspired by nature abound in theoretical computer science. Several formal language operations that have their basis on naturally occurring biochemical reactions have been proposed and studied. The actions of various enzymes on DNA strands, most of which are widely used in the field

*Address for correspondence: Department of Computer Science, The University of Western Ontario, London, ON, Canada

[0]This research was supported by a Natural Science and Engineering Council of Canada (NSERC) Discovery Grant and a University of Western Ontario Grant to L.K.

of biotechnology, are of particular interest. In this paper we propose and investigate a formal language operation that models the action of DNA Polymerase enzyme, an enzyme that plays a major role in the replication of DNA strands.

Other bio-inspired operations in the literature include splicing, insertion and deletion, substitution, and hairpin extension. *Splicing* is a formal language operation originally proposed by Tom Head [10] to model the recombination of DNA strands under the action of restriction enzymes and ligase enzymes. Various types of splicing systems have been developed based on this phenomenon and their properties were studied in, e.g., [29] [9] [19] [11] [15]. *Insertion-deletion* operations are basic to DNA processing and RNA editing in molecular biology. Insertion-Deletion systems were defined as formal models of computation based on these operations and have been widely studied in the literature, see, e.g., [17] [31] [33] [34] [30] [18] [5]. Insertion-deletion systems that are context-free [27], that have one sided-context [28] [23], and that are graph controlled [6] were also proposed. P-systems with insertion-deletion rules have been extensively studied in [22] [24] [2] [1] [7] [8]. A type of *substitution* operation inspired by errors occurring in biologically encoded information was proposed in [16]. *Hairpin formation* is a naturally occurring phenomenon whereby a DNA strand that is partially self-complementary attaches to itself. Based on this phenomenon, the formal language operation called hairpin completion as well as its inverse operation called hairpin reduction have been defined and extensively studied in the literature [4] [26] [25] [21].

In this paper we define and investigate a formal language operation that models the action of the DNA Polymerase enzyme on DNA strands. Recall that a *DNA single-strand* consists of four different types of units called *nucleotides* or *bases* strung together by an oriented *backbone* like beads on a wire. The distinct ends of a DNA single strand are called the 5' end and the 3' end respectively. The bases are Adenine (A), Guanine (G), Cytosine (C) and Thymine (T), and A can chemically bind to an opposing T on another single strand, while C can similarly bind to G. Bases that can thus bind are called *Watson/Crick (W/C) complementary*, and two DNA single strands with opposite orientation and with *W/C* complementary bases at each position can bind to each other to form a *DNA double strand* in a process called *base-pairing*.

The activity of DNA Polymerase presupposes the existence of a DNA single strand called *template* (Figure 1 (a)), and of a second short DNA strand called *primer*, that is Watson-Crick complementary to the template (Figure 1 (b)). Given a supply of individual nucleotides, the DNA polymerase enzyme extends the primer, at one of its ends only, by adding invididual nucleotides complementary to the template nucleotides, one by one, until the end of the template is reached (Figure 1 (c)). The newly formed DNA strand is a strand that starts with the primer and is partially Watson-Crick complementary to the template (Figure 1 (d)). In molecular biology laboratories, an iterated version of this process is used to obtain an exponential replication of DNA strands, in a protocol called *Polymerase Chain Reaction*, or *PCR*.

In this paper we introduce a simplified formal language model of DNA Polymerase enzymatic activity, called *template-directed extension*, or simply *directed extension*. The paper is organized as follows. Section 2 contains definitions and notations, including the definition of directed extension. In Section 3, we give proofs for the closure properties of the various language classes under directed extension. In particular, we show that the directed extension between two languages in LOGSPACE can result in an undecidable language. In Section 4, we define an inverse of directed extension and study language equations involving this operation. In Section 5, we compare our operation with related string operations, and we discuss iterated versions of directed extension.

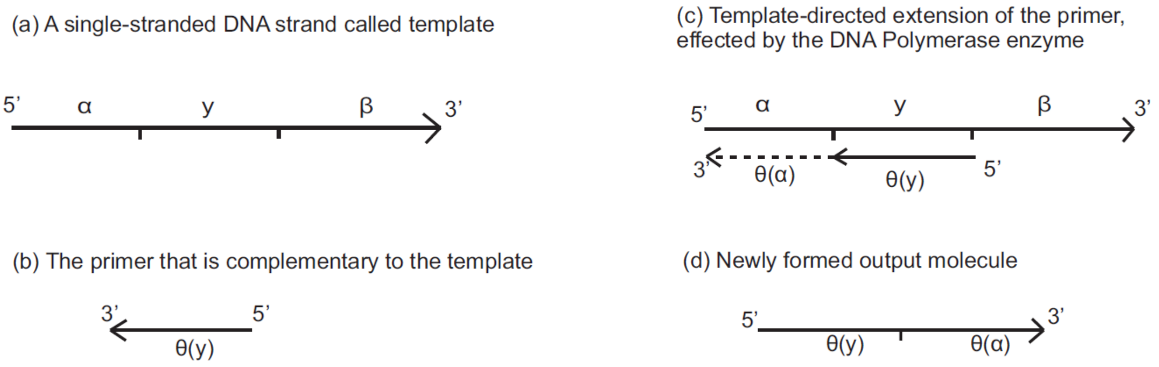
(a) A single-stranded DNA strand called template
5'
α
y
β
3'
(c) Template-directed extension of the primer, effected by the DNA Polymerase enzyme
5'
α
y
β
3'
3'
θ(α)
θ(y)
5'
(b) The primer that is complementary to the template
3'
5'
θ(y)
(d) Newly formed output molecule
5'
3'
θ(y)
θ(α)

Figure 1. Template directed extension of a primer, effected by DNA Polymerase enzyme. By $\theta(x)$ we denote the Watson-Crick complement of a DNA strand x.

2. Basic definitions and notations

An alphabet Σ is a finite non-empty set of symbols. Σ^* denotes the set of all words over Σ, including the empty word λ. Σ^+ is the set of all non-empty words over Σ. For words w, x, y, z such that $w = xyz$ we call the subwords x, y, and z *prefix*, *infix*, and *suffix* of z, respectively. The sets $Pref(w)$, $Inf(w)$, and $Suff(w)$ contain, respectively, all prefixes, infixes, and suffixes of w. This notation is extended to languages as follows: $Suff(L) = \bigcup_{w \in L} Suff(w)$. The complement of a language $L \subseteq \Sigma^*$ is $L^c = \Sigma^* \backslash L$. By FIN, REG, LIN, CF, CS, and RE we denote the families of finite, regular, linear (context-free), context-free, context-sensitive, and recursively enumerable languages, respectively.

An *involution* is a function $\theta : \Sigma^* \to \Sigma^*$ with the property that θ^2 is identity. θ is called an *antimorphism* if $\theta(uv) = \theta(v)\theta(u)$. Traditionally, the Watson-Crick complementarity of languages has been modelled as an antimorphic involution over the DNA alphabet $\Delta = \{A, C, G, T\}$, [12, 14]. Assuming the convention that a word x over this alphabet represents the DNA single strand x in the 5' to 3' direction, the activitity of DNA polymerase in Figure 1, given a template $\alpha y \beta$ and a primer y that occurs only once in $\alpha y \beta$, can be modelled as:

$$\alpha y \beta \ \bullet \ \theta(y) = \theta(y)\theta(\alpha) = \theta(\alpha y).$$

Assuming that all involved DNA strands are initially double-stranded, that is, whenever the strand x is available also its Watson-Crick complement $\theta(x)$ is available, we can further simplify this model and, given two words x, y over an alphabet Σ, we can define the *left x-directed extension of y* as

$$x \oplus' y = \{w \in \Sigma^* \mid \exists \alpha, \beta \in \Sigma^* : x = \alpha y \beta, w = \alpha y\},$$

and the *right x-directed extension of y* as

$$x \oplus y = \{w \in \Sigma^* \mid \exists \alpha, \beta \in \Sigma^* : x = \alpha y \beta, w = y \beta\},$$

From a mathematical point of view the left- and right-directed extensions are similar. For the remainder of this paper we will consider only the right-directed extension, which we will call simply directed extension.

Note also that, from a biological point of view, it does not make sense to consider an "empty primer" (a primer with length 0), but from a mathematical point of view this is well-defined and $y = \lambda$ is valid. We extend the definition of directed extension to languages in a natural way:

$$L_x \oplus L_y = \bigcup_{x \in L_x, y \in L_y} x \oplus y = \{w \in \Sigma^+ \mid \exists \alpha, \beta \in \Sigma^*, y \in L_y \colon \alpha y \beta \in L_x, w = y\beta\}.$$

3. Closure Properties

In this section we study closure properties of various language classes under directed extension. Throughout this section all languages are considered to be defined over a fixed alphabet Σ. The next lemma expresses the directed extension operation in terms of concatenation, intersection and suffix.

Lemma 3.1. If L_x and L_y are two languages over Σ, then $L_x \oplus L_y = \mathit{Suff}(L_x) \cap L_y\Sigma^*$.

Proof:

For the direct inclusion, consider $w \in L_x \oplus L_y$. This implies that $w = y\beta$ where $y \in L_y$ and $\alpha y \beta \in L_x$. Therefore, $w \in L_y\Sigma^*$ and $w \in \mathit{Suff}(L_x)$.

Conversely, let $w \in \mathit{Suff}(L_x) \cap L_y\Sigma^*$. Because $w \in \mathit{Suff}(L_x)$, there exists $\alpha \in \Sigma^*$ such that $\alpha w \in L_x$. Because $w \in L_y\Sigma^*$, there exists $y \in L_y$ and $\beta \in \Sigma^*$ such that $w = y\beta$. Thus, $w \in L_x \oplus L_y$. □

Corollary 3.2. Let $\mathcal{X}$ and $\mathcal{Y}$ be two language classes where $\mathcal{X}$ is closed under the suffix operator and $\mathcal{Y}$ is closed under concatenation with Σ^*.

i.) If $\mathcal{X}$ is closed under intersection with languages from $\mathcal{Y}$, then for all $L_x \in \mathcal{X}$ and $L_y \in \mathcal{Y}$ we have $L_x \oplus L_y \in \mathcal{X}$.

ii.) If $\mathcal{Y}$ is closed under intersection with languages from $\mathcal{X}$, then for all $L_x \in \mathcal{X}$ and $L_y \in \mathcal{Y}$ we have $L_x \oplus L_y \in \mathcal{Y}$.

In particular, REG and RE are closed under directed extension and, if $\mathcal{X}$ is LIN (CF) and $\mathcal{Y}$ is REG, then the result $L_x \oplus L_y$ is in LIN (CF).

Next, we show that directed extension can "simulate" intersection by utilizing markers at the beginning and end of words.

Lemma 3.3. Let L_1 and L_2 be languages over the alphabet Σ and let $\$ \notin \Sigma$ be a new symbol. Then,

$$\$L_1\$ \oplus \$L_2\$ = \$(L_1 \cap L_2)\$.$$

Proof:
For the direct inclusion, let $x \in L_1$ and $y \in L_2$. If the word $\$x\$$ has a factorization $\$x\$ = \alpha\$y\$\beta$, it is clear that $x = y$ and $\alpha = \beta = \lambda$ because \$ does not occur as letter in x. Therefore, if $w \in \$x\$ \oplus \$y\$$ for some $x \in L_1$ and $y \in L_y$, then $w \in \$(L_1 \cap L_2)\$$.

For the converse inclusion, let w be any string in $(L_1 \cap L_2)$. This implies that $\$w\$ \in \$L_1\$$ and $\$w\$ \in \$L_2\$$. Thus $\$w\$ \in \$L_1\$ \oplus \$L_2\$$. □

Lemma 3.3 allows us to classify the result of directed extension between two (linear) context-free languages.

Theorem 3.4. Let L_x be a context-free language and L_y be a context-free (or context-sensitive) language. The language $L_x \oplus L_y$ is context-sensitive, but not necessarily context-free.

Proof:
Consider the two (linear) context-free languages

$$L_x = \{\$a^m b^n c^n\$ \mid m \geq 1, n \geq 1\}, \qquad L_y = \{\$a^n b^n c^m\$ \mid m \geq 1, n \geq 1\}.$$

By Lemma 3.3, the L_x-directed extension of L_y yields the context-sensitive but not context-free language

$$L_x \oplus L_y = \{\$a^n b^n c^n\$ \mid n \geq 1\}.$$

In order to show that $L_x \oplus L_y$ is context-sensitive for $L_x \in$ CF and $L_y \in$ CS, we use Lemma 3.1 and note that the suffix operator applied to a context-free language gives a context-free language and that the class of context-sensitive languages is closed under intersection. □

Let LOG $=$ DSPACE(log) be the language class which contains all languages that can be accepted by a deterministic Turing Machine using at most $\mathcal{O}(\log n)$ space on an input of length n. For a language $L_x \in LOG$ we will show that the L_x-directed extension of a singleton language can produce an undecidable language. In order to do so, we utilize the undecidable Post Correspondence Problem (PCP) in the following formulation: Determine, for an arbitrary set $(x_1, y_1), (x_2, y_2), \cdots, (x_k, y_k)$ of pairs of corresponding non-null strings over the alphabet $\{a, b\}$, whether or not there exists a solution n, $i_1, i_2, i_3, \cdots, i_n$ such that $x_{i_1} x_{i_2} x_{i_3} \cdots x_{i_n} = y_{i_1} y_{i_2} y_{i_3} \cdots y_{i_n}$, $n \geq 1, i_j \in \{1, 2, \cdots, k\}$.

Theorem 3.5. There exists a language L_1 in LOG and a singleton language L_2 such that $L_1 \oplus L_2$ is not decidable.

Proof:
Let L_1 be a language over $\Sigma \cup \{\$\}$ consisting of all strings of the form $\alpha\$\beta$ where \$ does not appear within α or β. Here β is the encoding of an instance of the PCP and α is the encoding of a solution of this instance. We let L_2 be the singleton language $\{\$\}$. The resulting language $L_1 \oplus L_2$ contains all strings of the form $\$\beta$ such that $\alpha\$\beta \in L_1$; therefore, $\$\beta \in L_1 \oplus L_2$ if and only if β is the encoding of an instance of PCP which has a solution. Formally,

$$\begin{aligned} L_1 &= \{\alpha\$\beta \mid \beta \text{ is a PCP instance and } \alpha \text{ is a solution to } \beta\}, \\ L_2 &= \{\$\}, \\ L_1 \oplus L_2 &= \{\$\beta \mid \beta \text{ is a PCP instance that has a solution}\}. \end{aligned}$$

Because PCP is undecidable, it will follow that the language $L_1 \oplus L_2$ is undecidable as well. Let us show next how to encode α and β in a word $\alpha\$\beta \in L_x$ and how to decide L_x using logarithmic space.

Let $x_1, x_2, ..., x_k$ and $y_1, y_2, ..., y_k$ be an instance of PCP and let $i_1, i_2, ...i_n$ be a solution to this instance. We encode each integer i_j using a binary encoding, symbolized as $|i_j|$, which is of length $\lceil \log_2 k \rceil$ or less. Let $\alpha \$ \beta$ be encoded as

$$|i_1|M|i_2|M|i_3|M...|i_n|M\$Mx_1Mx_2Mx_3...Mx_kMCMy_1My_2My_3...My_kM$$

where M and C are separating symbols.

In order to decide if an arbitrary string w is in L_1, the first step is to verify that it is of the format described above and the second step is to verify that the integer sequence α is a solution of β. In order to decide L_1 we have to verify whether or not $x_{i_1}x_{i_2}x_{i_3}\cdots x_{i_n}$ and $y_{i_1}y_{i_2}y_{i_3}\cdots y_{i_n}$ are equal. We can easily see that the first step can be done in logarithmic space and that the second step can (at least) be decided. Thus, the language L_1 is decidable.

Now, we give a high-level construction of a Turing Machine which uses logarithmic working space with respect to the length of the input and decides whether α is a solution to β or not. Instead of generating both strings completely and then comparing them, we generate and compare both strings **letter by letter**. In order to do so, we only need to store pointers to the input tape on the work tape which can be implemented using only logarithmic space. A more detailed description of this Turing Machine follows.

We may assume the symbol S is written to the left of input and refer to it as the start symbol. The strings $x_{i_1}x_{i_2}\cdots x_{i_n}$ and $y_{i_1}y_{i_2}\cdots y_{i_n}$ are referred to as x and y respectively.

When we say address, we refer to the address on the input tape with respect to S, i.e. the number of symbols we have to move to the right starting from S on the input tape. The input tape looks as follows:

$$S|i_1|M|i_2|M|i_3|M...|i_n|M\$Mx_1Mx_2Mx_3...Mx_kMCMy_1My_2My_3...My_kM$$

The computation of the Turing Machine is described by Algorithm 1. We use the following variables in the pseudo-code:

x_{addr}	–	The address of current symbol of x that is being looked into
y_{addr}	–	The address of current symbol of y that is being looked into
x_{soln}	–	The value of the current index (i.e. i_j) of x
y_{soln}	–	The value of the current index (i.e. i_j) of y
$x_{solnAddr}$	–	Contains the address of x_{soln}
$y_{solnAddr}$	–	Contains the address of y_{soln}
$AddrValue$	–	A buffer storing the address to be calculated/used

Moreover, we use following simple functions:

- $Addr(s)$, where s is one of the symbols $S, \$, C$, returns the unique address of the symbol s on the input tape,

- $ValueAt(addr)$, where $addr$ is an address, returns the symbol on the input tape at address $addr$,

- $ReadIndex(index, addr)$, where $index$ is a variable on the work tape and $addr$ is an address, copies the binary representation of an index i_j which begins at address $addr$ into $index$; it also increments the address $addr$ such that it points to the first bit of $|i_{j+1}|$ if $j < n$ and to $Addr(\$)$ if $j = n$.

Algorithm 1

```
$x_{addr} := Addr(\$)$;
$y_{addr} := Addr(C)$;
$x_{solnAddr} = y_{solnAddr} := Addr(S)$;
repeat
    $x_{addr} := x_{addr} + 1$;
    $y_{addr} := y_{addr} + 1$;
    if $ValueAt(x_{addr}) = M$ then
        if $ValueAt(x_{solnAddr}) = \$$ then
            $x_{addr} := Addr(\$)$;
        else
            $ReadIndex(x_{soln}, x_{solnAddr})$;
            $AddrValue := Addr(\$)$;
            while $x_{soln} > 0$ do
                if $ValueAt(AddrValue) = M$ then
                    $x_{soln} := x_{soln} - 1$;
                end if
                $AddrValue := AddrValue + 1$;
            end while
            $x_{addr} := AddrValue$;
        end if
    end if
    $a := ValueAt(x_{addr})$;
    if $ValueAt(y_{addr}) = M$ then
        if $ValueAt(y_{solnAddr}) = \$$ then
            $y_{addr} := Addr(\$)$;
        else
            $ReadIndex(y_{soln}, y_{solnAddr})$;
            $AddrValue := Addr(C)$;
            while $y_{soln} > 0$ do
                if $ValueAt(AddrValue) = M$ then
                    $y_{soln} := y_{soln} - 1$;
                end if
                $AddrValue := AddrValue + 1$;
            end while
            $y_{addr} := AddrValue$;
        end if
    end if
    $b := ValueAt(y_{addr})$;
until $(a \neq b) OR (a = b = \$)$
if $a \neq b$ then
    return no;
else
    return yes;
end if
```

Then **Algorithm 1** will always halt with either a **yes** or a **no** because there is only a finite number of indexes encoded in α and hence in the case of not-finding a mismatch(including the mismatch due to one string finishing earlier than the other), the condition $a = b = \$$ will be satisfied giving a **yes** answer. The variables used in this algorithm, x_{addr}, y_{addr}, x_{soln}, y_{soln}, $x_{solnAddr}$, $y_{solnAddr}$ and $AddrValue$. All of them except for x_{soln} and y_{soln} are pointers to locations on read-tape and, hence, require only logarithmic space with respect to the input. We already know that x_{soln} and y_{soln} are within $\lceil \log_2 k \rceil$ space and hence within logarithmic space with respect to the input. Since all the variables can be stored in space logarithmic with respect to the input, we conclude that L_1 can be decided in logarithmic space. We conclude that if L_1 is in LOG and L_2 is a singleton language, then $L_1 \oplus L_2$ can be an undecidable language. □

Theorem 3.5 can be extended to any time or space complexity class which contains LOG as well as to decidable languages. In particular, CS is not closed under directed extension of singleton languages.

Corollary 3.6. The family of context-sensitive languages is not closed under directed extension. More precisely, for $L_x \in$ CS the L_x-directed extension of a singleton language may not be decidable.

Corollary 3.7. The language classes NTIME, DTIME, NSPACE and DSPACE (all of which include LOG) are not closed under directed extension. More precisely, if $L_x \in$ NTIME, DTIME, NSPACE, DSPACE then the L_x-directed extension of a singleton language may not be decidable.

Table 1. Summary of closure properties: each entry shows which language class $L_x \oplus L_y$ belongs to if L_x is from the corresponding language class in the left column and L_y is from the corresponding language class in the top row.

$L_x \backslash L_y$	FIN or REG	CF	CS	RE
REG	REG (Cor. 3.2)	CF (Cor. 3.2)	CS (Cor. 3.2)	RE (Cor. 3.2)
CF	CF (Cor. 3.2)	CS (Thm. 3.4)		RE (Cor. 3.2)
CS	RE (Cor. 3.2 and Cor. 3.6)			
RE	RE (Cor. 3.2)			

In Table 1 we summarize the results from this section. For two language classes $\mathcal{X}$ and $\mathcal{Y}$, it shows the language class $\mathcal{Z}$ from the Chomsky hierarchy such that for all $L_x \in \mathcal{X}$ and $L_y \in \mathcal{Y}$ we have $L_x \oplus L_y \in \mathcal{Z}$. Note that if we consider two language classes $\mathcal{X}$, $\mathcal{Y}$ which both contain the free monoid Σ^* for any alphabet Σ, we will require that $\$L\$ = \$L\$ \cap \$\Sigma^*\$ \in \mathcal{Z}$ for all languages $L \in \mathcal{X}$ or $L \in \mathcal{Y}$ which are defined over Σ, due to Lemma 3.3. If we restrict ourselves to classes in the Chomsky hierarchy (or standard space/time complexity classes), this statement can be strengthend as $\mathcal{X} \cup \mathcal{Y} \subseteq \mathcal{Z}$. This shows that all entries in Table 2 can also be considered "lower bounds" for the language class $\mathcal{Z}$.

Finally, let us also note that if L_x is a finite language, then $L_x \oplus L_y$ is finite for any L_y, even though it is not necessarily effectively finite if L_y is undecidable.

4. Equations and inverse operation

In this section we investigate the following problem: Given two languages L_x, L_0 over Σ^*, does there exist a language Y over Σ^* such that $L_x \oplus Y = L_0$? Furthermore, we show how to effectively construct maximal and minimal solutions, with respect to the inclusion relation. Throughout this section, we consider the languages L_x and L_0 to be constants. For the equation $L_x \oplus Y = L_0$ we call a language L_y a *solution* if it satisfies $L_x \oplus L_y = L_0$.

We can use the canonical right-inverse of the directed extension in order to decide the existence of a solution as well as to find the maximal solution. The canonical right-inverse of an arbitrary binary language operation $''+''$ is the binary language operation $''-''$ defined as

$$x - w = \{y \in \Sigma^* \mid w \in x + y\}.$$

It was proved that, if there exists a solution L_y of the equation $L_x + Y = L_0$, then $L_{max} = (L_x - L_0^c)^c$ is also a solution, and every other solution L'_y of this equation is contained in L_{max} [13]. In other words, for languages L_x, L_y, and L_0

$$L_x + L_y = L_0 \iff L_y \subseteq (L_x - L_0^c)^c.$$

It is easy to see that the right-inverse of directed extension is

$$\begin{aligned} x \ominus w &= \{y \in \Sigma^* \mid w \in x \oplus y\} \\ &= \begin{cases} \mathit{Pref}(w) & \text{if } x = \alpha w \\ \emptyset & \text{otherwise.} \end{cases} \end{aligned}$$

Therefore, we obtain that $L_{max} = (L_x \ominus L_0^c)^c$ is the maximal solution (with respect to inclusion) of $(L_x \oplus Y = L_0)$ if and only if $L_x \oplus Y = L_0$ has at least one solution.

This already implies that we can decide whether or not the equation $L_x \oplus Y = L_0$ has a solution L_y. Yet, we want to present a "more direct" approach to test solvability of this equation: we will show that the equation has a solution if and only if $L_x \oplus L_0 = L_0$.

Theorem 4.1. The equation $L_x \oplus Y = L_0$ has a solution L_y if and only if L_0 is a solution as well.

Proof:
Trivially, if $L_x \oplus L_0 = L_0$, then there exists an L_y such that $L_x \oplus L_y = L_0$.

Conversely, we need to prove that if $L_x \oplus L_y = L_0$, then $L_x \oplus L_0 = L_x \oplus L_y$. Let us consider a string $w \in L_x \oplus L_y$. This implies that w is a suffix of a word $x \in L_x$ and, therefore, $w \in x \oplus w \subseteq L_x \oplus L_0$. This proves that $L_x \oplus L_0 \supseteq L_x \oplus L_y$.

Now, take any $w' \in L_x \oplus w$ for some $w \in L_0 = L_x \oplus L_y$. Hence, w' is a suffix of some word $x \in L_x$ and, furthermore, there exists a word $y \in L_y$ which is a prefix of w which in turn is a prefix of w' by Lemma 3.1. Clearly, this implies that $w' \in x \oplus y \subseteq L_x \oplus L_y$. We conclude $L_x \oplus L_0 = L_x \oplus L_y$. □

Next, we investigate solutions which are *minimal* with respect to inclusion; that is, a solution L_y of the equation $L_x \oplus Y = L_0$ is minimal if for all words $y \in L_y$ the language $L_y \setminus \{y\}$ is not a solution: $L_x \oplus (L_y \setminus \{y\}) \neq L_0$. We present a general method to find a minimal solution if we already know one solution.

Theorem 4.2. If $L_x \oplus Y = L_0$ has the solution L_y, then $L_{min} = (L_y \backslash L_y \Sigma^+) \cap \mathit{Inf}(L_x)$ is a minimal solution.

Proof:
First, let us show that L_{min} is indeed a solution. Because $L_{min} \subseteq L_y$, we have $L_x \oplus L_{min} \subseteq L_x \oplus L_y = L_0$. Vice versa, for every $w \in L_0$ there exists $x \in L_x$ and $y \in L_y$ such that $w \in x \oplus y$. Let y' be the shortest prefix of y such that $y' \in L_y$. Because y' does not have a shorter prefix in L_y and because y' is an infix of x, we obtain that $y' \in L_{min}$. Now, since y' is also a prefix of w, we obtain that $w \in x \oplus y' \subseteq L_x \oplus L_{min}$.

For the sake of obtaining a contradiction, let us assume that L_{min} is not a minimal solution. This implies that either (a) there is $y \in L_{min}$ such that $L_x \oplus y = \emptyset$ or (b) there are two distinct strings $y_1, y_2 \in L_{min}$ such that a word w in $L_x \oplus y_1 \cap L_x \oplus y_2$ exists. Case (a) does not hold because it would imply that y is not an infix of any word in L_x. Case (b) implies that y_1 and y_2 are both prefixes of the word w which means that we may assume that y_1 is a prefix of y_2 without loss of generality. Since both words have to belong to L_y and $y_2 \in y_1\Sigma^*$, we conclude that $y_2 \notin L_{min}$ — a contradiction. □

From the two results in this section, Theorems 4.1 and 4.2, we infer that if the equation $L_x \oplus Y = L_0$ has a solution, then $L_{0,min} = (L_0 \backslash L_0 \Sigma^+) \cap \mathit{Inf}(L_x)$ is a minimal solution.

5. Discussion and conclusions

We now compare the directed extension operation with two other formal language operations that are biologically motivated and extend strings: the PA-matching operation and the superposition operation. The PA-matching operation is a binary operation proposed by Kobayashi et al [20] and inspired by the PA-Match operation that was part of Parallel Associate Memory(PAM) model proposed by Reif [32].

The PA-matching operation is meant to be implemented by some recombinant DNA processes and is defined as follows. Given two words $x \in V_1^+$ and $y \in V_2^+$, the result of the PA-matching between x and y is defined as:

$$PAm(x, y) = \{uv | x = uw, y = wv, \text{ for some } w \in (V_1 \cap V_2)^+, \text{ and } u \in V_1^*, v \in V_2^*\}$$

Note that PA-matching results in the extension of a word x by a suffix of y, if x has a suffix which is the same with a prefix of y. The main difference between this operation and directed extension is that here the common suffix/prefix that guides the extension is deleted from the result, while in the case of directed extension no deletion takes place.

The superposition operation is a binary operation proposed by Bottoni et al in [3] and can be implemented by the use of the DNA Polymerase enzyme. The result of the superposition operation between words $x \in V_1^+$ and $y \in V_2^+$, denoted by $x \diamond y$, consists of the set of all words $z \in (V_1 \cup \bar{V_2})^+$ defined as follows ($\bar{y}$ denotes the complement of y, that is, the image of y through a *morphic* involution):

1. If there exist $u \in V_1^*, w \in V_1^+, v \in V_2^*$ such that $x = uw, y = \bar{w}v$, then $z = uw\bar{v}$.
2. If there exist $u, v \in V_1^*$ such that $x = u\bar{y}v$, then $z = u\bar{y}v$.
3. If there exist $u \in V_2^*, w \in V_1^*$ such that $x = wv, y = u\bar{w}$, then $z = \bar{u}wv$.
4. If there exist $u, v \in V_2^*$ such that $y = u\bar{x}v$, then $z = \bar{u}x\bar{v}$.

The superposition operation also extends words but, in the case of superposition the extension can be bidirectional, while in the case of directed extension the extension is always uni-directional. This and other differences lead to the two operations being different, as illustrated by the difference in the closure properties of the two operations.

Table 2 summarizes the closure properties of the operations of directed extension, PA-matching and superposition.

Table 2. Closure properties under the directed extension operation, $\oplus$, compared to the PA-matching and superposition operations.

Class of L_x and L_y	$\oplus$	PAm	$\diamond$
Regular	Closed	Closed	Closed
Context Free	Not Closed	Not Closed	Not Closed
Context Sensitive	Not Closed	Not Closed	Closed
Recursively Enumerable	Closed	Closed	Closed

We end this paper by several remarks on iterated directed extension. When investigating language operations, it is common to investigate an iterated version of the operation as well. In particular, when studying biologically motivated operations as is the case here, the iterated version is sometimes the operation that better reflects the biological phenomenon in question (DNA replication) or experimental lab protocols (Polymerase Chain Reaction). Let us present here three natural versions of the iterated directed extension. We define

1. the *iterated self-directed extension of* L as $\mu^*(L) = \lim_{n\to\infty} \mu^n(L)$ where $\mu(L) = L \cup (L \oplus L)$,
2. the *L-iteration-directed extension of* L_y as $\nu_{L_y}^*(L) = \lim_{n\to\infty} \nu_{L_y}^n(L)$ where $\nu_{L_y}(L) = L \cup (L \oplus L_y)$, and
3. the *iterated L_x-directed extension of* L as $\xi_{L_x}^*(L) = \lim_{n\to\infty} \xi_{L_x}^n(L)$ where $\xi_{L_x}(L) = L \cup (L_x \oplus L)$.

Here, we use the notation that for any domain D and function $h\colon D \to D$ we have $h^0(L) = L$ and $h^i(L) = h(h^{i-1}(L))$ for $i \geq 1$.

Let us show that in all three cases we have $h^*(L) = h(L)$ for $h \in \{\mu, \nu_{L_y}, \xi_{L_x}\}$ which means that the results that we obtained in this paper can easily be extended to the iterated versions. Indeed, the only difference is that we add the term $h^0(L) = L$ to the directed extension.

For case 1.) consider a word $w \in \mu^2(L)$, that is (a) $w \in \mu(L)$ or (b) $w = x \oplus y$ for $x, y = \mu(L) = L \cup (L \oplus L)$. If (b) holds, we obtain from Lemma 3.1 that there exists $x' \in L$ such that x is a suffix of x'

and $y' \in L$ such that y' is a prefix of y (note that we do allow $x = x'$ or $y = y'$). Clearly, we also have $w \in x' \oplus y'$ and may conclude that $w \in L \oplus L \subseteq \mu(L)$. This implies that $\mu^2(L) \subseteq \mu(L)$ and, due to the inductive definition of μ^i we have $\mu^i(L) = \mu(L)$ for any $i \geq 1$. We conclude that $\mu^*(L) = \mu(L)$. The result follows by analogous arguments for the cases 2.) and 3.).

References

[1] A. Alhazov, A. Krassovitskiy, Y. Rogozhin, and S. Verlan. P systems with insertion and deletion exo-operations. *Fundamenta Informaticae*, 110(1-4):13–28, 2011.

[2] A. Alhazov, A. Krassovitskiy, Y. Rogozhin, and S. Verlan. P systems with minimal insertion and deletion. *Theoretical Computuer Science*, 412(1-2):136–144, 2011.

[3] P. Bottoni, A. Labella, V. Manca, and V. Mitrana. Superposition based on Watson-Crick-like complementarity. *Theory of Computing Systems*, 39(4):503–524, 2006.

[4] D. Cheptea, C. Martín-Vide, and V. Mitrana. A new operation on words suggested by DNA biochemistry: Hairpin completion. In *Transgressive Computing,* TC 2006, pages 216–228, 2006.

[5] M. Daley, L. Kari, G. Gloor, and R. Siromoney. Circular contextual insertions/deletions with applications to biomolecular computation. In Proceedings of 6th International Symposium on *String Processing and Information Retrieval,* SPIRE 1999, pages 47–54, 1999.

[6] R. Freund, M. Kogler, Y. Rogozhin, and S. Verlan. Graph-controlled insertion-deletion systems. In I. McQuillan and G. Pighizzini, editors, Proceedings of 12th International Workshop on *Descriptional Complexity of Formal Systems,* DCFS 2010, volume 31 of *Electronic Proceedings in Theoretical Computer Science*, pages 88–98, 2010.

[7] R. Freund, Y. Rogozhin, and S. Verlan. P systems with minimal left and right insertion and deletion. In J. Durand-Lose and N. Jonoska, editors, Proceedings of 11th International Conference on *Unconventional Computation and Natural Computation,* UCNC 2012, volume 7445 of *LNCS*, pages 82–93, 2012.

[8] R. Freund, Y. Rogozhin, and S. Verlan. Generating and accepting P systems with minimal left and right insertion and deletion. *Natural Computing*, 13(2):257–268, 2014.

[9] R. W. Gatterdam. Splicing systems and regularity. *International Journal of Computer Mathematics*, 31(1-2):63–67, 1989.

[10] T. Head. Formal language theory and DNA: An analysis of the generative capacity of specific recombinant behaviors. *Bulletin of Mathematical Biology*, 49(6):737–759, 1987.

[11] T. Head, D. Pixton, and E. Goode. Splicing systems: Regularity and below. In M. Hagiya and A. Ohuchi, editors, Revised Papers from the 8th International Workshop on *DNA Based Computers: DNA Computing,* DNA 8, volume 2568 of *LNCS*, pages 262–268, 2003.

[12] S. Hussini, L. Kari, and S. Konstantinidis. Coding properties of DNA languages. *Theoretical Computer Science*, 290(3):1557–1579, 2003.

[13] L. Kari. On language equations with invertible operations. *Theoretical Computer Science*, 132(1-2):129–150, 1994.

[14] L. Kari, R. Kitto, and G. Thierrin. Codes, involutions, and DNA encodings. In W. Brauer, H. Ehrig, J. Karhumaki, and A. Salomaa, editors, *Formal and Natural Computing*, volume 2300 of *LNCS*, pages 376–393, 2002.

[15] L. Kari and S. Kopecki. Deciding whether a regular language is generated by a splicing system. In D. Stefanovic and A. Turberfield, editors, *DNA Computing and Molecular Programming*, volume 7433 of *LNCS*, pages 98–109, 2012.

[16] L. Kari and E. Losseva. Block substitutions and their properties. *Fundamenta Informaticae*, 73(1-2):165–178, 2006.

[17] L. Kari, G. Păun, G. Thierrin, and S. Yu. At the crossroads of DNA computing and formal languages: Characterizing recursively enumerable languages using insertion-deletion systems. In *DNA Based Computers III (DNA3)*, volume 48 of *DIMACS: Series in Discrete Mathematics and Theoretical Computer Science*, pages 329–346, 1999.

[18] L. Kari and P. Sosík. On the weight of universal insertion grammars. *Theoretical Computer Science*, 396(1-3):264–270, 2008.

[19] S. Kim. An algorithm for identifying spliced languages. In D. Lee and S.-H. Teng, editors, Proceedings of 11th Annual *International Symposium on Algorithms and Computation,* ISAAC 2000, volume 1276 of *LNCS*, pages 403–411, 1997.

[20] S. Kobayashi, V. Mitrana, G. Păun, and G. Rozenberg. Formal properties of PA-matching. *Theoretical Computer Science*, 262(1-2):117–131, 2001.

[21] S. Kopecki. On iterated hairpin completion. *Theoretical Computer Science*, 412(29):3629–3638, 2011.

[22] A. Krassovitskiy, Y. Rogozhin, and S. Verlan. Computational power of P systems with small size insertion and deletion rules. In T. Neary, D. Woods, A. K. Seda, and N. Murphy, editors, Proceedings of International Workshop on *The Complexity of Simple Programs,* CSP 2008, volume 1 of *Electronic Proceedings in Theoretical Computer Science*, pages 108–117, 2008.

[23] A. Krassovitskiy, Y. Rogozhin, and S. Verlan. Further results on insertion-deletion systems with one-sided contexts. In C. Martín-Vide, F. Otto, and H. Fernau, editors, Proceedings of 2nd International Conference on *Language and Automata Theory and Applications,* LATA 2008, volume 5196 of *LNCS*, pages 333–344, 2008.

[24] A. Krassovitskiy, Y. Rogozhin, and S. Verlan. Computational power of insertion-deletion (P) systems with rules of size two. *Natural Computing*, 10(2):835–852, 2011.

[25] F. Manea, C. Martín-Vide, and V. Mitrana. On some algorithmic problems regarding the hairpin completion. *Discrete Applied Mathematics*, 157(9):2143–2152, 2009.

[26] F. Manea and V. Mitrana. Hairpin completion versus hairpin reduction. In S. B. Cooper, B. Löwe, and A. Sorbi, editors, Proceedings of 3rd Conference on *Computability in Europe,* CiE 2007, volume 4497 of *LNCS*, pages 532–541, 2007.

[27] M. Margenstern, G. Păun, Y. Rogozhin, and S. Verlan. Context-free insertion-deletion systems. *Theoretical Computer Science*, 330(2):339–348, 2005.

[28] A. Matveevici, Y. Rogozhin, and S. Verlan. Insertion-deletion systems with one-sided contexts. In J. Durand-Lose and M. Margenstern, editors, Proceedings of 5th International Conference on *Machines, Computations, and Universality,* MCU 2007, volume 4664 of *LNCS*, pages 205–217, 2007.

[29] D. Pixton. Regularity of splicing languages. *Discrete Applied Mathematics*, 69(1-2):101–124, 1996.

[30] G. Păun, M. J. Pèrez-Jimènez, and T. Yokomori. Representations and characterizations of languages in Chomsky hierarchy by means of insertion-deletion systems. *International Journal of Foundations of Computer Science*, 19(4):859–871, 2008.

[31] G. Păun, G. Rozenberg, and A. Salomaa. *DNA Computing: New Computing Paradigms(Texts in Theoretical Computer Science. An EATCS Series).* Springer-Verlag New York, Inc., 2006.

[32] J. H. Reif. Parallel molecular computation. In Proceedings of the 7th Annual ACM *Symposium on Parallel Algorithms and Architectures,* SPAA 1995, pages 213–223, 1995.

[33] A. Takahara and T. Yokomori. On the computational power of insertion-deletion systems. *Natural Computing*, 2(4):321–336, 2003.

[34] M. Yong, J. Xiao-Gang, S. Xian-Chuang, and P. Bo. Minimizing of the only-insertion insdel systems. *Journal of Zhejiang University Science A*, 6(10):1021–1025, 2005.