November 2015

Formal models of the extension activity of DNA polymerase enzymes

Srujan Kumar Enaganti
The University of Western Ontario

Supervisor
Professor Lila Kari
The University of Western Ontario

Graduate Program in Computer Science

A thesis submitted in partial fulfillment of the requirements for the degree in Doctor of Philosophy

FORMAL MODELS OF THE EXTENSION ACTIVITY OF DNA POLYMERASE ENZYMES

(Thesis format: Integrated Article)

by

Srujan Kumar Enaganti

Graduate Program in Computer Science

A thesis submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

The School of Graduate and Postdoctoral Studies

The University of Western Ontario

London, Ontario, Canada

Abstract

The study of formal language operations inspired by enzymatic actions on DNA is part of ongoing efforts to provide a formal framework and rigorous treatment of DNA-based information and DNA-based computation. Other studies along these lines include theoretical explorations of splicing systems, insertion-deletion systems, substitution, hairpin extension, hairpin reduction, superposition, overlapping concatenation, conditional concatenation, contextual intra- and intermolecular recombinations, as well as template-guided recombination.

First, a formal language operation is proposed and investigated, 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 the *primer*) is Watson-Crick complementary and can thus bind to a substring of the longer string u (called the *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. This language operation is called L_1*-directed extension of* L_2 and the closure properties of various language classes, including the classes in the Chomsky hierarchy, are studied under directed extension. Furthermore, the question of finding necessary and sufficient conditions for a given language of target strings to be generated from a given template language when the primer language is unknown is answered. The canonic inverse of directed extension is used in order to obtain the optimal solution (the minimal primer language) to this question.

The second research project investigates properties of the binary string and language operation *overlap assembly* as defined by Csuhaj-Varju, Petre and Vaszil as a formal model of the linear self-assembly of DNA strands: The overlap assembly of two strings, xy and yz, which share an “overlap” y, results in the string xyz. In this context, we investigate overlap assembly and its properties: closure properties of various language families under this operation, and related decision problems. A theoretical analysis of the possible use of iterated overlap assembly to generate combinatorial DNA libraries is also given.

The third research project continues the exploration of the properties of the overlap assembly operation by investigating closure properties of various language classes under iterated overlap assembly, and the decidability of the completeness of a language. The problem of deciding whether a given string is terminal with respect to a language, and the problem of deciding if a given language can be generated by an overlap assembly operation of two other given languages are also investigated.

Keywords: Formal language models, bio-operations, DNA computing, combinatorial library, decidability

Co-Authorship Statement

This thesis essentially consists of three research articles published in journals, and/or under review. All of them are co-authored by the author of the thesis, Srujan Kumar Enaganti (S.K.E.), the author's supervisor Prof. Lila Kari (L.K.) and by Dr. Steffen Kopecki (S.K.). Two of them have also been co-authored by Prof. Oscar H. Ibarra (O.H.I.). By convention, all the authors of papers in theoretical computer science journals are ordered alphabetically by their last names. Below is a detailed description of the contributions of all the authors in the papers.

Paper #1, "A formal language model of DNA polymerase enzymatic activity" - Chapter 3

S.K.E. - modelling the bio-operation, framing the problem mathematically, formal definitions and initial approach, all the proofs, first manuscript draft, manuscript editing;

L.K. - topics, research ideas, manuscript writing and editing;

S.K. - research ideas and results, in particular the final version of results in Section 3.4, semantic proof revisions, manuscript editing.

Paper #2, "On the overlap assembly of strings and languages" - Chapter 4

S.K.E. - modelling the bio-operation, framing the problem mathematically, formal definitions and initial approach, first manuscript draft, manuscript editing, the preliminary results of Section 4.3 (these were later extended/superseded by results due to O.H.I., in the final version of this section of the paper. The preliminary results by S.K.E. are included as Appendix A of this thesis), a major part of Section 4.5;

O.H.I. - research ideas, Section 4.2 starting with paragraph, "We will use the following ...", all the results in Section 4.3 and in Section 4.4, manuscript writing and editing;

L.K. - topics, research ideas, manuscript writing and editing;

S.K. - research ideas, semantic proof revisions of some results in all sections, Lemma 4.5.1, manuscript editing.

Paper #3, “Further remarks on the overlap assembly operation” - Chapter 5

S.K.E. - research ideas, first manuscript draft, manuscript editing, results in Sub-section 5.2.2, Proposition 1, Proposition 2, Theorem 5.3.2, Proposition 3 (Section 5.3), Theorems 5.4.1, 5.4.2, 5.4.3 (Section 5.4);

O.H.I. - Sub-section 5.2.3, Theorem 5.3.1 (Section 5.3), Theorem 5.4.4 and Corollaries 5.4.5, 5.4.6, 5.4.7, 5.4.8 (Section 5.4), all the results in Section 5.5, manuscript writing and editing;

L.K. - topic, research ideas, manuscript writing and editing;

S.K. - results in Sub-section 5.2.2, Proposition 1, Proposition 2, Theorem 5.3.2, Proposition 3 (Section 5.3), Theorems 5.4.1, 5.4.2, 5.4.3 (Section 5.4), semantic proof revisions of some results in all sections, manuscript editing.

Acknowlegements

First and foremost, I thank my supervisor Prof. Lila Kari whose guidance and support has helped me throughout the period of my doctoral program. I am grateful to her for suggesting these research topics, helping me with engendering of appropriate and plausible research ideas and the constructive criticism of my results which helped improve them, among many others.

I convey my wholehearted and utmost thanks to Dr. Steffen Kopecki without whose help these research results in their current form would not have been possible. His help, particularly in proofreading my drafts and rewriting some of my proofs has been instrumental.

I convey my special thanks to Prof. Oscar H. Ibarra who has extended some results and added many other interesting results.

I am thankful to all the staff and other faculty at the Department of Computer Science for their amicable accessibility, assistance with various tasks and facilitating access to various resources.

This thesis would have been impossible if not for the support of my parents and my wife, especially the latter, whose constant encouragement and support has kept me in good stead in all situations and helped me complete my thesis.

Contents

[1]Reprinted from Fundamenta Informaticae, 138(1-2), S.K. Enaganti, L. Kari, S. Kopecki, A formal language model of DNA polymerase enzymatic activity, 179-192, Copyright (2015), with permission from IOS press

[2]A version of this chapter, including an abstract, has been submitted to the Natural Computing journal (S.K. Enaganti, O.H. Ibarra, L. Kari, S. Kopecki. On the overlap assembly of strings and languages.)

[3]A version of this chapter, including an abstract, has been submitted to the Information and Computation journal (S.K. Enaganti, O.H. Ibarra, L. Kari, S. Kopecki. Further remarks on DNA overlap assembly.)

List of Tables

List of Figures

Chapter 1

Introduction

Formal language theory is a fundamental part of theoretical computer science. In recent times, advances in molecular biology and biotechnology, particularly the ones related to the interactions of DNA molecules have inspired several new formal models in the field. Thomas J. Head, in his seminal paper "Formal language theory and DNA: an analysis of the generative capacity of specific recombinant behaviors" has initiated this area of research by formulating an operation modelling the actions of enzymes on DNA [9].

1.1 Motivation

The underlying impetus for this research stems from the area of DNA Computing which aims towards developing DNA and molecular biology hardware that can be used to solve computational problems more efficiently than the traditional silicon-based computers. Leonard Adleman, through his DNA-based experiments had solved an instance of the Hamiltonian Path Problem [1], which was a proof-of-principle that a computational task can be achieved using solely DNA-based molecular biology processes. Since then, in the last two decades, there have been several instances of algorithms being implemented experimentally using DNA and other bio-molecules such as RNA and proteins. This has also, in turn, inspired the development of several formal language models of DNA-based processes such as insertion-deletions, hairpin completion etc.

One of the primary goals in developing these models is to bring the study of DNA-information and DNA-computation into a unified formal framework. There has been an enormous amount of data that has been collected in many fields of molecular biology, but the unifying principles governing them are still to be discovered and formulated. There is a hope that these formal language models could contribute to condensing the vast knowledge in the field to basic concepts akin to physical sciences. Another goal has also been to implement molecular level automata that can implement logical functions using molecular phenomena. A practical application is a molecular automaton that can work in a living cell, assess the biochemical parameters within the cell in real time, and produce output molecules indicating the appropriate action such as drug-delivery at that point in time [2,3]. Such molecular automata have the advantage of being biological entities that are able to work at a microscopic level within a biological cell.

There are several operations inspired from enzymatic actions on DNA that have been proposed in the literature such as splicing, insertion and deletion, and hairpin extension. As partially referred to earlier, splicing is a formal language operation originally proposed by Tom Head [9] to model the recombination of DNA strands under the action of restriction enzymes and ligase enzymes. There have been various types of splicing systems that have been developed based on this phenomenon and their properties studied in, e.g., [7, 10, 12, 15, 21]. The operations of insertion and deletion 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., [6, 13, 14, 22–25]. Based on the phenomenon of hairpin formation, a naturally occurring phenomenon whereby a DNA strand that is partially self-complementary attaches to itself, 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, 16–18].

The main purpose of this thesis is to develop some more formal models that can describe

DNA-molecular actions in a formal way, and also to study the properties of these formal models from a computational perspective. In particular, the focus is on formalizing the action of DNA polymerase enzymes over DNA strands as a formal operation. There have been a few models inspired by the action of polymerase enzyme such as hairpin completion, hairpin reduction [4, 16–18] and overlapping concatenation [19].

The next section gives a brief introduction to the molecular biology of DNA and some enzymes whose actions have inspired the bio-operations referred to earlier, and the ones studied in this thesis. This background is given to enable any general computer science audience to understand the rest of the contents of the thesis.

1.2 Molecular biology basics

1.2.1 DNA structure

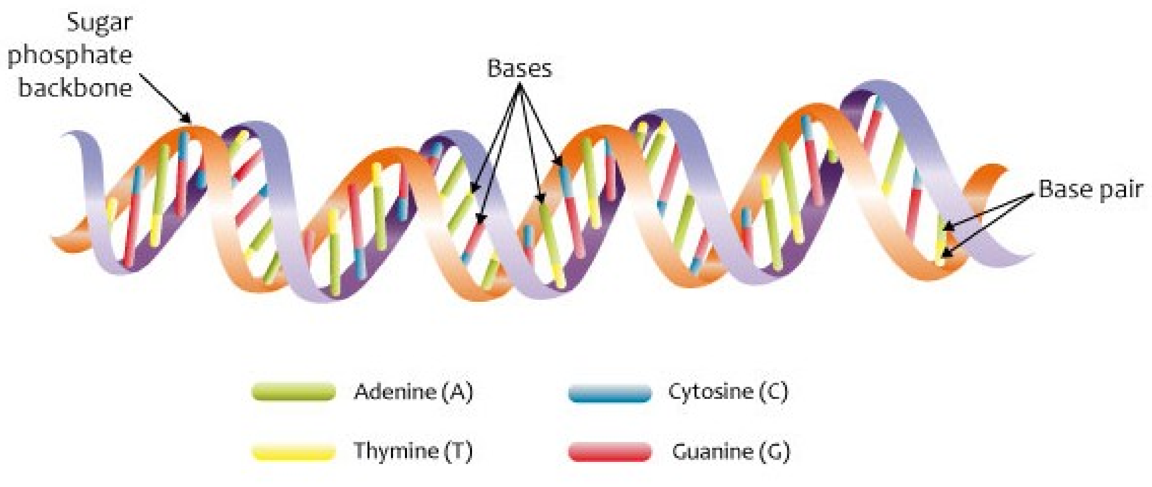

Figure 1.1: DNA structure (Source: [8])

Deoxyribonucleic acid (DNA) is the blueprint of life. It consists of a linear sequence of units called "nucleotides" connected through a backbone consisting of sugar and phosphate groups. Each nucleotide is associated with a base and the nucleotides differ only by presence of a different base. There are four nucleotides, each having one of the four bases: Adenine, Thymine, Guanine and Cytosine, represented by letters A, T, G and C respectively. The nucleotides exhibit complementary behaviour with each other according to Watson-Crick complementarity: Adenine and Thymine are complementary to each other, and Guanine and Cytosine are complementary to each other. Adenine-Thymine and Guanine-Cytosine are sometimes termed as Watson-Crick base pairs. The sugar consists of five carbon atoms which are numbered 1’ (one prime) to 5’ (five prime). The 1’-carbon is connected to the base and the 3’ and 5’ carbons connect with the phosphate group. By convention, any (single) strand of DNA is given a direction where the two ends of the DNA strand sequence are identified as the 5’-end and the 3’-end respectively based on 5’ and 3’ carbon atoms of nucleotides that flank each end respectively. To identify relative positions within a DNA sequence, the terms upstream and downstream are used. By convention, upstream is towards the 5’ end and downstream is towards the 3’ end of the molecule. DNA is often double stranded and the composing single strands are anti-parallel, i.e., they run in 5’–3’ and 3’–5’ directions respectively. In addition, the sequences of the single strands that comprise the DNA double strand have complementary bases at each position. For example, consider the double stranded DNA shown in Figure 1.2 where one strand has the sequence 5’-ATGCTC-3. The other strand will be reverse complementary to this and has the sequence 5’-GAGCAT-3’ (read in the direction of 5’ to 3’ by convention).

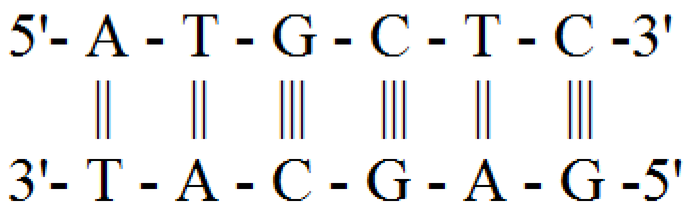

Figure 1.2: A double-stranded DNA sequence

The sugar-phosphate backbone is held together with phosphodiester bonds, which are the covalent bonds between the phosphate group and the sugar at its 3' and 5' ends respectively. The complementary bases are held together through hydrogen bonds between electronegative hydrogen atoms and a nitrogen or oxygen atom on the other base, and other forces such as Van der Waals forces, see Figure 1.1 and Figure 1.2. In Figure 1.2, one can see two lines between A and T and three lines between G and C indicating the number of hydrogen bonds involved in the respective base-pairs.

RiboNucleic Acid (RNA) also plays a vital role in life processes. RNA molecules are similar to DNA molecules but they have the nucleotide Uracil (represented as U) instead of Thymine (T), and the sugar in the sugar-phosphate backbone is ribose and not 2-deoxyribose like in DNA. RNA strands are often single-stranded and can form double strands with other self complementary RNA or DNA strands.

1.2.2 Enzymes

Enzymes are large biological molecules (mostly proteins) responsible for accelerating, or catalyzing many chemical reactions that sustain life. The primary role of enzymes is to catalyze reactions, by accelerating the rate and specificity of metabolic reactions ranging from the digestion of food to the synthesis of DNA. They are highly selective with respect to their active sites and their actions. The chemical compound they act upon is called the substrate.

Enzymes catalyze a wide variety of reactions and their standard classification is based on the types of reactions that they are involved in. There have been six major classes identified, namely oxidoreductases, transferases, hydrolases, lyases, isomerases and ligases. Within the scope of this thesis, we deal with three types of enzymes, polymerases (a special kind of transferases), restriction enzymes (which are transferases and hydrolases) and DNA ligases.

Polymerases

A polymerase is an enzyme whose central biological function is the synthesis of polymers of nucleic acids (DNA and RNA). Polymerases work by using an existing strand as an information template and then joining nucleotides complementary to it to form a new strand complementary to the original. They typically require a primer, which is a short strand of DNA/RNA (generally about 18-22 nucleotides long) that serves as a starting point for DNA/RNA synthesis.

The most important classes of polymerases are DNA polymerases and RNA polymerases. They are further divided into two classes each depending on the template strands (which are the strands the enzymes derive their information from). DNA polymerases are used to replicate existing DNA (as in cell division) are called DNA-based DNA polymerases. DNA polymerases that are used to assemble a DNA strand using an existing RNA strand as a template are called RNA-based DNA polymerases. In this thesis, the word DNA polymerase is exclusively used to refer to DNA-based DNA polymerase enzyme. Analogously, RNA polymerases are used to assemble RNA molecules using an existing DNA or RNA strand as a template and assembling appropriate nucleotides using base-pairing interactions. There are many important subclasses of polymerases such as reverse transcriptases used by viruses such as HIV, which generate complementary DNA (cDNA) from an RNA template through a process called reverse transcription.

The bio-operations that are studied in this thesis are almost exclusively inspired by the extension activity of the DNA polymerase enzymes. More details about DNA polymerase enzymes and some of their applications will be discussed in Sub-section 1.2.3.

Restriction enzymes

A restriction enzyme (also called restriction endonuclease) is an enzyme that cuts DNA at or around a specific recognition nucleotide sequence known as restriction site. The restriction enzymes are classified based on various factors such as the differences in their structure, nature of the substrate that they act upon, or if the recognition and cleavage sites are separate from one another, or if they cut their DNA substrate near their recognition site or far away, among others. All the naturally occurring restriction enzymes are classified into four major types referred to as type I, type II, type III and type IV restriction enzymes. Type I enzymes cleave very far (sometimes in the order of thousands of nucleotides away) from the recognition site and often at random locations. Type II enzymes cleave within a short specific distance (typically less than 20 nucleotides) from the recognition site and do not have a methylase function (which is replacing a hydrogen atom with a methyl (CH_3) group). Some of the type II enzymes cut within the recognition site and some of them cut outside the recognition site. For example, the type IIS (a subtype of type II) restriction enzyme *Fok*I which has 5'-GGATG-3' recognition site is known to cut 9 nucleotides downstream and 13 nucleotides upstream of the nearest nucleotide of the recognition site. Type III enzymes cleave within a short specific distance from the recognition site but exist as part of a complex which has a methylase activity. Type IV enzymes target only modified DNA such as methylated DNA. Among all these, type II restriction enzymes are the most abundant and widely studied. A detailed discussion about the structure and mechanism of all sub-types of type II restriction enzymes is presented in [20].

A key feature of restriction enzymes is that, when they cut a double-stranded DNA molecule, different types of ends are produced. The two important types of ends produced are blunt ends and sticky ends. In a blunt-ended DNA molecule, both strands terminate in a base pair. A sticky end is an overhang, i.e., a stretch of unpaired nucleotides at the end of a DNA molecule, of length at least two. These unpaired nucleotides can be in either strand, creating either 3' or 5' overhangs.

There also exist some special variants of restriction enzymes called nickases (also called nicking endonucleases), which instead of cleaving both the strands of a double-stranded DNA cleave only one of the strands and thus produce DNA molecules that are “nicked” rather than cleaved.

DNA ligases

DNA ligases are a class of enzymes which facilitate the joining of DNA strands together by catalyzing the formation of a phosphodiester bond (the covalent bond linking the 5' or 3' carbon atom of a sugar with a phosphate group) between adjacent nucleotides of the same strand. They have become an indispensable tool in modern molecular biology research for generating recombinant DNA molecules (DNA molecules that are formed by the combination of DNA sequences from multiple sources). They are used, along with restriction enzymes, to insert DNA fragments such as genes into plasmids (which are small and often circular DNA sequences found in some organisms).

1.2.3 DNA polymerases and polymerase chain reaction

The DNA polymerase enzymes are among the most important enzymes in a cell since they are useful for DNA replication, a process that is needed in cell division. They have also been proven to be vital in the field of biotechnology, particularly for their application in the technique called polymerase chain reaction.

Polymerase chain reaction

The polymerase chain reaction (PCR) is a widely-used technique in molecular biology and biotechnology that is used for many applications including genotyping, cloning, mutation detection, sequencing, forensics, and paternity testing. Basically, it is a process that extracts a desired subsequence from a longer DNA sequence by exponentially replicating copies of the sub-sequence. The sub-sequence that is amplified is usually referred to as an "amplicon".

Most often, a DNA polymerase called the *Thermus acquaticus (Taq)* polymerase (extracted from a thermophilic bacterium that lives in thermal hot springs) is used in the PCR process, as it is a heat resistant polymerase that is more resistant to the temperature changes, and works actively at its normal optimal temperature (which is higher than the normal optimal temperature for most other polymerases which is around 37). It acquired vital commercial importance due to its widespread use in PCR.

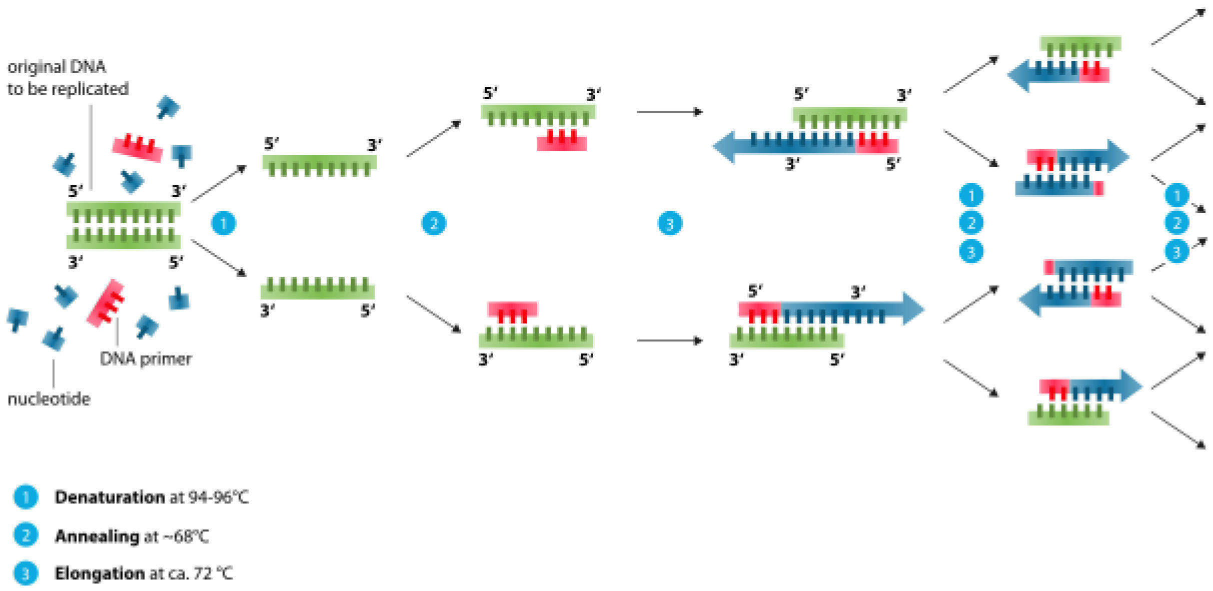

Figure 1.3: Polymerase chain reaction (Source: [11])

A PCR process consists of usually many cycles that are repeated until a sufficient amount of the desired output is obtained as shown in Figure 1.3. The process takes a double-stranded DNA molecule containing the desired amplicon as a subsequence as an input, along with two short single DNA strands that can work as primers. The short strands introduced are chosen such that one of them matches exactly with the prefix and the other is Watson-Crick complementary to the suffix of the desired amplicon. Each PCR cycle consists of three stages: denaturation, annealing and elongation. In the stage of denaturation, the solution is heated to temperatures of up to 95°C. At such high temperatures, the double-stranded DNA breaks up into its constituent single strands. In the annealing stage, the temperature is lowered to 68°C (or similar temperature depending on the primers). The primers attach themselves to either of the template strands at the respective places, where Watson-Crick complementary sequences are found. In the extension stage, the temperature is increased slightly to the polymerase optimal temperature (around 75°C for *Taq* polymerase). At this stage, the polymerase enzyme is most active and starts elongating the primers in the 5' to 3' direction of primer. The cycles are repeated 20–40 times (depending on the purpose) and, at the end of final extension, the reaction mixture is held at 4 degrees and it contains the desired product (primer delimited) in great excess.

1.3 Summary and organization of the thesis

In this thesis, two different models of computation are studied: "Directed Extension" and "Overlap Assembly". The directed extension is an operation that models the action of the DNA polymerase enzyme over DNA strands. It takes two strings as input representing the template strand (providing information for extension) and primer (providing the starting point of extension), respectively, and produces output strings representing all possible new strands that may be generated by the extension of the primer according to the template. The overlap assembly is an operation between two different words with a partial overlap, where the suffix of one overlaps with the prefix of the other, and produces a new word that is a concatenation of both the words without the repetition of the common sub-word. Overlap assembly of DNA strands can also be achieved by the action of the DNA polymerase enzyme.

Chapter 2 gives a survey of research literature, describing several language operations that were inspired by enzymatic actions on DNA.

Chapters 3, 4 and 5 constitute the original research contributions of this thesis.

Chapter 3 includes a study of the directed extension operation, formulated by the author of the thesis. The operation is defined between words, and then extended to languages. Section 3.3 studies the closure properties of various language classes, including the ones in the Chomsky hierarchy, with respect to the operation. In Section 3.4, an inverse of the operation is defined, and an evaluation of the necessary and sufficient conditions for the existence of an inverse is conducted. Then, some related questions with respect to language equations are resolved.

Chapter 4 discusses the operation of overlap assembly. A study of closure properties of the language classes within the Chomsky hierarchy, and various other classes, under overlap assembly is put forth in Section 4.3. Section 4.4 deals with related decision problems. In Section 4.5, an iterated version of the overlap assembly operation is studied. Then a theoretical analysis of how it can be applied to produce a DNA combinatorial library, useful for implementing many DNA-based algorithms, is presented.

Chapter 5 probes further into the properties of the overlap assembly operation. Section 5.3 compares overlap assembly with the related superposition operation and shows how all the positive closure properties of the iterated version of the former follow from the latter. Using this, closure properties of various language classes with respect to iterated overlap assembly are re-established. Although they were previously studied in [5], some of the existing results are strengthened here. Section 5.4 gives a study of the properties of the terminating sets of a language, previously defined in Chapter 4. Section 5.5 studies some decidability problems. The first problem is to decide whether a given language is complete. The other problems are to decide if a string is terminal with respect to a language, and to decide if a language can be generated by two other given languages.

In Chapter 6, some concluding remarks and future directions of work are given.

Bibliography

[1] L. M. Adleman. Molecular computation of solutions to combinatorial problems. *Science*, 266(5187):1021–1024, 1994.

[2] Y. Benenson, B. Gil, U. Ben-Dor, R. Adar, and E. Shapiro. An autonomous molecular computer for logical control of gene expression. *Nature*, 429(6990):423–429, 2004.

[3] Y. Benenson, T. Paz-Elizur, R. Adar, E. Keinan, Z. Livneh, and E. Shapiro. Programmable and autonomous computing machine made of biomolecules. *Nature*, 414(6862):430–434, 2001.

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

[5] E. Csuhaj-Varjú, I. Petre, and G. Vaszil. Self-assembly of strings and languages. *Theoretical Computer Science*, 374(1-3):74–81, 2007.

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

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

[8] N. N. Genetics and G. E. C. [(http://creativecommons.org/licenses/by sa/4.0)]. DNA double helix. `http://www.geneticseducation.nhs.uk/blog/wp-content/uploads/2014/03/DNA-diagram.jpg`, 2014.

[9] 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.

[10] T. Head, D. Pixton, and E. Goode. Splicing systems: regularity and below. In M. Hagiya and A. Ohuchi, editors, *DNA Based Computers: DNA Computing,* DNA 8, volume 2568 of *LNCS*, pages 262–268, 2003.

[11] E. [(http://creativecommons.org/licenses/by sa/3.0)]. Polymerase chain reaction. `https://commons.wikimedia.org/wiki/File:Polymerase_chain_reaction.svg`, 2014.

[12] 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 (DNA 18)*, volume 7433 of *LNCS*, pages 98–109, 2012.

[13] 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*, pages 329–346, 1999.

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

[15] S. M. Kim. An algorithm for identifying spliced languages. In T. Jiang and D. Lee, editors, Proc. *Computing and Combinatorics Conference,* COCOON, volume 1276 of *LNCS*, pages 403–411, 1997.

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

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

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

[19] C. Martín-Vide, G. Păun, J. Pazos, and A. Rodríguez-Patón. Tissue P systems. *Theoretical Computer Science*, 296(2):295–326, 2003.

[20] A. Pingoud, M. Fuxreiter, V. Pingoud, and W. Wende. Type II restriction endonucleases: structure and mechanism. *Cellular and Molecular Life Sciences*, 62(6):685–707, 2005.

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

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

[23] G. Păun, G. Rozenberg, and A. Salomaa. *DNA Computing: New Computing Paradigms*. Springer-Verlag New York, Inc., 1998.

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

[25] 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.

Chapter 2

Literature review

Summary

Computational models based on DNA bio-operations are an important part of formal language theory due to the fact that they can be useful in directly simulating naturally occurring DNA/RNA processes. Since Tom Head's paper [13] which defined "splicing" as a formal language operation modelling recombinant DNA, several other DNA-based bio-operations have been introduced and widely studied. Herein, we present a comprehensive overview of computational systems based on bio-operations such as splicing systems, insertion-deletion systems and hairpin completion-reduction systems. Besides describing their biological basis and motivations, we focus on noteworthy computational properties of such systems such as universal computability, closure properties, complexity, and reversibility.

2.1 Introduction

There has been a wide variety of formal operations proposed in the literature, to model the actions of enzymes on DNA strands. Formal systems based on such bio-operations include splicing systems, insertion-deletion systems, hairpin completion-reduction systems etc.

Since a DNA strand is essentially a linear sequence over an alphabet of 4 nucleotides (given by the respective letters), it can be modelled to represent information or data. As early as 1980, Ebling and Jiménez-Montaño have used context-free and context-sensitive grammars for the description of polypeptides and polynucleotides [8].

The roadmap of this chapter is as follows: In Section 2.2, we briefly give the notations followed in the chapter. In Section 2.3, we discuss splicing systems, inspired by recombinant behaviour of DNA. In Section 2.4, we talk about insertion-deletion systems and in Section 2.5, we talk about hairpin completion-reduction systems. Finally, we give some of our conclusions in Section 2.6.

2.2 Definitions and notations

An alphabet A is a finite non-empty set of symbols. A^* denotes the set of all words over A, including the empty word λ. A^+ is the set of all non-empty words over A. The length of a word $x \in A^*$ is given by $|x|$. For words w, x, y, z such that $w = xyz$ we call x, y, and z *prefix*, *infix*, and *suffix* of w, respectively. The sets $\mathrm{pref}(w)$, $\mathrm{inf}(w)$, and $\mathrm{suff}(w)$ contain, respectively, all prefixes, infixes, and suffixes of w. A prefix (resp., infix or suffix) x of w is *proper* if $x \neq w$. We employ the following notation: $\mathrm{Pref}(w) = \mathrm{pref}(w) \setminus \{w\}$, $\mathrm{Inf}(w) = \mathrm{inf}(w) \setminus \{w\}$, and $\mathrm{Suff}(w) = \mathrm{suff}(w) \setminus \{w\}$. This notation is naturally extended to languages; for example, $\mathrm{Suff}(L) = \bigcup_{w \in L} \mathrm{Suff}(w)$.

θ is called a *morphism* if $\theta(uv) = \theta(u)\theta(v)$ for all $u, v \in A^*$. θ is called an *anti-morphism* if $\theta(uv) = \theta(v)\theta(u)$ for all $u, v \in A^*$. A function $I : A^* \to A^*$ is called an involution if $I(I(x)) = x$ for all $x \in A^*$. Traditionally, the Watson-Crick complementarity of DNA strands has been modelled as an anti-morphic involution over the DNA alphabet $\Delta = \{A, C, G, T\}$.

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. For basic elements of formal language theory, we refer to [18].

2.3 Splicing systems

2.3.1 Introduction

The formal language operation of splicing is designed to model DNA recombination under the action of restriction enzymes and ligases. When a double-stranded DNA strand is acted upon by certain restriction enzymes, the result is that the DNA strand is cut at specific locations, giving rise to sticky or blunt ends (see Section 1.2.2). The sticky ends formed can sometimes bind to sticky ends of different DNA strands if they are Watson-Crick complementary. Such pairs of sticky ends are termed compatible sticky ends. The DNA ligase can then join the backbone, completing thus the formation of the new double strand. Some common examples of restriction enzymes that are used for such reactions are *Taq*I, *SciN*I, *BamH*I and *Bgl*II.

We illustrate this process with the example in Figure 2.1. Let there be two double-stranded sequences in the solution, with one of them being $5' - \alpha - TCGA - \beta - 3'$ and the other being $5' - \gamma - GCGC - \delta - 3'$ where α, β, γ and δ are sequences of DNA. Assume that we have two restriction enzymes *Taq*I and *SciN*I added to reaction mixture. Given appropriate conditions, the enzymes will recognize their respective recognition sequences in both the strands and cut them according to their restriction mechanism. The enzymes *Taq*I and *SciN*I will produce the sticky ends as shown in Figure 2.1. The compatible sticky-ends can bind together and result in the formation of a new strand $5' - \alpha - TCGC - \delta - 3'$ under the action of DNA ligase enzyme. Similarly, the other two sticky ends can bind together and form $5' - \gamma - GCGA - \beta - 3'$. This is in effect the bio-operation of splicing.

The formal operation of splicing was introduced by Tom Head to formalize the recombinant behaviour of DNA strands under the action of restriction enzymes and ligases [13].

In the next sub-sections, we give definitions for splicing systems and follow that up with their properties, and different types of splicing systems. In Sub-section 2.3.4, we then define circular splicing systems and their properties. Finally we give some of the results in literature

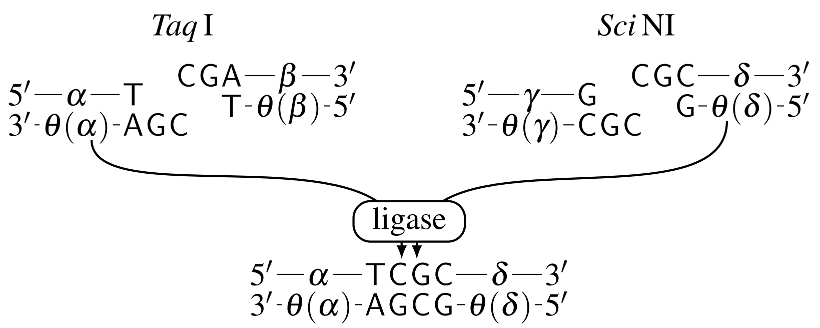

Figure 2.1: The bio-operation of splicing: The input strands $5' - \alpha - TCGA - \beta - 3'$ and $5' - \gamma - GCGC - \delta - 3'$ are acted upon by the restriction enzymes *Taq*I and *SciN*I and these enzymes break the strands at the restriction sites (locations indicated) resulting in the formation of sticky ended DNA strands. Later, two of the strands with compatible sticky ends bind to each other and with the help of DNA ligase enzyme form the new strand $5' - \alpha - TCGC - \delta - 3'$. The other two sticky ends can also bind and would analogously form the strand $5' - \gamma - GCGA - \beta - 3'$ (Not shown in the picture).

about regularity properties of either types of systems.

2.3.2 Definitions of splicing systems

There are three important definitions of splicing systems that are well-studied in the literature.

Definition (Head's [13]) A splicing system $S_H = (A, I, B, C)$ consists of a finite alphabet A, a finite set $I \subseteq A^*$ of initial strings, and finite sets B and C of triples (c, x, d) with c, x and d in A^*. Each such triple in B or C is called a pattern. For each such triple the string cxd is called a site and the string x is called a crossing. Patterns in B are called left patterns and patterns in C are called right patterns. The left patterns are associated with restriction enzymes producing 5' overhangs and the right patterns are associated with restriction enzymes producing 3' overhangs. Splicing can happen between two words only if both the corresponding patterns involved in the operation are either left (i.e. both are in B) or right (i.e. both are in C). Given two words $ucxdv$ and $pexfq$ in L, if (c, x, d) and (e, x, f) are patterns in B (resp. C), the splicing operation generates the strings $ucxfq$ and $pexdv$.

Definition A splicing scheme is a pair $\sigma = (A, R)$, where A is an alphabet and $R \subseteq A^*\#A^*\$A^*\#A^*$

is a set of splicing rules.

Definition (Paun's [44]) A splicing system $S_{PA} = (A, I, R)$ consists of a finite set $I \subseteq A^*$ as the initial language and a corresponding splicing scheme (A, R). Each rule r in R is of the form $r = (u_1, u_2; u_3, u_4)$ (and represented by the string $u_1\#u_2\$u_3\#u_4$), with $u_i \in A^*$, for $i = 1, 2, 3, 4$ and $\#, \$ \notin A$. Given two words $x = x_1u_1u_2x_2$, $y = y_1u_3u_4y_2$, and the rule $r = u_1\#u_2\$u_3\#u_4$, the splicing operation produces $w = x_1u_1u_4y_2$ and $w' = y_1u_3u_2x_2$. Formally, we can write $(x, y) \vdash_r \{w, w'\}$.

The Paun's definition of splicing is the most widely used model of splicing system in the area of DNA computing.

Definition (Pixton's [41]) A splicing system $S_{PI} = (A, I, R)$ consists of a finite alphabet A, a finite set of strings $I \subseteq A^*$ as initial language, a set of rules R where for all r in R, we have $r = (\alpha, \alpha'; \beta)$, for $\alpha, \alpha', \beta \in A^*$. Given two words $x = \varepsilon\alpha\eta$, $y = \varepsilon'\alpha'\eta'$ and the rule $r = (\alpha, \alpha'; \beta)$, the splicing operation produces $w = \varepsilon\beta\eta'$ and $w' = \varepsilon'\beta\eta$. Formally, we can write $(\varepsilon\alpha\eta, \varepsilon'\alpha'\eta') \vdash_r \{\varepsilon\beta\eta', \varepsilon'\beta\eta\}$.

Note that in Pixton's definition, the word β is introduced to make the system more generic than Paun's system. This way we are not only cutting and pasting at the recognition site but also substituting it with a new word given by the splicing rule.

Every splicing system generates a language by the iterated application of splicing rules to its initial language. Thus, every splicing system is associated with a corresponding splicing language. Formally, let $R^0(L) = L$ and $R(L) = \{w \in A^* \mid \exists w', w'' \in L, \exists r \in R : (w', w'') \vdash_r w\}$. For each non-negative integer i, we have $R^{i+1}(L) = R^i(L) \cup R(R^i(L))$. The language $R^*(L) = \cup\{R^i(L) : i \geq 0\}$ is the language generated from L through the iterated application of the rule set R.

A language is said to be a splicing language if there is a splicing system that can generate it. Formally, a language L is a splicing language if $L = R^*(I)$ for some splicing system $S = (A, I, R)$ (defined by either Paun's or Pixton's definitions).

A splicing system $S = (A, I, R)$ is said to be a finite splicing system if both R and I are finite sets. A language generated by such a system is called finite splicing language. For finite splicing systems, it was proved that the family of languages generated by Head's system is strictly included in the family generated by Paun's system which is in turn strictly included in the family generated by Pixton's system [5].

Sometimes, splicing systems that have a splicing scheme with some restrictions are studied.

Definition [15] A splicing scheme R is *reflexive* if for every splicing rule $(u, u'; v', v)$ (Paun's definition) in R, there is a corresponding rule $(u, u'; u, u')$ that is in R. A splicing system using a reflexive splicing scheme is called reflexive splicing system and correspondingly, the language generated by such a system is called as a *reflexive splicing language*.

Definition [15] A splicing scheme R is *symmetric* if for every splicing rule $(u, u'; v', v)$ (Paun's definition) in R, there is a corresponding rule $(v', v; u, u')$ that is in R. A splicing system with a symmetric splicing scheme is called a symmetric splicing system and correspondingly, the language generated by such a system is called as a *symmetric splicing language*.

Restricted versions of splicing systems based on the type and size of splicing rules are also defined. These include the following:

Definition [40] A splicing system $S = (A, I, R)$ (Paun's version) in which all rules in R have the form $(a, \lambda; a, \lambda)$ where $a \in A$ is called *simple splicing system*.

Definition [11] A splicing system $S = (A, I, R)$ (Paun's version) in which I and A are finite and every rule in R has the form $(a, \lambda; b, \lambda)$, where a, b are in A is called a *semi-simple splicing system*.

Definition [13] A *null-context splicing system* is a splicing system $S = (A, I, B, C)$ (Head's version) for which each cleavage pattern in B and C has the form (λ, x, λ).

Definition [28] A splicing system (A,I,R) (Paun's version) in which I and R are finite and every rule in R has the form $(u,\lambda;v,\lambda)$, where u , v are in A^+ is called a *semi-null splicing system.*

Definition [13] A *uniform splicing system* is a null-context splicing system $S=(A,I,X,X)$ (Head's version) for which there is a positive integer P such that $X=A^P$.

Definition [1] A splicing system $S=(A,I,R)$ (Paun's version) in which A is finite and every rule in R has the form $(u_1,u_2;u_3,u_4)$, where u_1,u_2,u_3,u_4 are in A or equal to λ is called an *alphabetic splicing system.*

Several relationships among the splicing systems defined above are established: If $\mathbb{A},\mathbb{B}$ are two classes of splicing systems, let $\mathbb{A}\subseteq\mathbb{B}$ (i.e., subset inclusion) mean that $\mathbb{A}$ is a special case of $\mathbb{B}$. Then, it was proved that, simple splicing system $\subseteq$ semi-simple splicing system $\subseteq$ semi-null splicing system $\subseteq$ uniform splicing system $\subseteq$ null-context splicing system [49,55].

2.3.3 Closure properties of splicing systems

The study of closure properties is of particular interest for finite splicing systems. Culik and Harju proved that every finite splicing system (Head's definition) produces a regular language [19]. Pixton proved the same property for Paun's variant and then for his own variant of splicing [41]. Gatterdam gave an example of a regular language that cannot be generated by any finite splicing system [10]. A natural question was if it can be decided whether any given regular language can be generated by some finite splicing system.

There were several attempts for solving this problem that have been proposed in the literature. Kim has solved the problem for a special case of regular languages [26]. Goode, Head and Pixton have resolved the special case of determining if a regular language can be generated by a reflexive splicing system [16]. Finally, Kari and Kopecki solved the problem for the general case of the problem for all variants of splicing [21].

It was proved that splicing systems with suitable I and R can reach the power of any Turing machine [17, 44]. Table 2.1 summarizes the closure properties of splicing systems. The rows represent the class to which the initial language I belongs to and the column represents the class in which the language of rules R is in.

I\R	Fin	Reg	CF	CS	RE
Fin	REG	RE	RE	RE	RE
Reg	REG	RE	RE	RE	RE
CF	CF	RE	RE	RE	RE
CS	RE	RE	RE	RE	RE
RE	RE	RE	RE	RE	RE

Table 2.1: Closure properties of splicing systems [56]

2.3.4 Definitions of circular splicing systems

Splicing systems were introduced for circular languages by Tom Head in [14].

Definition An equivalence relation is a binary relation $\sim$ satisfying three properties:

- For every element a in X, $a \sim a$ (reflexivity),
- For every two elements a and b in X, if $a \sim b$, then $b \sim a$ (symmetry),
- For every three elements a, b, and c in X, if $a \sim b$ and $b \sim c$, then $a \sim c$ (transitivity).

The equivalence class of an element a is denoted $[a]$ and is defined as the set

$$[a] = \{x \in X \mid a \sim x\}.$$

Definition A circular word is an equivalence class with respect to the conjugacy relation $\sim$ defined by $xy \sim yx$, for x, y in A^*. A circular language $C = \sim L$ is a set of circular words.

Definition A full linearization of a circular language C, denoted by $Lin(C)$, is the set of all the words in A^* corresponding to the elements of C, i.e. $Lin(C) = \{w \in A^* \mid \sim w \in C\}$.

Definition A circular language obtained from a regular language is called circular regular language (belong to class $REG \sim$). Formally, $REG \sim = \{C \subseteq \sim A^* \mid \exists L \in REG :$ such that $\sim L = C\}$. It is observed that $C \in REG \sim$ if and only if $Lin(C)$ is regular.

There are three main types of splicing systems based on circular languages.

Definition (Head's) [14] A circular splicing system $SC_H = (A, I, T, P)$, where $I \subseteq \sim A^*$ is the initial circular language, $T \subseteq A^* \times A^* \times A^*$ and P is a binary relation on T, such that if (p, x, q), $(u, y, v) \in T$ and $(p, x, q)P(u, y, v)$ then $x = y$. Given $\sim hpxq$, $\sim kuxv \in \sim A^*$ with $(p, x, q)P(u, x, v)$, the splicing operation produces $\sim hpxvkuxq$.

Definition (Paun's) [17] A system $SC_{PA} = (A, I, R)$, where $I \subseteq \sim A^*$ is the initial circular language, $R \subseteq A^*\#A^*\$A^*\#A^*$, with $\#, \$ \notin A$, is the set of rules. Then given a rule $r = u_1\#u_2\$u_3\#u4$ and two circular words $\sim hu_1u_2$, $\sim ku_3u_4$, the rule cuts and linearizes the two strings obtaining u_2hu_1 and u_4ku_3, and pastes and circularizes them obtaining $\sim u_2hu_1u_4ku_3$. Formally, we can write $(\sim hu_1u_2, \sim ku_3u_4) \vdash_r \sim u_2hu_1u_4ku_3$.

Definition (Pixton's) [41] A system $SC_{PI} = (A, I, R)$, where $I \subseteq \sim A^*$ is the initial circular language, $R \subseteq A^* \times A^* \times A^*$ is the set of rules. R is such that for every $r = (\alpha, \alpha'; \beta) \in R$ there exists β' such that $\bar{r} = (\alpha', \alpha; \beta') \in R$. Given rules r, $\bar{r}$, and two circular words $\sim \alpha\varepsilon$, $\sim \alpha'\varepsilon'$, the two rules r, $\bar{r}$ cut and linearize the two strings, obtaining $\varepsilon\alpha, \varepsilon'\alpha'$ and then paste, substitute and circularize them producing $\sim \varepsilon\beta\varepsilon'\beta'$. Formally, we can write $(\sim \alpha\varepsilon, \sim \alpha'\varepsilon') \vdash_{r,\bar{r}} \sim \varepsilon\beta\varepsilon'\beta'$. Any pair of rules $(r, \bar{r})$ of the given form can be used.

Analogous to linear splicing systems, every circular splicing system generates a circular language by the iterated application of splicing rules to its initial circular language. Thus, every circular splicing system is associated with a corresponding circular language called circular splicing language. Formally, let $R^0(L) = L$ and for any language $C \subseteq \sim A^*$, let $R(C) = \{w \in \sim A^* \mid \exists w', w'' \in C, \exists r \in R : (w', w'') \vdash_r w\}$. For each non-negative integer i, we have $R^{i+1}(L) = R^i(L) \cup R(R^i(L))$. The language $R^*(L) = \cup\{R^i(L) : i \geq 0\}$ is the language generated from

L through the iterated application of the rule set R. We note that for Pixton's systems, the splicing operation is combined action of the pair of rules r and $\bar{r}$.

The definitions of splicing scheme, and the definitions of splicing systems based on types of splicing schemes (such as symmetric/reflexive splicing) are all defined analogously for circular splicing systems as they are for linear splicing systems. Restricted versions of circular splicing systems based on the type and size of splicing rules are also analogously defined as that of linear splicing systems.

It was proven that Head's splicing system and Paun's splicing system are equally powerful [2,4] and Pixton's splicing system is more powerful than these [4].

Closure properties of circular languages

The computational power of circular splicing systems is unknown even if I and R are finite (without any other assumptions) [56]. Many questions such as characterizing the class of languages that finite circular splicing systems generate, are still to be answered. Unlike the linear case, it is proved that a finite circular splicing system (Head's definition) can generate non-regular languages [4]. Some results related to finite circular splicing systems and circular regular languages are given below.

Pixton proved that if R is a symmetric and reflexive splicing system scheme and C_0 is a circular regular language (i.e. $C_0 \in REG\sim$) then the circular splicing language $L(S_{PI})$ determined by $S_{PI} = (A, R, C_0)$ (Pixton's definition), is regular [41]. Bonizzoni et al. proved that it is decidable whether a circular regular language L on a one-letter alphabet is generated by a finite (Paun) circular splicing system in [3]. They also proved that there exists a circular regular language that cannot be generated by any finite circular splicing system. Berstel et al. proved that given a circular splicing language and a circular regular language, it is decidable whether they are equal [1]. They also proved that the language generated by a finite alphabetic circular splicing system is always context-free.

2.4 Insertion deletion systems

2.4.1 Introduction

The formalization of the insertion of a string into another has been considered first with linguistic motivation [37] already in the 60's. The insertion operation and its iterated variant is considered as a generalization of Kleene's operations of concatenation and closure [12].

The operations of insertion and deletion are also of interest in the field of molecular genetics. Gene insertion-deletion operations are basic to DNA processing and RNA editing in molecular biology. For example, it was reported that the insertion/deletion of a DNA fragment can effect the cell life [47]. There are many occasions where insertion-deletions correspond to a mismatched annealing of DNA sequences [46].

The operation of insertion of a word into another results in the set of all words that are formed when the first word is inserted into the second word (most often in a specified context). The operation of deletion of a word from another word results in all possible words obtained by deletion of all occurrences one at a time of the first word from the latter. The operations can be naturally extended to languages.

Before delving into the definitions of the formal model, a biological basis for the insertion operation will be explained below, with an example. Let us imagine a situation where we have a DNA strand x_1uvx_2z in the test-tube where x_1, x_2, u, v and z are all strings. If we add a strand $\theta(u)\theta(y)\theta(v)$ into the test tube where $\theta(u)$ and $\theta(v)$ are the Watson-Crick complements of u and v respectively and $\theta(y)$ is the Watson-Crick complement of y, then the following process illustrated in Figure 2.2, can take place.

The strand x_1uvx_2z will anneal to $\theta(u)\theta(y)\theta(v)$ such that u sticks to $\theta(u)$ and v sticks to $\theta(v)$ and the strand $\theta(y)$ will fold up as in (b) of Figure 2.2. When the phosphodiester bond between u and v is broken by a special type of restriction enzyme such as a nickase (which can

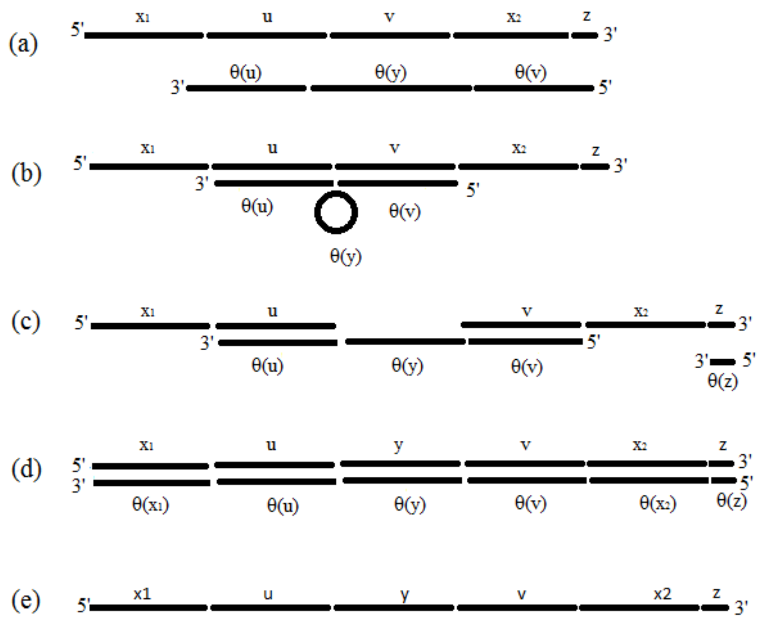

Figure 2.2: A biochemical implementation of the insertion operation. (a) Initially, there are two single strands of DNA x_1uvx_2z and $\theta(v)\theta(y)\theta(u)$ in the reaction mixture. (b) u anneals to $\theta(u)$ and v anneals to $\theta(v)$ due to Watson-Crick complementarity. (c) Then the phosphodiester bond between segments u and v is broken by a special type of restriction enzyme such as nickase and a DNA polymerase enzyme along with a short strand $\theta(z)$ (that shall act as a primer) are added to the reaction mixture. (d) The primer $\theta(z)$ anneals to the segment z of the strand x_1uvx_2z and then the DNA polymerase enzyme extends it. With the help of DNA ligase, this eventually results in the formation of the double strand of $\theta(z)\theta(x_2)\theta(v)\theta(y)\theta(u)\theta(x_1)$ and its complement u_1uvx_2z. (e) Upon heating, the double-stranded DNA breaks apart and we get two complementary single strands. In effect, y has been inserted into x_1uvx_2z forming x_1uyvx_2z.: The picture is a modified version of the one from [46].

cut exactly the upper strand x_1uvx_2z), we get to the configuration (c) in Figure 2.2. Now we use $\theta(z)$ as primer and then the enzyme DNA polymerase can create the complement of the rest of the strand and result in the formation of $\theta(z)\theta(x_2)\theta(v)\theta(y)\theta(u)\theta(x_1)$. Analogously its complement x_1uyvx_2z is also created as shown in (d) of Figure 2.2. Then we heat the solution and the double strand breaks into its constituent single strands. We then can separate the single strand x_1uyvx_2z and this is effectively the string obtained by the insertion of y into x_1uvx_2z between u and v. This shows how string insertion can be achieved biochemically. Analogously, one can also show how the operation of deletion of strings can be achieved biochemically. An alternative way to simulate insertion and deletion operations biochemically could be to use site-directed mutagenesis [20], a process that is used to create targeted, specific changes in double-stranded plasmid DNA. Other than insertions and deletions, the process can also make specific DNA alterations such as substitutions. It is used for investigating the structure and biological activity of DNA, RNA and protein molecules.

The formal model of Insertion-Deletion (InsDel) systems is based on the operations of insertion and deletion. Various types of InsDel systems have been defined in the literature and their respective properties have also been widely studied. A problem that has been pursued with interest is the universal computability of an InsDel system, and finding the smallest size of an InsDel system that can produce any given recursively enumerable language.

2.4.2 Definition of insertion-deletion systems

Formally, an Insertion-Deletion (InsDel) system [22] is a construct

$$\gamma = (V, T, A, I, D)$$

where V is an alphabet, $T \subseteq V$, A is a finite subset of V^*, and I, D are finite subsets of $V^* \times V^* \times V^*$.

For $x, y \in V^*$, $x \Rightarrow y$ iff one of the following two cases holds:

(i) $x = x_1uvx_2, y = x_1uzvx_2$, for some $x_1, x_2 \in V^*$ and $(u, z, v) \in I$ (an insertion step);

(ii) $x = x_1uzvx_2, y = x_1uvx_2$, for some $x_1, x_2 \in V^*$ and $(u, z, v) \in D$ (a deletion step).

The language generated by γ is:

$$L(\gamma) = \{w \in T^* \mid x \Rightarrow^* w, \text{ for some } x \in A\}.$$

The complexity of an insertion-deletion system is often characterized by the maximum lengths of the words in the insertion and deletion rules. Such maximum lengths of the words are termed as weights of the InsDel system. Formally, an InsDel system $\gamma = (V,T,A,I,D)$ is said to be of *weight* (n,m,p,q) if

- $\max\{|z| \mid (u,z,v) \in I\} = n$
- $\max\{|u| \mid (u,z,v) \in I$ or $(v,z,u) \in I\} = m$
- $\max\{|z| \mid (u,z,v) \in D\} = p$
- $\max\{|u| \mid (u,z,v) \in D$ or $(v,z,u) \in D\} = q$

By $INS_n^m DEL_p^q$, we denote the family of languages that can be generated by any InsDel system respecting the weight limits of m,n,p,q. When one of the weights is unbounded, we replace it with infinity (∞).

2.4.3 Closure properties of InsDel systems

A fundamental property of insertion-deletion systems is that $INS_n^m DEL_p^q \subseteq INS_{n'}^{m'} DEL_{p'}^{q'}$, for all $0 \leq n \leq n'$, $0 \leq m \leq m'$, $0 \leq p \leq p'$ and $0 \leq q \leq q'$.

It was shown that $RE = INS_\infty^\infty DEL_\infty^\infty$ [25, 39]. The authors in [25] showed that an InsDel system in the family $INS_3^6 DEL_2^7$ is Turing universal while the authors in [39] had given a construction of an InsDel system in the family $INS_3^2 DEL_3^0$ that is Turing universal. A natural question was to find the minimal complexity (i.e. minimum values of m,n,p,q) of an insertion-deletion system that generates any recursively enumerable language. There have been several attempts to solve this problem that have been proposed in the literature. Kari et al. have given few low complexity values for insertion-deletion systems that can generate any RE language [22]. In particular, it was proved that the insertion deletion systems $INS_1^2 DEL_1^1$, $INS_2^1 DEL_2^0$ or $INS_1^2 DEL_2^0$ can generate all recursively enumerable languages [22]. Paun et al. improved the result by showing that any recursively enumerable language can be generated by a system of complexity $INS_1^1 DEL_2^0$ [46]. Takahara et al. further improved the result by proving that a system of complexity $INS_1^1 DEL_1^1$ can generate any RE language [52].

2.4.4 Insertion-only systems

An InsDel system where the deletion rules are completely absent is called an Insertion-Only system. A formal definition for an insertion-only system is given as follows:

Definition: A (pure) insertion grammar of weight $n \geq 0$ is a triple $G = (V, A, P)$, where

- V is a finite alphabet
- $A \subseteq V^*$ is a finite set of axioms
- P is a finite set of insertion rules of the form (u, x, v), for $u, x, v \in V^*$
- $n = max\{|u| | (u, x, v) \in P$ or $(v, x, u) \in P\}$

The families $INS_{\infty}^{m}DEL_{0}^{0}$ for all $m \geq 1$ are also exactly those generated by Galiukschov grammars defined in [9]. The following results have been arrived in [9], [42], [43], [51] :

- $FIN \subset INS_{\infty}^{0}DEL_{0}^{0} \subset INS_{\infty}^{1}DEL_{0}^{0} \subset ... \subset INS_{\infty}^{\infty}DEL_{0}^{0} \subset CS$,
- The set of regular languages is incomparable with all families $INS_{\infty}^{m}DEL_{0}^{0}$, for all $m \geq 0$, and $REG \subset INS_{\infty}^{\infty}DEL_{0}^{0}$,
- The family $INS_{\infty}^{1}DEL_{0}^{0}$ is within CF, but the class of context-free languages is incomparable with $INS_{\infty}^{m}DEL_{0}^{0}$ for all $m \geq 2$.

The problem of finding the minimal complexity for insertion-only grammars that can generate any recursively enumerable language has been widely studied, similar to the same problem for a general insertion-deletion system.

Martin-Vide et al. proved that any RE language can be written in the form of $L = g(h^{-1}(L'))$, where g is a weak coding, h is a morphism and $L' \in INS_{4}^{7}DEL_{0}^{0}$ [39]. Yong et al. improved the result by proving that any language $L \in RE$ can be written in the form $L = g(h^{-1}(L'))$ where g is a weak coding, h is a morphism and $L' \in INS_{2}^{4}DEL_{0}^{0}$ [54]. Paun et al. gave a further result by proving that any language $L \in RE$ can be represented in the form $L = h(L' \cap D)$, where $L' \in INS_{3}^{0}DEL_{0}^{0}$, where h is a projection, and D is a Dyck language [45]. Kari and

Sosik in [24] further improved the result by proving that for any recursively enumerable language L there exists a morphism h, a weak coding g and a language $L_1 \in INS_3^3DEL_0^0$ such that $L = g(h^{-1}(L_1))$.

2.4.5 Context-free insertion-deletion systems

Insertion-Deletion systems without any context controlling the insertion and deletion operations were first investigated in [38]. Contrary to expectation, the authors found that such a system can generate a particular RE language even with one axiom and can generate any RE language if it has two axioms.

A context-free insertion deletion system is an insertion-deletion system where the insertions and deletions can happen at any context. Formally, in a context-free insertion-deletion system all the insertion rules of the form $(\lambda, \alpha, \lambda) \in I$ and all the deletion rules of the form $(\lambda, \alpha, \lambda) \in D$, where λ denotes the empty string. $INS_\infty^0DEL_\infty^0$ denotes the family of languages generated by context-free InsDel systems.

Margenstern et al. proved that $INS_\infty^0DEL_\infty^0$ is universal [38]. The authors then improved the bound and arrived at Theorem 2.4.1.

Theorem 2.4.1. *RE = $INS_2^0DEL_3^0$ and RE = $INS_3^0DEL_2^0$.*

Verlan proved in [53], that $INS_2^0DEL_2^0 = INS_2^0DEL_0^0 \subset CF$ and that $INS_m^0DEL_1^0 = INS_m^0DEL_0^0 \subset CF$ for any $m > 0$. It was also established [53] that $INS_2^0DEL_2^0$ is incomparable with REG but that $INS_1^0DEL_p^0 \subset REG$ for any $p > 0$.

2.4.6 Circular insertion-deletion systems

Insertions and deletions of circular strands of DNA into or from long linear strands occur in biological systems. For example, this is the way in which some viruses insert themselves into the host DNA and thus infect them.

Daley, Kari, Gloor and Siromoney proposed a generalization of insertions and deletion of words to model these processes [7].

Circular contextual insertion

The operation of circular insertion consists of two distinct phases. In the first phase, a circular string is cut at some location to form a linear string. In the second phase, the newly generated linear string is inserted into a specific location on a previously existing linear string.

The authors [7] define circular contextual guided insertion of a circular string $\sim v$ into the linear string u as follows:

Definition A *circular insertion scheme* is a triple $I = (X, C, G)$ where X is an alphabet, $C \subseteq X^* \times X^*$ is a context set and $G \subseteq X^* \times X^*$ is a guide set.

A guide set is a set of locations on the circular string which indicate where it may be cut to produce a linear strand (that can be inserted). A context set is a set of locations (i.e. borders of locations) where an insertion can be made.

Definition Given two words u, $v \in X^*$, the *circular contextual guided insertion* of $\sim v$ in to u according to the circular insertion scheme I is defined as:

$$u \longleftarrow\!\sim v = \{u_1 x \alpha w \beta y u_2 \mid (x,y) \in C, (\alpha, \beta) \in G, u = u_1 x y u_2, \sim v = \sim \alpha w \beta, u_1, u_2, w \in X^*\}$$

Note that the circular contextual guided insertion of $\sim v$ into u with guide set (α, β) in a context (x, y) can result in multiple strings where all linearizations of $\sim v$ of the form $\alpha w \beta$ can be inserted into u between any substrings x and y of u such that $u = u_1 x y u_2$.

The authors [7] proved a series of closure properties resulting in Theorem 2.4.2.

Theorem 2.4.2. *The classes of regular, context-free, context-sensitive and recursively enumerable languages are closed under circular guided contextual insertion.*

Circular insertion-deletion system

The authors [7], state that, as rewriting mechanisms, the insertion-type rules alone are not sufficient to generate any RE language. In order to define systems that can achieve the computational power of any Turing machine, they combine the circular insertions with some (linear) deletions and define a circular insertion-deletion system. Formally, a circular insertion-deletion system [7] is a tuple

$$ID^{\blacklozenge} = (X, T, I^{\blacklozenge}, D, A)$$

where

- X is an alphabet (cardinality of X is at least 2),
- $T \subseteq X$ is the terminal alphabet,
- $I^{\blacklozenge} \subseteq X^* \times X^* \times \sim X^* \times X^* \times X^*$ is the finite set of circular insertion rules,
- $D \subseteq X^* \times X^* \times X^*$ is finite set of deletion rules, and
- $A \subseteq X^+$ is a linear strand called the axiom.

If u, $v \in X^*$, u derives v according to $ID^{\blacklozenge}$ and we write $u \Rightarrow v$ if v is obtained from u by either a guided contextual circular insertion or by a linear contextual deletion, that is, one of the following two cases happen:

- $u = \alpha c_1 c_2 \beta$, $v = \alpha c_1 | g_1 x' g_2 | c_2 \beta$ and $I^{\blacklozenge}$ contains the circular insertion rule $(c_1, g_1, \sim x, g_2, c_2)_I$ where $g_1 x' g_2 \in Lin(\sim x)$,
- $u = \alpha c_1 x c_2 \beta$, $v = \alpha c_1 c_2 \beta$ and D contains the linear deletion rule $(c_1, x, c_2)_D$.

The sequence of derivations $u_1 \Rightarrow u_2 \Rightarrow \ldots \Rightarrow u_k$, $k \geq 0$ is denoted by $u_1 \Rightarrow^* u_k$. The language $L(ID^{\blacklozenge})$ accepted by the circular insertion-deletion system $ID^{\blacklozenge}$ is

$$L(ID^{\blacklozenge}) = \{v \in T^* \mid A \Rightarrow^* v, A \text{ is the axiom}\}$$

It was proved that the circular insertion-deletion system is Turing universal through the following result in Theorem 2.4.3:

Theorem 2.4.3. *[7] If a language is accepted by a Turing machine TM, then there exists a circular insertion-deletion system $ID^{\blacklozenge}$ accepting the same language.*

2.5 Hairpin completion - reduction systems

2.5.1 Introduction

Hairpins are naturally occurring DNA secondary structures whereby a DNA strand attaches to itself due to the presence of a Watson-Crick complementary sequence(s) present elsewhere in the same strand. Hairpin formations by single-stranded DNA molecules are used in the field of DNA computing to explore the feasibility of building an autonomous molecular computer. For example, a solution of an instance of satisfiability problem using molecular biology techniques and involving hairpin formations was reported in [48].

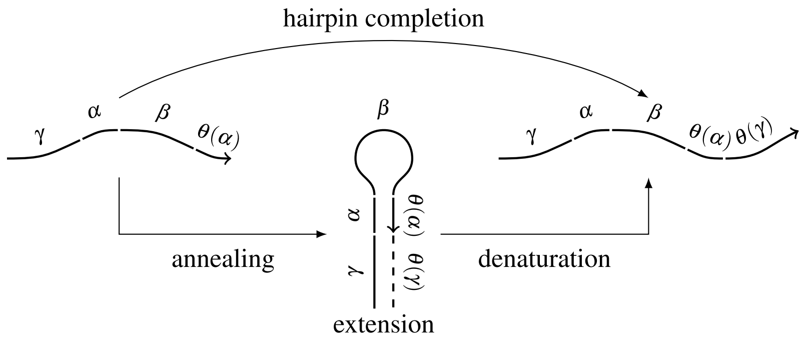

Figure 2.3: An illustration of the hairpin completion operation: In the first step of annealing, the input string $\gamma\alpha\beta\theta(\alpha)$ forms a hairpin structure with the segments α and $\theta(\alpha)$ attached with each other. In the second step of extension, the polymerase enzyme extends the hairpin structure to form a new DNA segment $\theta(\gamma)$, using γ as a template. In the final step of denaturation, the hairpin structure is broken (as all double strands break apart upon heating) and the strand $\gamma\alpha\beta\theta(\alpha)\theta(\gamma)$ is obtained as the output of the operation.

The hairpin completion as a bio-operation consists of three essential steps: annealing, extension of DNA strand and denaturation. These steps are quite similar to the steps discussed in Section 1.2.3 about Polymerase Chain Reaction. In the process of annealing, a part of DNA strand attaches to itself because of Watson-Crick complementary sequence present elsewhere in the strand. In the process of extension, a DNA polymerase enzyme extends the primer (which is the part that got annealed) by attaching complementary bases to the remaining part of the template. Finally the DNA hairpin becomes a single strand through the process of denaturation where the double-stranded parts of the strand are separated.

Figure 2.5.1 illustrates hairpin completion. We consider an example where we have a test tube with a solution containing a population of DNA strands $\gamma\alpha\beta\theta(\alpha)$. Through the process of annealing, the strands α and $\theta(\alpha)$ attach to each other due to their Watson-Crick complementarity. Then the polymerase enzyme acts and extends the string $\theta(\alpha)$ by the string $\theta(\gamma)$. Finally, the process of denaturation breaks the weak hydrogen bonds connecting the complementary bases and results in the single strand $\gamma\alpha\beta\theta(\alpha)\theta(\gamma)$.

Inspired by the above biochemical reaction, a formal operation called "hairpin completion" has been proposed by Mitrana et al [6].

2.5.2 Definition of hairpin completion

The hairpin completion can happen in both directions, i.e. to the left or to the right of the word. The result set of hairpin completion is the union of sets obtained by the hairpin completion to the left and to the right. Let A be an alphabet. For any $w \in V^+$ we define its k-hairpin completion, denoted by $HC_k(w)$, as follows:

$$\begin{aligned}
\text{(Left) } HCP_k(w) &= \{\theta(\gamma)w \mid w = \alpha\beta\theta(\alpha)\gamma, |\alpha| = k, \alpha, \beta \in A^+, \gamma \in A^*\}, \\
\text{(Right) } HCS_k(w) &= \{w\theta(\gamma) \mid w = \gamma\alpha\beta\theta(\alpha), |\alpha| = k, \alpha, \beta \in A^+, \gamma \in A^*\}, \\
\text{(Both) } HC_k(w) &= HCP_k(w) \cup HCS_k(w).
\end{aligned}$$

The hairpin completion of w is defined as:

$$HC(w) = \bigcup_{k \geq 1} HC_k(w).$$

The hairpin completion is naturally extended to languages and is defined as:

$$HC_k(L) = \bigcup_{w \in L} HC_k(w).$$

2.5.3 Iterated hairpin completions

The iterated hairpin completion is the result of applying the operation of hairpin completion repeatedly, which may produce new strings since there can be new sub-words in the outputs which can be involved in hairpin formations. The operation of iterated hairpin completion can also be seen as naturally inspired because, after hairpin completion, the newer strand is longer and potentially there is a chance that it contains new parts that are Watson-Crick complementary to each other. Thus, another step of hairpin completion may continue and produce newer outputs and this cycle may continue indefinitely. The situation is described in Figure 2.4.

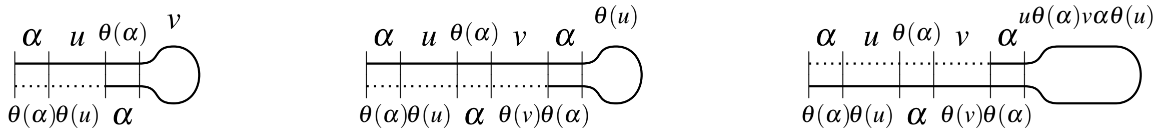
α u $\theta(\alpha)$ v
$\theta(\alpha)\theta(u)$ α
α u $\theta(\alpha)$ v α $\theta(u)$
$\theta(\alpha)\theta(u)$ α $\theta(v)\theta(\alpha)$
α u $\theta(\alpha)$ v α $u\theta(\alpha)v\alpha\theta(u)$
$\theta(\alpha)\theta(u)$ α $\theta(v)\theta(\alpha)$

Figure 2.4: Example of iterated hairpin completion. Given an initial strand $\alpha u\theta(\alpha)v\alpha$, the hairpin completion will give the strand $\alpha u\theta(\alpha)v\alpha\theta(u)\theta(\alpha)$. The new strand can also undergo hairpin completion, as it contains $\theta(\alpha)$ that can anneal to α and be extended further. Thus another application of hairpin completion will happen giving rise to the new strand $\alpha u\theta(\alpha)v\alpha\theta(u)\theta(\alpha)\theta(v)\alpha\theta(u)\theta(\alpha)$. This strand again can undergo yet another hairpin completion. Thus the process continues until no further strands are produced through another hairpin completion.

The iterated version of the hairpin completion is defined as:

$$HC_k^0(w) = \{w\}, \quad HC_k^{n+1}(w) = HC_k(HC_k^n(w)), \quad HC_k^*(w) = \bigcup_{n \geq 0} HC_k^n(w),$$
$$HC^0(w) = \{w\}, \quad HC^{n+1}(w) = HC(HC^n(w)), \quad HC^*(w) = \bigcup_{n \geq 0} HC^n(w).$$

The iterated hairpin completion is extended to languages as:

$$HC_k^*(L) = \bigcup_{w \in L} HC_k^*(w) \quad \text{and} \quad HC^*(L) = \bigcup_{w \in L} HC^*(w).$$

Now, we will briefly describe some algorithmic problems based on hairpin completion and its iterated version.

Hairpin completion distance and common ancestors

The hairpin completion distance between two words x and y is defined as the minimal number of hairpin completions which can be applied either to x in order to obtain y, or to y in order to obtain x. If none of them can be obtained from each other, the distance is said to be infinity and is denoted as ∞.

Formally, the k-hairpin completion distance between x and y is given as:

$$HCD_k(x,y) = \begin{cases} min\{p \mid x \in HC_k^p(y) \text{ or } y \in HC_k^p(x)\}, \\ \infty, \text{ if neither } x \in HC_k^*(y) \text{ nor } y \in HC_k^*(x). \end{cases}$$

The problem of finding a word at a certain k-hairpin completion distance from a given initial word was studied by Manea et al [30]. They established that, given a word x, and $k,n \in \mathbb{N}$, the problem of whether there exists a y_n such that $HCD_k(x,y_n) = n$ is decidable in $O(|x|)$ time. They further gave an algorithm to compute the k-hairpin completion distance between two words x and y in $O(n^3)$ time where n is the length of the longest word among x and y. Later, Manea improved it by giving an algorithm that runs in $O(n^2 \log n)$ time and using $O(n^2)$ space, where the longest word has length n [29].

Definition A word w is called a k-hairpin completion ancestor of two words x and y if $\{x,y\} \subseteq HC_k^*(w)$. It is called the minimum-distance common ancestor if it further satisfies the property that $HCD_k(w,x) + HCD_k(w,y) \leq HCD_k(w',x) + HCD_k(w',y)$, for all w' such that $\{x,y\} \subseteq HC_k^*(w')$. It is called as the maximum-distance common ancestor if it further satisfies the property that $HCD_k(w,x) + HCD_k(w,y) \geq HCD_k(w',x) + HCD_k(w',y)$, for all w' such that $\{x,y\} \subseteq HC_k^*(w')$.

Manea and Mitrana proved, [34], that the existence of a common k-hairpin completion ancestor for two given words x and y is decidable in $O(max(|x|,|y|)^3)$ time for any $k \geq 1$. They gave an algorithm that finds such an ancestor in cubic time, if it exists.

Manea improved this result and gave a quadratic time algorithm for finding an arbitrary common hairpin completion ancestor of two words [29] stating that given two words x and w, and an integer k, a common k-hairpin completion ancestor of the words x and w in $O(max(|x|,|w|)^2)$ time and space.

Manea also gave an efficient algorithm that can compute the minimum-distance common hairpin completion ancestor [29]. He stated that given two words x and w, and an integer k, his algorithm can compute the minimum-distance common k-hairpin completion ancestor of the words x and w in time $O(max(|x|,|w|)^2 log_2(max(|x|,|w|)))$, using $O(max(|x|,|w|)^2)$ space. Manea explained how the results extend to maximum-distance common k-hairpin completion ancestor and stated that given two words x and w, and an integer k, the maximum-distance common k-hairpin completion ancestor of the words x and w can be found in time $O(max(|x|,|w|)^2 log_2(max(|x|,|w|)))$, using $O(max(|x|,|w|)^2)$ space. He further proved that all the maximum-distance common k-hairpin ancestors of the words x and w can be identified in $O(max(|x|,|w|)^2)$ time and space [29].

2.5.4 Definition of hairpin reduction

Hairpin reduction is the formal operation defined as the mathematical inverse of hairpin formation. As such, it does not have a direct analogy in DNA biochemistry. The intuitive idea behind the operation is to see whether, given a string, it is possible to determine if it is produced by hairpin completion of another string and if so, determine it. The words obtained as the result of hairpin reduction of a string are those whose hairpin completion will produce the given string.

Let V be an alphabet and $k \geq 1$. For any $w \in V^+$ we define the k-hairpin reduction of w, denoted by $HR_k(w)$, as follows:

$$\begin{aligned}
\text{(Left) } HRP_k(w) &= \{\alpha\beta\theta(\alpha)\theta(\gamma) \mid w = \gamma\alpha\beta\theta(\alpha)\theta(\gamma), |\alpha| = k, \alpha,\beta,\gamma \in V^+\}, \\
\text{(Right) } HRS_k(w) &= \{\gamma\alpha\beta\theta(\alpha) \mid w = \gamma\alpha\beta\theta(\alpha)\theta(\gamma), |\alpha| = k, \alpha,\beta,\gamma \in V^+\}, \\
\text{(Both) } HR_k(w) &= HRP_k(w) \cup HRS_k(w).
\end{aligned}$$

The hairpin reduction of w is defined by

$$HR(w) = \bigcup_{k \geq 1} HR_k(w)$$

The hairpin reduction is extended to languages as:

$$HR_k(L) = \bigcup_{w \in L} HR_k(w) \quad \text{and} \quad HR(L) = \bigcup_{w \in L} HR(w).$$

The iterated version of hairpin reduction is defined as:

$$\begin{aligned}
&HR_k^0(w) = \{w\}, \quad HR_k^{n+1}(w) = HR_k(HR_k^n(w)), \quad HR_k^*(w) = \bigcup_{n \geq 0} HR_k^n(w), \\
&HR^0(w) = \{w\}, \quad HR^{n+1}(w) = HR(HR^n(w)), \quad HR^*(w) = \bigcup_{n \geq 0} HR^n(w).
\end{aligned}$$

The iterated hairpin reduction is extended to languages as:

$$HR_k^*(L) = \bigcup_{w \in L} HR_k^*(w) \quad \text{and} \quad HR^*(L) = \bigcup_{w \in L} HR^*(w).$$

Now, we will look into the closure properties and complexity results of hairpin completion, hairpin reduction and iterated hairpin completion.

2.5.5 Closure properties for hairpin completions and reductions

Before we give the properties, we give definition of space constructible functions.

Definition [50] A function $f : \mathbb{N} \to \mathbb{N}$ is space constructible if the function that maps the string 1^n to the binary representation of $f(n)$ is computable in space $O(f(n))$.

Hairpin completions

Cheptea et al. stated that the classes of regular and context-free languages are not closed under hairpin completion [6]. They further proved that the hairpin completion of a regular language is always linear and that the hairpin completion of a context-free language is always context-sensitive [6]. For space complexity classes, it is proved that if $f(n) \geq \log n$ is any space-constructible function, the class NSPACE$(f(n))$ is closed under hairpin completion [6]. That implies that the class of languages accepted by a non-deterministic Turing machine using $f(n)$ workspace on input words of length n, for any $f(n) \geq logn$, is closed under hairpin completion. Since we know that any language in CS is accepted by a Turing machine using linear space (which is greater than logarithmic), i.e., CS is in $NSPACE(f(n))$ for $f(n) \geq \log n$, we conclude that the class of context-sensitive languages is closed under the hairpin completion.

For time complexity classes, it was proved that if L is in NTIME$(f(n))$, then $HC_k(L) \in$ NTIME$(nf(n))$ and if $L \in$ DTIME$(f(n))$, then $HC_k(L) \in$ DTIME$(nf(n))$ for all $k \geq 1$, [6]. This led to the fact that both classes PTIME and NPTIME are closed under the hairpin completion operation.

Iterated hairpin completions

If $f(n) \geq \log n$ is a space-constructible function, the class NSPACE$(f(n))$ was proved to be closed under iterated hairpin completion, [6]. By an argument similar to that of non-iterated hairpin completion, one can see that the class of context-sensitive languages is closed under iterated hairpin completion. It was proved that, for any $k \geq 1$, the iterated k-hairpin completion of a regular language is not even necessarily a context-free language, [6]. Further, it was established that if L is a regular language, then $HC_k^*(L)$ will always be in PTIME for any $k \geq 1$, [6].

The problem of determining the class of languages obtained by the iterated hairpin completion of singleton languages was pursued with interest. In [36], the authors state the problem of determining if the class is contained within REG or CF as open. Kopecki answered this by giving an example of a singleton language whose iterated hairpin completion is not context-free and further proved that the result of the iterated hairpin completion of any singleton language is in the class $NSPACE(\log n)$ [27]. The result also proved that the iterated hairpin completion of either of the classes of regular and context-free languages is not within CF. Kari et al. resolved the problem of deciding if the iterated hairpin completion of a singleton is in REG and gave necessary and sufficient conditions for the same [23].

Hairpin reduction

It was proved that the class of regular languages is closed under k-hairpin reduction for any $k \geq 1$, but the class of context-free languages is not closed under the same [34]. Later, it was proved that the class of linear languages is not closed under hairpin reduction [35]. But, it has been proven that for every $k \geq 1$, if L is recognizable in $O(f(n))$ time, then $HR_k(L)$ is recognizable in $O(nf(n))$ time [34, 35]. This led to the conclusion that the classes PTIME and NPTIME are closed under hairpin reduction. For space complexity classes, Manea et al. proved in [34] that if $f(n) \geq \log n$ be a space-constructible function such that $f(n+n/2) \in$

$O(f(n))$, then NSPACE$(f(n))$ and DSPACE$(f(n))$ are closed under k-hairpin reduction for any $k \geq 1$. The result was extended in [35] to include all space constructible functions $f \geq \log n$ satisfying the condition $f(2n) \in O(f(n))$. It is clear that $f(n) = cn$ for some $c > 0$ satisfies these conditions and hence it follows that the class of context-sensitive languages is closed under hairpin reduction.

It is obvious that the class of recursively enumerable languages is closed under (non-iterated and iterated) hairpin completion and hairpin reduction. Here, we summarize all the important results stated thus far about the closure of various language classes under these operations in Table 2.2.

Operation\L	**REG**	**CF**	**CS**	**RE**
Hairpin Completion	LIN	CS	CS	RE
Iterated HpC	Not CF	Not CF	CS	RE
Hairpin Reduction	REG	Not CF	CS	RE

Table 2.2: Closure properties under hairpin operations [6, 23, 27, 34–36]

2.5.6 Hairpin lengthening

The formal operation of hairpin completion is directly inspired by the molecular process of hairpin formation in DNA bio-chemistry. However, in order to model hairpin formations more closely to the biological reality, a variant of hairpin completion called hairpin lengthening, was proposed by Manea et al. in [31]. This is because, in most practical scenarios, the hairpin loop may not be extended fully by the DNA polymerase enzyme. Thus, the hairpin lengthening is different from hairpin completion because the extension may not be complete.

Let A be an alphabet. For any $w \in A^+$ and $k \geq 1$, the k-hairpin lengthening of w, denoted by $HL_k(w)$, is defined as follows:

$$\begin{aligned}
\text{(Left) } HLP_k(w) &= \{\theta(\delta)w \mid w = \alpha\beta\theta(\alpha)\gamma, |\alpha| = k, \alpha,\beta,\gamma \in V^+, \delta \in \text{pref}(\gamma)\},\\
\text{(Right) } HLS_k(w) &= \{w\theta(\delta) \mid w = \gamma\alpha\beta\theta(\alpha), |\alpha| = k, \alpha,\beta,\gamma \in V^+, \delta \in \text{suff}(\gamma)\},\\
\text{(Both) } HL_k(w) &= HLP_k(w) \cup HLS_k(w).
\end{aligned}$$

Similar to hairpin completion, the lengthening can happen at the left or at the right of the word. Hence, the result of the operation is the union of the words formed through lengthening of the initial word in either direction.

The hairpin lengthening of w is defined by $HL(w) = \bigcup_{k\geq 1} HL_k(w)$. The hairpin lengthening is naturally extended to languages as:

$$HL_k(L) = \bigcup_{w\in L} HL_k(w).$$

The iterated version of hairpin lengthening for a word w and a language L is defined as:

$$HL_k^0(w) = \{w\}, \qquad HL_k^{n+1}(w) = HL_k(HL_k^n(w)),$$

$$HL_k^*(w) = \bigcup_{n\geq 0} HL_k^n(w), \qquad HL_k^*(L) = \bigcup_{w\in L} HL_k^*(w).$$

Closure properties and complexity results

Manea et al. proved that the class of regular languages is not closed under k-hairpin lengthening for $k \geq 1$ [33]. Later, it was proved that the classes of linear and context-free languages are not closed under k-hairpin lengthening for $k \geq 1$ [32]. On the other hand, it was proved that the class of regular languages is closed under iterated k-hairpin lengthening for $k \geq 1$ [33] and later, it was proved that the classes of linear and context-free languages are closed under iterated k-hairpin lengthening for $k \geq 1$ [32].

The operation of hairpin lengthening has also been studied from the complexity point of view. It was proved that, for every $k \geq 1$, and every language L recognizable in $O(f(n))$ time, the k-hairpin lengthening of L is recognizable in $O(nf(n))$ time, and the iterated k-hairpin lengthening of L is recognizable in $O(n^2 f(n))$ time, [31]. Further, it was proved that if L is regular, $HL_k(L)$ is recognizable in $O(n)$ time, and if L is context-free, $HL_k(L)$ is recognizable in $O(n^3)$ time, [31]. Also it was proved that, for every $k \geq 1$ and every linear language L, the iterated k-hairpin lengthening of L is recognizable in quadratic time, and for every context-free language L the iterated k-hairpin lengthening of L is recognizable in cubic time [32].

Another related complexity measure called the k-hairpin lengthening distance was defined between any two words [31]. Informally, it is the number of times that the operation of hairpin lengthening has to be applied to one of the strings to obtain the other. If it is impossible to obtain one from the other, the distance is taken to be infinity and noted as (∞). Formally, the k-hairpin lengthening distance between x and y, denoted by $HLD_k(x,y)$, is defined by:

$$HLD_k(x,y) = \begin{cases} min\{p \mid x \in HL_k^p(y) \text{ or } y \in HL_k^p(x)\}, \\ \infty, \text{ if neither } x \in HL_k^p(y) \text{ nor } y \in HL_k^p(x). \end{cases}$$

Manea et al. proved that the k-hairpin lengthening distance between two words x and w can be computed in $O(max(|x|,|w|)^2)$ time, [31].

2.6 Conclusions

In this chapter, we have seen a review of three major formal systems based on operations inspired from DNA processes mediated by enzymes. Further work can include refinement of models to capture the biological reality more closely. Another future direction is to model some other enzymatic actions, for, e.g., restriction enzymes that cut away from their recognition sites as they can create wide variety of interesting newer sequences.

Bibliography

[1] J. Berstel, L. Boasson, and I. Fagnot. Splicing systems and the Chomsky hierarchy. *Theoretical Computer Science*, 436:2–22, 2012.

[2] P. Bonizzoni, C. D. Felice, G. Mauri, and R. Zizza. Circular splicing and regularity. *Theoretical Informatics and Applications*, 38:189–228, 2004.

[3] P. Bonizzoni, C. D. Felice, G. Mauri, and R. Zizza. On the power of circular splicing. *Discrete Applied Mathematics*, 150:51–66, 2005.

[4] P. Bonizzoni, C. D. Felice, and R. Zizza. A characterization of (regular) circular languages generated by monotone complete splicing systems. *Theoretical Computer Science*, 411:4149–4161, 2010.

[5] P. Bonizzoni, C. Ferretti, G. Mauri, and R. Zizza. Separating some splicing models. *Information Processing Letters*, 79(6):255–259, 2001.

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

[7] M. Daley, L. Kari, G. Gloor, and R. Siromoney. Circular contextual insertions/deletions with applications to biomolecular computation. In Proc. *String Processing and Information Retrieval,* SPIRE, pages 47–54, 1999.

[8] W. Ebeling and M. A. Jiménez-Montaño. On grammars, complexity, and information measures of biological macromolecules. *Mathematical Biosciences*, 52(12):53–71, 1980.

[9] B. Galiukschov. Semicontextual grammars. *Mat. logica i mat. lingv., Kalinin Univ.*, pages 38–51, 1981.

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

[11] E. Goode and D. Pixton. Semi-simple splicing systems. In C. Martín-Vide and V. Mitrana, editors, *Where Mathematics, Computer Science, Linguistics and Biology Meet*, pages 343–352. Springer Netherlands, 2001.

[12] D. Haussler. Insertion languages. *Information Sciences*, 31(1):77–89, 1983.

[13] 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.

[14] T. Head. Splicing schemes and DNA. In G. Rozenberg and A. Salomaa, editors, *Lindenmayer Systems: Impacts on Theoretical Computer Science and Developmental Biology*, 1992.

[15] T. Head. Splicing languages generated with one sided context. In G. Păun, editor, *Computing with Bio-molecules: Theory and Experiments*, 1998.

[16] T. Head, D. Pixton, and E. Goode. Splicing systems: regularity and below. In M. Hagiya and A. Ohuchi, editors, *DNA Based Computers: DNA Computing,* DNA 8, volume 2568 of *LNCS*, pages 262–268, 2003.

[17] T. Head, G. Păun, and D. Pixton. Language theory and molecular genetics: Generative mechanisms suggested by DNA recombination. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, pages 295–360, 1997.

[18] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Inc., 1978.

[19] K. C. II and T. Harju. Splicing semigroups of dominoes and DNA. *Discrete Applied Mathematics*, 31(3):261–277, 1991.

[20] N. E. B. Inc. Site-directed mutagenesis. `https://www.neb.com/applications/cloning-and-synthetic-biology/site-directed-mutagenesis/`.

[21] 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 (DNA 18)*, volume 7433 of *LNCS*, pages 98–109, 2012.

[22] 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*, pages 329–346, 1999.

[23] L. Kari, S. Seki, and S. Kopecki. On the regularity of iterated hairpin completion of a single word. *Fundamenta Informaticae*, 110(1-4):201–215, 2011.

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

[25] L. Kari and G. Thierrin. Contextual insertions/deletions and computability. *Information and Computation*, 131(1):47–61, 1996.

[26] S. M. Kim. An algorithm for identifying spliced languages. In T. Jiang and D. Lee, editors, Proc. *Computing and Combinatorics Conference,* COCOON, volume 1276 of *LNCS*, pages 403–411, 1997.

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

[28] E. G. Laun. *Constants and Splicing Systems*. PhD thesis, State University of New York at Binghamton, 1999.

[29] F. Manea. A series of algorithmic results related to the iterated hairpin completion. *Theoretical Computer Science*, 411(48):4162–4178, 2010.

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

[31] F. Manea, C. Martín-Vide, and V. Mitrana. Hairpin lengthening. In F. Ferreira, B. Löwe, E. Mayordomo, and L. M. Gomes, editors, *Programs, Proofs, Processes*, volume 6158 of *LNCS*, pages 296–306, 2010.

[32] F. Manea, C. Martín-Vide, and V. Mitrana. Hairpin lengthening: language theoretic and algorithmic results. *Journal of Logic and Computation*, 25(4):987–1009, 2015.

[33] F. Manea, R. Mercas, and V. Mitrana. Hairpin lengthening and shortening of regular languages. In H. Bordihn, M. Kutrib, and B. Truthe, editors, *Languages Alive*, volume 7300 of *LNCS*, pages 145–159, 2012.

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

[35] F. Manea, V. Mitrana, and T. Yokomori. Two complementary operations inspired by the DNA hairpin formation: completion and reduction. *Theoretical Computer Science*, 410(45):417–425, 2009.

[36] F. Manea, V. Mitrana, and T. Yokomori. Some remarks on the hairpin completion. *International Journal of Foundations of Computer Science*, 21(5):859–872, 2010.

[37] S. Marcus. *Algebraic Linguistics: Analytical Models*. Mathematics in Science and Engineering. Academic Press, 1967.

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

[39] C. Martín-Vide, G. Păun, and A. Salomaa. Characterizations of recursively enumerable languages by means of insertion grammars. *Theoretical Computer Science*, 205(1-2):195–205, 1998.

[40] A. Mateescu, G. Păun, G. Rozenberg, and A. Salomaa. Simple splicing systems. *Discrete Applied Mathematics*, 84(13):145–163, 1998.

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

[42] G. Păun. On semicontextual grammars. *Bull. Math. Soc. Sci. Math. Roumanie*, 28(76):63–68, 1984.

[43] G. Păun. Two theorems about Galiukschov semicontextual languages. *Kybernetika*, 21(5):360–365, 1985.

[44] G. Păun. On the splicing operation. *Discrete Applied Mathematics*, 70(1):57–79, 1996.

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

[46] G. Păun, G. Rozenberg, and A. Salomaa. *DNA Computing: New Computing Paradigms*. Springer-Verlag New York, Inc., 1998.

[47] B. Rigat, C. Hubert, F. Alhenc-Gelas, F. Cambien, P. Corvol, and F. Soubrier. An insertion/deletion polymorphism in the angiotensin I-converting enzyme gene accounting for half the variance of serum enzyme levels. *Journal of Clinical Investigation*, 86(4):13431346, 1990.

[48] K. Sakamoto, H. Gouzu, K. Komiya, D. Kiga, S. Yokoyama, T. Yokomori, and M. Hagiya. Molecular computation by DNA hairpin formation. *Science*, 288(5469):1223–1226, 2000.

[49] N. H. Sarmin, Y. Yusof, and F. Wan Heng. Some characterizations in splicing systems. In C. Ozel and A. Kilicman, editors, Proc. *International Conference on Mathematical Science,* ICMS, volume 1309 of *American Institute of Physics Conference Series*, pages 411–418, 2010.

[50] M. Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1996.

[51] C. C. Squier. Semicontextual grammars: an example. *Bull. Math. Soc. Sci. Math. Roumanie*, 32(80):167–170, 1988.

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

[53] S. Verlan. On minimal context-free insertion-deletion systems. *Journal of Automata, Languages and Combinatorics*, 12(1-2):317–328, 2007.

[54] 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.

[55] Y. Yusof, N. H. Sarmin, T. E. Goode, M. Mahmud, and F. W. Heng. Hierarchy of certain types of DNA splicing systems. *International Journal of Modern Physics: Conference Series*, 9:271–277, 2012.

[56] R. Zizza. Splicing systems. *Scholarpedia*, 5(7):9397, 2010. revision #137071.

Chapter 3

A formal language model of DNA polymerase enzymatic activity[1]

3.1 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 of biotechnology, are of particular interest. In this paper we propose and investigate a formal language operation that models the activity of DNA polymerase enzymes, enzymes that play 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., [9,11,15,19,29]. *Insertion-deletion* operations are basic to DNA processing and RNA editing in molecular biology. Insertion-

[1]Reprinted from Fundamenta Informaticae, 138(1-2), S.K. Enaganti, L. Kari, S. Kopecki, A formal language model of DNA polymerase enzymatic activity, 179-192, Copyright (2015), with permission from IOS press

Deletion systems were defined as formal models of computation based on these operations and have been widely studied in the literature, see, e.g., [5, 17, 18, 30, 31, 33, 34]. Insertion-deletion systems that are context-free [27], that have one sided-context [23, 28], and that are graph controlled [6] were also proposed. P-systems with insertion-deletion rules have been extensively studied in [1, 2, 7, 8, 22, 24]. 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, 21, 25, 26].

In this paper we define and investigate a formal language operation that models the extension activity of DNA Polymerase enzymes 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 3.1 (a)), and of a second short DNA strand called *primer*, that is Watson-Crick complementary to the template (Figure 3.1 (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 3.1 (c)). The newly formed DNA strand is a strand that starts with the primer and is partially Watson-Crick complementary to the template (Figure 3.1 (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*.

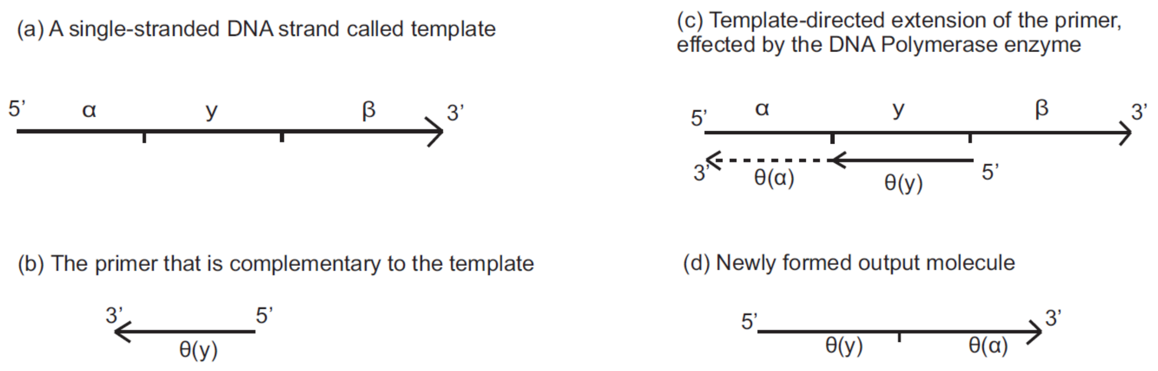
(a) A single-stranded DNA strand called template
5'
α
y
β
3'
(c) Template-directed extension of the primer, effected by the DNA Polymerase enzyme
5'
α
y
β
3'
3'
θ(α)
θ(y)
5'
(b) The primer that is complementary to the template
3'
5'
θ(y)
(d) Newly formed output molecule
5'
3'
θ(y)
θ(α)

Figure 3.1: 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.

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 3.2 contains definitions and notations, including the definition of directed extension. In Section 3.3, 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 3.4, we define an inverse of directed extension and study language equations involving this operation. In Section 3.5, we compare our operation with related string operations, and we discuss iterated versions of directed extension.

3.2 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 x, y, and z *prefix*, *infix*, and *suffix* of z, respectively. The sets $\text{Pref}(w)$, $\text{Inf}(w)$, and $\text{Suff}(w)$ contain, respectively, all proper prefixes, infixes, and suffixes of w. This notation is extended to languages as follows: $\mathrm{Suff}(L) = \bigcup_{w \in L} \mathrm{Suff}(w)$. 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 activity of DNA polymerase in Figure 3.1, 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, the primer needs to be of some minimal length and 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:

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

3.3 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 3.3.1. *If L_x and L_y are two languages over Σ, then $L_x \oplus L_y = \mathrm{Suff}(L_x) \cap L_y\Sigma^*$.*

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 $w \in \mathrm{Suff}(L_x)$.

Conversely, let $w \in \mathrm{Suff}(L_x) \cap L_y\Sigma^*$. Because $w \in \mathrm{Suff}(L_x)$, 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, $w \in L_x \oplus L_y$. $\square$

Corollary 3.3.2. *Let $\mathscr{X}$ and $\mathscr{Y}$ be two language classes where $\mathscr{X}$ is closed under the suffix operator and $\mathscr{Y}$ is closed under concatenation with Σ^*.*

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

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

In particular, REG and RE are closed under directed extension and, if $\mathscr{X}$ is LIN (CF) and $\mathscr{Y}$ is REG, then the result $L_x \oplus L_y$ is in LIN (CF). Similarly, if $\mathscr{X}$ is REG and $\mathscr{Y}$ is LIN (CF), 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 3.3.3. *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 3.3.3 allows us to classify the result of directed extension between two (linear) context-free languages.

Theorem 3.3.4. *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 3.3.3, the L_x-directed extension of L_y yields the context-sensitive but not context-free language

$$L_x \oplus L_y = \{\$a^n b^n c^n\$ \mid n \geq 1\}. \tag{3.1}$$

In order to show that $L_x \oplus L_y$ is context-sensitive for $L_x \in \mathrm{CF}$ and $L_y \in \mathrm{CS}$, we use Lemma 3.3.1 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 $\mathscr{O}(\log n)$ 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 3.3.5. *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 $j = n$.

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

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
```

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. $\square$

Theorem 3.3.5 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 3.3.6. *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 3.3.7. *The language classes* NTIME, DTIME, NSPACE *and* DSPACE *(all of which include LOG) are not closed under directed extension. More precisely, if L_x is in* NTIME, DTIME, NSPACE, DSPACE *then the L_x-directed extension of a singleton language may not be decidable.*

In Table 3.1 we summarize the results from this section. For two language classes $\mathscr{X}$ and $\mathscr{Y}$, it shows the language class $\mathscr{Z}$ from the Chomsky hierarchy such that for all $L_x \in \mathscr{X}$ and $L_y \in \mathscr{Y}$ we have $L_x \oplus L_y \in \mathscr{Z}$. Note that if we consider two language classes $\mathscr{X}$, $\mathscr{Y}$ which both contain the free monoid Σ^* for any alphabet Σ, we will require that $\$L\$ = \$L\$ \cap \$\Sigma^*\$ \in \mathscr{Z}$ for all languages $L \in \mathscr{X}$ or $L \in \mathscr{Y}$ which are defined over Σ, due to Lemma 3.3.3. If we restrict ourselves to classes in the Chomsky hierarchy (or standard space/time complexity classes), this statement can be strengthened as $\mathscr{X} \cup \mathscr{Y} \subseteq \mathscr{Z}$. This shows that all entries in Table 3.2 can also be considered "lower bounds" for the language class $\mathscr{Z}$.

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.

$L_x \backslash L_y$	FIN or REG	CF	CS	RE
REG	REG (Cor. 3.3.2)	CF (Cor. 3.3.2)	CS (Cor. 3.3.2)	RE (Cor. 3.3.2)
CF	CF (Cor. 3.3.2)	CS (Thm. 3.3.4)		RE (Cor. 3.3.2)
CS	RE (Cor. 3.3.2 and Cor. 3.3.6)			
RE	RE (Cor. 3.3.2)			

Table 3.1: 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.

3.4 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

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

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 3.4.1. *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 3.3.1. 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 3.4.2. *If $L_x \oplus Y = L_0$ has the solution L_y, then $L_{min} = (L_y \backslash L_y\Sigma^+) \cap \mathrm{Inf}(L_x)$ 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 3.4.1 and 3.4.2, we infer that if the equation $L_x \oplus Y = L_0$ has a solution, then $L_{0,min} = (L_0 \backslash L_0\Sigma^+) \cap \mathrm{Inf}(L_x)$ is a minimal solution.

3.5 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 DNA polymerase enzymes. 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 3.2 summarizes the closure properties of the operations of directed extension, PA-matching and superposition.

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.

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

Table 3.2: Closure properties under the directed extension operation, $\oplus$, compared to the PA-matching and superposition operations.

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 $\nu^*_{L_y}(L) = \lim_{n\to\infty} \nu^n_{L_y}(L)$ where $\nu_{L_y}(L) = L \cup (L \oplus L_y)$, and

3. the *iterated* L_x*-directed extension of* L as $\xi^*_{L_x}(L) = \lim_{n\to\infty} \xi^n_{L_x}(L)$ 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 3.3.1 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.).

Bibliography

[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 Proc. *Transgressive Computing,* TC, pages 216–228, 2006.

[5] M. Daley, L. Kari, G. Gloor, and R. Siromoney. Circular contextual insertions/deletions with applications to biomolecular computation. In Proc. *String Processing and Information Retrieval,* SPIRE, 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, Proc. *Descriptional Complexity of Formal Systems,* DCFS, volume 31 of *EPTCS*, 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, Proc. *Unconventional Computation and Natural Computation,* UCNC, 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. *Int. J. 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, *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. Karhumäki, 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 (DNA 18)*, 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*, 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. M. Kim. An algorithm for identifying spliced languages. In T. Jiang and D. Lee, editors, Proc. *Computing and Combinatorics Conference,* COCOON, 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, Proc. *Complexity of Simple Programs,* CSP, volume 1 of *EPTCS*, 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, Proc. *Language and Automata Theory and Applications,* LATA, 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, Proc. *Computability in Europe,* CiE, 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, Proc. *Machines, Computations, and Universality,* MCU, 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. *Int. J. of Foundations of Computer Science*, 19(4):859–871, 2008.

[31] G. Păun, G. Rozenberg, and A. Salomaa. *DNA Computing: New Computing Paradigms*. Springer-Verlag New York, Inc., 1998.

[32] J. H. Reif. Parallel molecular computation. In Proc. *Symposium on Parallel Algorithms and Architectures,* SPAA, 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.

Chapter 4

On the overlap assembly of strings and languages[1]

4.1 Introduction

In this paper we investigate properties of a formal language operation that models the linear self-assembly of DNA strands which partially "overlap". This binary operation which, given input the strands *xy* and *yz* (where the overlap *y* is non-empty), produces the output *xyz*, was introduced in [7] where it was called "(self)-assembly" of strings and languages. To distinguish it from other types of DNA self-assembly, this operation is herein called *overlap assembly*. Experimentally, (parallel) overlap assembly of DNA strands under the action of DNA Polymerase enzymes was used for gene shuffling in, e.g., [47]. In the context of experimental DNA Computing, overlap assembly was used in, e.g., [8, 13, 25, 42] for the formation of combinatorial DNA or RNA libraries. This operation can also be viewed as modelling a special case of an experimental procedure called cross-pairing PCR, introduced in [15] and studied in, e.g., [14, 16, 17, 35].

Conceptually, the study of overlap assembly as a formal language operation is part of a

[1]A version of this chapter, including an abstract, has been submitted to the Natural Computing journal (S.K. Enaganti, O.H. Ibarra, L. Kari, S. Kopecki. On the overlap assembly of strings and languages.)

larger effort of formalizing DNA processes as computations, which dates back to 1987 when Tom Head proposed *splicing* as a formal language operation that models the recombination of DNA strands under the cut-and-paste action of restriction enzymes and ligases. Various types of splicing systems have been defined and their properties were studied in, e.g., [18, 19, 27, 31, 43]. Other bio-operations include *insertions* and *deletions* of strands, which are basic processes in RNA editing in molecular biology: based on these, *insertion-deletion systems* were defined as formal models of computation and have been widely studied, see, e.g., [9, 29, 30, 45, 46, 48, 49].

Another example of a bio-inspired operation is a type of *substitution* operation that models errors occurring in DNA-encoded information, and that was proposed in [28]. *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, see [5, 32, 36, 37]. In the context of studies of cellular computing, the operations of contextual intra- and inter-molecular recombinations were proposed in [26, 33], the operations of loop, direct-repeat excision (ld), hairpin, inverted-repeat excision/reinsertion (hi) and double loop, alternating direct-repeat excision-reinsertion (dlad) were proposed in [11, 44], and the template-guided recombination was introduced in [2], as models for gene assembly in ciliates. Lastly, in [12], a language operation called *directed extension* was proposed, that models the enzymatic activity of DNA Polymerase enzymes. The activity of DNA Polymerase presupposes the existence of a DNA single strand called *template*, and of a second short DNA strand called *primer*, that is Watson-Crick complementary to the template and binds to it. Given a supply of individual nucleotides, DNA polymerase then extends the primer, at one of its ends only, by adding individual nucleotides complementary to the template nucleotides, one by one, until the end of the template is reached. Experimentally, the iteration of this process is used to obtain an exponential replication of DNA strands, in a protocol called *Polymerase Chain Reaction (PCR)*.

Among operations related to overlap assembly we cite the *superposition* operation, which was studied in [3, 38]. Superposition extends DNA strands in both directions, assuming the existence of Okazaki fragments in the solution. Another related operation, called *overlapping concatenation* was introduced as part of a study of tissue P systems, [39], that was designed to solve the shortest common superstring problem efficiently [34]. The overlapping concatenation between two words returns the longer word if it contains the other word as an infix, and otherwise returns the shortest string which contains the first word as a prefix and the second word as a suffix. Lastly, an operation called *conditional concatenation* was introduced in [10]: the conditional concatenation of two words returns their concatenation only when among their substrings (scattered substrings, of various forms) one can find a pair in a given control set.

This paper, which is a theoretical analysis of overlap assembly as a formal language operation, is organized as follows. Section 4.2 contains definitions and notations, including the definition of overlap assembly. In Sections 4.3, 4.4 we prove closure properties of various language classes under overlap assembly and investigate related decision problems. In Section 4.5, we investigate the iterated overlap assembly and demonstrate that, in theory, it can be an effective tool to generate a DNA combinatorial library.

4.2 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 w, respectively. The sets $\mathrm{Pref}(w)$, $\mathrm{Inf}(w)$, and $\mathrm{Suff}(w)$ contain, respectively, all proper prefixes, infixes, and suffixes of w. By *proper*, we mean that the sets do not include the word w itself. This notation is extended to languages as follows: $\mathrm{Suff}(L) = \bigcup_{w \in L} \mathrm{Suff}(w)$. The complement of a language $L \subseteq \Sigma^*$ is $L^c = \Sigma^* \backslash L$.

An *involution* is a function $\theta : \Sigma^* \to \Sigma^*$ with the property that θ^2 is the identity. θ is called an *antimorphism* if $\theta(uv) = \theta(v)\theta(u)$. Traditionally, the Watson-Crick complementarity of DNA strands has been modeled as an antimorphic involution over the DNA alphabet $\Delta = \{A, C, G, T\}$.

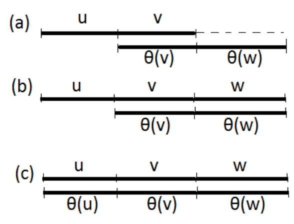

Figure 4.1: (a) The two input DNA single-strands, uv and $\theta(w)\theta(v)$ bind to each other through their complementary segments v and $\theta(v)$, forming a partially double-stranded DNA complex. (b) DNA Polymerase extends the 3' end of the strand uv. (c) DNA polymerase extends the 3' end of the other strand. The resulting DNA double strand is considered to be the output of the *overlap assembly* of the two input single strands.

Using the convention that a word x over this alphabet represents the DNA single strand x in the 5' to 3' direction, the overlap assembly of a strand uv with a strand $\theta(w)\theta(v)$ first forms a partially double-stranded DNA molecule with v in uv and $\theta(v)$ in $\theta(w)\theta(v)$ attached to each other, see Figure 1(a). DNA Polymerase enzyme will extend the 3' end of uv with the strand w, see Figure 1(b). Similarly, the 3' end of $\theta(w)\theta(v)$ will be extended, resulting in a full double strand whose upper strand is uvw, see Figure 1(c). Formally, the overlap assembly between uv and $\theta(w)\theta(v)$ is uvw. Assuming that all involved DNA strands are initially double-stranded, that is, whenever the strand x is available, its Watson-Crick complement $\theta(x)$ is also available, this model can be simplified as follows: Given two words x, y over an alphabet Σ, the *overlap assembly of x with y* is defined as, [7],

$$x \overline{\odot} y = \{z \in \Sigma^+ \mid \exists u, w \in \Sigma^*, \exists v \in \Sigma^+ : x = uv, y = vw; z = uvw\}$$

The definition of overlap assembly can be extended to languages in the natural way. Note that, for a realistic model, we would need additional restrictions such as the fact that the "overlap" v should be of a sufficient length for the Watson-Crick pairing to happen, and should also not appear as a substring in other strings involved. In this paper, however, we do not invoke any of these restrictions.

A similar operation, the *superposition*, has been proposed by Bottoni et al. [3]. The result of the superposition operation between words $x, y \in \Sigma^+$, denoted by $x \diamond y$, consists of the set of all words $z \in \Sigma^+$ obtained by any of the four following cases ($\bar{\ }$ denotes the *morphic complement*, i.e., $\bar{\ }$ is a mapping such that $\overline{uv} = \bar{u}\bar{v}$ and $\overline{\overline{u}} = u$ for all words u, v):

1. If there exist $u, v \in \Sigma^*, w \in \Sigma^+$ such that $x = uw, y = \overline{w}v$, then $z = uw\bar{v} \in x \diamond_1 y$.

2. If there exist $u, v \in \Sigma^*$ such that $x = u\bar{y}v$, then $z = u\bar{y}v \in x \diamond_2 y$.

3. If there exist $u, v \in \Sigma^*, w \in \Sigma^+$ such that $x = wv, y = u\overline{w}$, then $z = \bar{u}wv \in x \diamond_3 y$.

4. If there exist $u, v \in \Sigma^*$ such that $y = u\bar{x}v$, then $z = \bar{u}x\bar{v} \in x \diamond_4 y$.

As before, the superposition is naturally extended to languages. The superposition operation and the overlap assembly are closely related. In particular, when we replace the complement $\bar{\ }$ by the identity, then case 1 is identical to the overlap assembly $x \mathbin{\overline{\odot}} y = x \diamond_1 y$; case 3 is symmetrical to the overlap assembly $x \mathbin{\overline{\odot}} y = y \diamond_3 x$; furthermore, cases 2 and 4 give $x \diamond_2 y = y \diamond_4 x = x$ if y is an infix of x. From this observation, it easily follows that when we consider the overlap assembly of one language L by itself, we have $L \mathbin{\overline{\odot}} L = L \diamond L$. However, in the general case of two languages or when we consider a "real" complement function, the overlap assembly $L_x \mathbin{\overline{\odot}} L_y$ does not give the same result as the superposition $L_x \diamond L_y$.

We will use the following notations: NPDA for nondeterministic pushdown automaton; DPDA for deterministic pushdown automaton; NCA for an NPDA that uses only one stack symbol in addition to the bottom of the stack symbol, which is never altered; DCA for deterministic NCA; NFA for nondeterministic finite automaton; DFA for deterministic finite automaton; NLBA for nondeterministic linear-bounded automaton; DLBA for deterministic linear-bounded automaton; NTM for nondeterministic Turing machine; DTM for deterministic Turing machine. As is well-known, NFAs, NPDAs, NLBAs, halting DTMs, and DTMs, accept exactly the regular languages, context-free languages (CFLs), context-sensitive languages

(CSLs), recursive languages, and recursively enumerable languages. We refer the reader to [20] for the formal definitions of these devices.

A *counter* is an integer variable that can be incremented by 1, decremented by 1, left unchanged, and tested for zero. It starts at zero and cannot store negative values. Thus, a counter is a pushdown stack on unary alphabet, in addition to the bottom of the stack symbol which is never altered.

An automaton (NFA, NPDA, NCA, etc.) can be augmented with a finite number of counters, where the "move" of the machine also now depends on the status (zero or non-zero) of the counters, and the move can update the counters. It is well known that a DFA augmented with two counters is equivalent to a DTM [41].

In this paper, we will restrict the augmented counter(s) to be reversal-bounded in the sense that each counter can only reverse (i.e., change mode from nondecreasing to nonincreasing and vice-versa) at most r times for some given r. In particular, when $r = 1$, the counter reverses only once, i.e., once it decrements, it can no longer increment. Note that a counter that makes r reversals can be simulated by $\lceil \frac{r+1}{2} \rceil$ 1-reversal counters. Closure and decidable properties of various machines augmented with reversal-bounded counters have been studied in the literature (see, e.g., [21, 22]). We will use the notation NFCM, NPCM, NCM, etc, to denote an NFA, NPDA, NCA, etc., augmented with reversal-bounded counters.

Example 1. $L = \{xx^r \mid x \in (a+b)^+, |x|_a = |x|_b\}$ can be accepted by an NPCM M with two 1-reversal counters. (The notation $|x|_a$ denotes the number of a's in the string x.) Note that L is not a CFL.

Briefly, M operates as follows: It scans the input and uses the pushdown stack to check that the input is a palindrome (this requires M to "guess" the middle of the string) while using two counters C_1 and C_2 to store the numbers of a's and b's it encounters. Then, at the end of the input, on λ-transitions (i.e., without reading any input symbol), M decrements C_1 and C_2 simultaneusly and verifies that they become zero at the same time. Note that the counters are

1-reversal.

Example 2. $L_k = \{x_1\#\cdots\#x_k \mid x_i \in (a+b)^+, x_j \neq x_k \text{ for } j \neq k\}$ can be accepted by an NFCM M_k with $k(k+1)/2$ 1-reversal counters.

M_k operates as follows: It reads the input and verifies that for $1 \leq i < j \leq k$, x_i and x_j disagree in at least one position. To accomplish this, while scanning x_i, M_k stores in counter C_i a "guessed" position p_i of x_i and records in the state the symbol a_{p_i} in that location. Then later, when it is scanning x_j, M_k stores in counter C_j a guessed location p_j of x_j and records in the state the symbol a_{p_j} in that location. At the end of the input, on λ-transitions, M_k checks that $a_{p_i} \neq a_{p_j}$ and $p_i = p_j$ (by decrementing counters C_i and C_j simultaneously and confirming that they become zero at the same time).

4.3 Closure properties

In this section we study closure properties of various language classes under overlap assembly. We begin with the following general result.

Theorem 4.3.1. *Let $\mathscr{A}$ and $\mathscr{B}$ be two families of languages satisfying the following properties, where # is a symbol not in Σ:*

1. *If $L_x \subseteq \Sigma^*$ is in $\mathscr{A}$ and $L_y \subseteq \Sigma^*$ is in $\mathscr{B}$, then:*

   $L_x^{\#} = \{u\#v \mid |v| > 0, uv \in L_x\}$ *is in $\mathscr{A}$, and*

   $L_y^{\#} = \{v\#w \mid |v| > 0, vw \in L_y\}$ *is in $\mathscr{B}$.*

2. *If $L_1 \subseteq \Sigma^*$ is in $\mathscr{A}$, then $L_1\#\Sigma^*$ is in $\mathscr{A}$.*

   If $L_2 \subseteq \Sigma^*$ is in $\mathscr{B}$, then $\Sigma^*\#L_2$ is in $\mathscr{B}$.

3. *$\mathscr{A}$ is closed under intersection with languages in $\mathscr{B}$.*

4. *If $L \subseteq \Sigma^*\#\Sigma^+\#\Sigma^*$ is in $\mathscr{A}$ and h is a homomorphism that maps # to λ (the empty word) and leaves all other symbols unchanged, then $h(L)$ is in $\mathscr{A}$.*

Then $\mathscr{A}$ is closed under overlap assembly with $\mathscr{B}$, i.e., for any $L_x \in \mathscr{A}$ and $L_y \in \mathscr{B}$, $L_x \overline{\odot} L_y$ is in $\mathscr{A}$.

Proof. Let $L_x, L_y \subseteq \Sigma^*$ be in $\mathscr{A}$ and $\mathscr{B}$, respectively. Let # be a symbol not in Σ. Then by (1), $L_x^\#$ is in $\mathscr{A}$ and $L_y^\#$ is in $\mathscr{B}$. Then by (2), $L_x^\#\#\Sigma^*$ is in $\mathscr{A}$ and $\Sigma^*\#L_y^\#$ in $\mathscr{B}$. Since $\mathscr{A}$ is closed under intersection with languages in $\mathscr{B}$ by (3), $L_x^\#\#\Sigma^* \cap \Sigma^*\#L_y^\#$ is in $\mathscr{A}$. Finally, from (4), $L_x \overline{\odot} L_y = h(L_x^\#\#\Sigma^* \cap \Sigma^*\#L_y^\#)$ is in $\mathscr{A}$. □

A symmetric theorem also holds when the roles of $\mathscr{A}$ and $\mathscr{B}$ in above theorem are switched.

Corollary 4.3.2. *The families of regular languages, context-sensitive languages, recursive languages, recursively enumerable languages, and NFCM languages are closed under overlap assembly.*

Proof. Consider the case $\mathscr{A} = \mathscr{B}$. It is known or easily verified that the families above satisfy the properties in Theorem 4.3.1. In fact, for each family, one can effectively construct the machines satisfying the closure properties listed in the theorem. See, e.g., [20,21]. □

Corollary 4.3.3.

1. *If L_x is regular (resp., context-free, context-sensitive, recursive, recursively enumerable) and L_y is regular, then is $L_x \overline{\odot} L_y$ is regular (resp., context-free, context-sensitive, recursive, recursively enumerable).*

2. *If L_x is regular and L_y is regular (resp., context-free, context-sensitive, recursive, recursively enumerable), then $L_x \overline{\odot} L_y$ is regular (resp., context-free, context-sensitive, recursive, recursively enumerable).*

Proof. Part 1 follows from Theorem 4.3.1. Part 2 follows from the symmetric version of Theorem 4.3.1 with the roles of $\mathscr{A}$ and $\mathscr{B}$ switched. □

Corollary 4.3.4. *If one of L_x and L_y is accepted by an NPCM and the other is accepted by an NFCM, then $L_x \overline{\odot} L_y$ is accepted by an NPCM.*

Proof. This follows from Theorem 4.3.1 and its symmetric version by taking $\mathscr{A}$ to be the class of NPCM languages and $\mathscr{B}$ to be the class of NFCM languages. □

DSPACE($S(n)$) (resp., NSPACE($S(n)$)) denotes the family of languages accepted by $S(n)$ space-bounded DTMs (resp., NTMs). PTIME denotes the family of languages accepted by polynomial time-bounded DTMs.

Theorem 4.3.5. *Let L_x and L_y be CFLs (i.e., accepted by NPDAs). Then*

1. $L_x \overline{\odot} L_y$ *is in* $DSPACE((\log n)^2)$.

2. $L_x \overline{\odot} L_y$ *is in PTIME.*

Proof. Let $L_x, L_y \subseteq \Sigma^*$ be languages. It is known that CFLs can be accepted by DTMs in $(\log n)^2$ space, i.e., they are in $DSPACE((\log n)^2)$. So let M_x and M_y be $(\log n)^2$ space-bounded DTMs that accept L_x and L_y, respectively. We construct a $(\log n)^2$ space-bounded DTM M accepting $L_x \overline{\odot} L_y$ as follows. Given input z of length n , M needs to determine if there is a partition $z = uvw$ for some $u, w \in \Sigma^*$ and $v \in \Sigma^+$ such that $|v| > 0$, $uv \in L_x$ and $vw \in L_y$. To do this, M needs two counters to record the positions i and j where v begins and ends. These counters need $\log n$ space to implement on the DTM. M can systematically examine all possible values of $1 \leq i \leq j \leq n$ to see if for some $i \leq j$, uv is accepted by M_x and vw is accepted by M_y. Clearly, M operates in $(\log n)^2$ space.

The construction for Part 2 follows from Part 1 by noting that CFLs are in PTIME. □

Corollary 4.3.6. *If L_x and L_y are CFLs, then $L_x \overline{\odot} L_y$ is a DCSL (deterministic CSL), but not necessarily a CFL.*

Proof. That $L_x \overline{\odot} L_y$ is a DCSL follows from Theorem 4.3.5 and the observation that $DSPACE((\log n)^2)$ is properly contained in DSPACE(n)(the family of DCSLs). Now let

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

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

Clearly, L_x and L_y are LCFLs. In fact, they can be accepted by DCAs that make only one reversal on the counters. However, $L_x \overline{\odot} L_y = \{\#a^m b^m c^m \$ \mid m \geq 1\}$ is not CF. $\square$

The ideas in the proof of Theorem 4.3.5 can be used to show the following:

Corollary 4.3.7. *The space classes* NSPACE(S) *and* DSPACE(S) *are each closed under overlap assembly for any space bound $S(n) \geq \log n$.*

As stated in Corollary 4.3.2, the family of NFCM languages is closed under overlap assembly. We give another proof below as the construction is needed later. For easy reference, since Corollary 4.3.2 includes other families, we restate the result for NFCM only, in the theorem below.

Theorem 4.3.8. *The family of languages accepted by NFCMs is closed under overlap assembly.*

Proof. Let L_x and L_y be accepted by NFCMs M_x and M_y, respectively. We construct an NFCM M to accept $L_x \overline{\odot} L_y$ as in the proof of Theorem 4.3.5. The only change is that when given input z, M guesses the beginning and end locations i and j of v in the partition $z = uvw$. M simulates M_x on the prefix of z that ends in position j (i.e., on uv) and starts simulating M_y starting in position i of the input z. M accepts if both M_x and M_y accepts. Note that if M_x and M_y have k_1 and k_2 reversal-bounded counters, respectively, then M will have $k_1 + k_2$ reversal-bounded counters. $\square$

Let $\mathbb{N}$ be the set of non-negative integers and k be a positive integer. A subset Q of $\mathbb{N}^k$ is a *linear set* if there exist vectors $\vec{v}_0, \vec{v}_1, \ldots, \vec{v}_n \in \mathbb{N}^k$ such that $Q = \{\vec{v}_0 + i_1\vec{v}_1 + \cdots + i_n\vec{v}_n \mid i_1, \ldots, i_n \in \mathbb{N}\}$. A finite union of linear sets is called a *semilinear set*.

A bounded language $L \subseteq w_1^* \cdots w_k^*$ (for some $k \geq 1$ and non-null words $w_1, \ldots, w_k$) is semilinear if there is a semilinear set $Q \subseteq \mathbb{N}^k$ such that $L = \{w_1^{i_1} \cdots w_k^{i_k} \mid (i_1, \ldots, i_k) \in Q\}$.

Corollary 4.3.9. *The family of semilinear languages is closed under overlap assembly.*

Proof. It is known that a bounded language L (i.e., $\subseteq w_1^* \cdots w_k^*$ for some $k \geq 1$ and words $w_1, \ldots, w_k$) is semilinear if and only if it can be accepted by an NFCM [21]. The result follows from Theorem 4.3.8. $\square$

Corollary 4.3.10. *The family of bounded languages accepted by DFCMs (i.e., DFAs augmented with reversal-bounded counters) is closed under overlap assembly.*

Proof. This follows from Corollary 4.3.9 and the fact that every NFCM accepting a bounded language can be converted to an equivalent DFCM [23]. $\square$

Finally, we consider the family of languages accepted by visibly pushdown automata. A visibly pushdown automaton (VPDA) [1], also known as input-driven pushdown automaton [40], is a restricted version of an NPDA. It is an NPDA where the input symbol determines the (push/stack) operation of the stack. It has a distinguished symbol $\perp$ at the bottom of the stack which is never altered or occur anywhere else. The input alphabet Σ is partitioned into three disjoint alphabets: Σ_c, Σ_r, Σ_ℓ. The machine pushes a specified symbol on the stack if it reads a *call symbol* in Σ_c on the input; it pops a specified symbol if the specified symbol is at top of the stack and it is not the bottom of the stack $\perp$ (otherwise it it does not pop $\perp$) if it reads a *return symbol* in Σ_r on the input; it does not use the (top symbol of) the stack and can only change state if it reads a *local symbol* in Σ_ℓ on the input. The partition into call, return, and local symbols is a property that is inherent to the alphabet Σ. Therefore, if two machines M_x and M_y operate on the same input alphabet Σ, then they have the same set of call, return, and local symbols, respectively.

A VPDA augmented with reversal-bounded counters is called VPCM. We allow the machine to have ε-moves, but in such moves, the stack is not used, only the state and counters are used and updated. Acceptance of an input string is when machine eventually falls off the right end of the input in an accepting state. See [22] for a formal definition.

Theorem 4.3.11. *The family of languages accepted by VPCMs is closed under overlap assembly.*

Proof. The proof is similar to that of Theorem 4.3.8. In that proof, M_x and M_y are VPCMs. The VPCM M constructed from M_x and M_y needs only one pushdown stack, since the operations on the stack of these two machines (being input-driven) are synchronized, i.e., M_x pushes, pops, or leaves the stack unchanged if and only if M_y pushes, pops, or leaves the stack unchanged. □

Clearly, if both M_x and M_y are VPDAs (i.e., have no reversal-bounded counters), then so is M. Hence:

Corollary 4.3.12. *The family of languages accepted by VPDAs is closed under overlap assembly.*

We summarize this section's results regarding closure properties of language classes in the Chomsky hierarchy (plus finite languages) under overlap assembly in Table 4.1. For two language classes $\mathscr{X}$ and $\mathscr{Y}$, the intersection of row $\mathscr{X}$ with column $\mathscr{Y}$ shows the language class $\mathscr{Z}$ from the Chomsky hierarchy such that for all $L_x \in \mathscr{X}$ and $L_y \in \mathscr{Y}$ we have $L_x \overline{\odot} L_y \in \mathscr{Z}$. Noting that $FIN \subseteq REG \subseteq CF \subseteq CS \subseteq RE$ (modulo the condition that λ is not allowed in CS languages), all the entries in Table 4.1 (except for the case when L_x and L_y are finite) follow from Corollary 4.3.2. The case when $L_x \in FIN$ and $L_y \in FIN$, the result is in FIN is obvious.

Also note that each entry in the table is the smallest class from the Chomsky hierarchy which includes $L_x \overline{\odot} L_y$ for all $L_x \in \mathscr{X}$ and $L_y \in \mathscr{Y}$. This follows from Corollary 4.3.6 and the following observation: For a language $L \subseteq \Sigma^*$ and a symbol $\$ \notin \Sigma$, the languages $\$L$ and $L\$$ belong to the same classes in the Chomsky hierarchy as L. Furthermore, $L\$ \overline{\odot} \{\$\} = L\$$ and $\{\$\} \overline{\odot} \$L = \$L$.

$\mathscr{X} \backslash \mathscr{Y}$	FIN	REG	CF	CS	RE
FIN	FIN	REG	CF	CS	RE
REG	REG	REG	CF	CS	RE
CF	CF	CF	CS	CS	RE
CS	CS	CS	CS	CS	RE
RE	RE	RE	RE	RE	RE

Table 4.1: Closure properties of language classes in the Chomsky hierarchy under overlap assembly.

4.4 Decision problems

We have seen in Corollary 4.3.6 that the families of context-free languages (CFLs) and linear context-free languages (LCFLs) are not closed under overlap assembly. We will show that it is undecidable whether or not the overlap assembly of two CFLs (resp., LCFLs) is a CFL (resp., LCFL).

An NPDA (resp., DPDA) is 1-reversal if its stack makes only one reversal, i.e., once it pops, it can no longer push. It is well-known that 1-reversal NPDAs accept exactly the LCFLs. In the following theorems, "DCAs" always means a general DCA, i.e., there is no restriction on counter reversals.

Theorem 4.4.1. *It is undecidable, given 1-reversal DPDAs (resp., DCAs) M_x and M_y accepting languages L_x and L_y, respectively, whether $L_x \overline{\odot} L_y$ is a CFL or not.*

Proof. Let $L_1, L_2 \subseteq \Sigma^*$ be accepted by 1-reversal DPDAs. Let $a, b, c, \#, \$$ be new symbols. Define the following languages:

$$L_x = \{\#a^m w b^m c^n \$ \mid m, n \geq 1, w \in L_1\}$$
$$L_y = \{\#a^m w b^n c^m \$ \mid m, n \geq 1, w \in L_2\}$$

It is easily verified that L_x and L_y can also be accepted by 1-reversal DPDAs. Then $L = L_x \overline{\odot} L_y = L_x \cap L_y$. Clearly, $L = \emptyset$ if and only if $L_1 \cap L_2 = \emptyset$. Now if $L = \emptyset$, then it is obviously a CFL. If $L \neq \emptyset$, we claim that it is not a CFL. For suppose L is a CFL. Apply a homomorphism that maps all symbols in Σ to λ (the empty word) and leaves all other symbols unchanged. Then the resulting language, L', must also be context-free, since CFLs are closed under homomorphism. We get a contradiction, since $L' = \{\#a^m b^m c^m \$ \mid m \geq 1\}$ is not context-free. The result now follows, since the emptiness of intersection of two languages accepted by 1-reversal DPDAs is undecidable [20].

If $L_1, L_2 \subseteq \Sigma^*$ are accepted by DCAs, define the languages:

$$L_x = \{\#wa^m b^m c^n\$ \mid m,n \geq 1, w \in L_1\}$$
$$L_y = \{\#wa^m b^n c^m\$ \mid m,n \geq 1, w \in L_2\}$$

Note that L_x and L_y can be accepted by DCAs as well. Using the same arguments as before, $L_x \overline{\odot} L_y$ is context-free if and only if $L_1 \cap L_2 = \emptyset$. However, the emptiness of intersection of two languages accepted by DCAs is undecidable [21]. $\square$

We need the notion of Parikh map of a language in the proof of the next result. Let $\Sigma = \{a_1, \ldots, a_k\}$. The Parikh map of a language $L \subseteq \Sigma^*$ is defined as

$$\{(|w|_{a_1}, \ldots, |w|_{a_k}) \mid w \in L\},$$

where $|w|_{a_i}$ is the number of a_i's in w.

Theorem 4.4.2. *It is undecidable, given 1-reversal DPDAs (resp., DCAs) M_x and M_y accepting languages L_x and L_y, respectively, whether $L_x \overline{\odot} L_y$ can be accepted by an NFCM.*

Proof. Let $L_1, L_2 \subseteq \Sigma^*$ be accepted by 1-reversal DPDAs, and $a, b, c, \#, \$$ be new symbols. Define the following languages:

$$L_x = \{\#zwcz^R\$ \mid z \in (a+b)^+, w \in L_1\}$$
$$L_y = \{\#zwcz^R\$ \mid z \in (a+b)^+, w \in L_2\}$$

It is easily verified that L_x and L_y can be accepted by 1-reversal DPDAs. Then $L = L_x \overline{\odot} L_y = L_x \cap L_y$. If $L = \emptyset$, then it is obvious that it can be accepted by an NFCM. If $L \neq \emptyset$ and is accepted by an NFCM, then we can construct another NFCM that accepts the language, L', obtained by applying a homomorphism that maps all symbols in Σ to λ and leaves all other symbols unchanged. Clearly, $L' = \{\#zcz^R\$ \mid z \in (a+b)^+\}$. But it is known that L' cannot be accepted by an NFCM [6]. It follows that L cannot be accepted by an NFCM. Hence, it is undecidable whether $L_x \overline{\odot} L_y$ can be accepted by an NFCM.

For the second part, let $L_1, L_2 \subseteq \Sigma^*$ be accepted by DCAs, and $a, b, \#, \$$ be new symbols. Define the following languages:

$$L_x = \{\#a^{i_1}ba^{i_1+1}ba^{i_2}ba^{i_2+1}\cdots a^{i_k}ba^{i_k+1}w\$ \mid$$
$$k \geq 1, i_1, \cdots, i_k \geq 1, i_1 = 1, w \in L_1\}$$
$$L_y = \{\#a^{j_1}ba^{j_2}ba^{j_2+1}\cdots a^{j_{k-1}}ba^{j_{k-1}+1}ba^{j_k}w \mid$$
$$k \geq 1, j_1, \cdots, j_k \geq 1, j_1 = 1, w \in L_2\}$$

Then $L_x \overline{\odot} L_y = \{\#a^1ba^2ba^3ba^4\cdots a^{2k-1}ba^{2k}w\$ \mid k \geq 1, w \in L_1 \cap L_2\}$. Hence, $L_x \overline{\odot} L_y = \emptyset$ if and only if $L_1 \cap L_2 = \emptyset$. Suppose $L_x \overline{\odot} L_y \neq \emptyset$. One can verify that the Parikh map of if $L_x \overline{\odot} L_y \neq \emptyset$ is not a semilinear set. Since the Parikh map of any NFCM language is semilinear [21], it follows that if $L_x \overline{\odot} L_y \neq \emptyset$, it cannot be accepted by an NFCM. We conclude that $L_x \overline{\odot} L_y$ is accepted by an NFCM if and only if $L_1 \cap L_2 = \emptyset$, which is undecidable. □

Another interesting decision question is to decide, whether $L_x \overline{\odot} L_y$ is empty, finite, or infinite.

Theorem 4.4.3.

1. *It is decidable, given L_x and L_y, one of which is accepted by an NPCM and the other by an NFCM, whether $L_x \overline{\odot} L_y$ is empty, finite, or infinite.*

2. *It is decidable, given L_x and L_y, accepted by VPCMs, whether $L_x \overline{\odot} L_y$ is empty, finite, or infinite.*

Proof. This follows from Corollary 4.3.4 and Theorem 4.3.11 and the fact that it is decidable, given an NPCM, whether the language it accepts is empty, finite, or infinite. □

We end this section with a discussion of a special case of overlap assembly, when the languages L_x and L_y are the same. More precisely, if $L \subseteq \Sigma^*$, let

$$L_{\overline{\odot}} = L \overline{\odot} L = \{uvw \mid v \in \Sigma^+, u, w \in \Sigma^*, uv, vw \in L\}.$$

Obviously, the positive closure and decidable results in the previous section and this section when the class $\mathscr{A}$ = class $\mathscr{B}$ also hold for this special case of overlap assembly (by taking $L_y = L_x$). However the proofs for the non-closure and undecidable results need to be modified.

Theorem 4.4.4. *If L is accepted by a DCA, then $L_{\overline{\odot}}$ need not be a CFL.*

Proof. Let $L = \{\%a^m\#b^m c^n \mid m, n \geq 1\} \cup \{\#b^m c^m\$ \mid m \geq 1\}$. Clearly, L can be accepted by a DCA that makes only one reversal on its counter.

Suppose $L_{\overline{\odot}}$ is a CFL. Define the regular language $L' = \%a^+\#b^+c^+\$$. Then the language $L'' = L_{\overline{\odot}} \cap L'$ must also be a CFL. We get a contradiction since $L'' = \{\%a^m\#b^m c^m\$ \mid m \geq 1\}$ is not a CFL. □

Theorem 4.4.5. *It is undecidable, given a language L accepted by a 1-reversal DPDA (resp., DCA) M, whether $L_{\overline{\odot}}$ is a CFL.*

Proof. Let $L_1, L_2 \subseteq \Sigma^*$ be accepted by 1-reversal DPDAs. Define

$$L = \{\%a^m\#b^n wc^m\$ \mid m, n \geq 1, w \in L_1\} \cup$$
$$\{\#b^m wc^m\$\$ \mid m \geq 1, w \in L_2\}.$$

It can be verified that L can be accepted by a 1-reversal DPDA. Then by an argument similar to that in the proof of Theorem 4.4.1, $L_{\overline{\odot}}$ is a CFL if and only if $L_1 \cap L_2 = \emptyset$, which is undecidable.

Now, let $L_1, L_2 \subseteq \Sigma^*$ be accepted by DCAs. Define

$$\begin{aligned} L = &\{\%a^m\#b^nc^mw\$ \mid m,n \geq 1, w \in L_1\} \cup \\ &\{\#b^mc^mw\$\$ \mid m \geq 1, w \in L_2\}. \end{aligned}$$

It can be verified that L can be accepted by a DCA. By an argument similar to that in the proof of Theorem 4.4.1, $L_{\overline{\odot}}$ is a CFL if and only if $L_1 \cap L_2 = \emptyset$, which is undecidable. □

4.5 Iterated overlap assembly

We define a combinatorial library of words as a set of the form

$$\{\alpha_1\alpha_2\cdots\alpha_n \mid \alpha_i \in \{X_i, Y_i\} \text{ for } i = 1, \ldots, n\}$$

where $X_1, X_2, \ldots, X_n, Y_1, Y_2, \ldots, Y_n \in \Sigma^+$ are distinct sequences. It is often required that all X_i and Y_i are of the same length. However, some experiments use the fact that X_i and Y_i have different lengths: for example, in [42] all X_i have the same length which is shorter than the length of all Y_i, thus allowing to use gel electrophoresis to separate the strings from this library by how many X_i they contain.

Combinatorial libraries of DNA strands have applications in many areas, including DNA computing where, e.g., a *mix-and-split* procedure was used to generate the solution space (a combinatorial library of binary numbers) for a chess problem, [13]. A similar technique was used to generate the pool of solutions to a 20-variable solution of the 3-SAT problem, [4], the largest experiment to date that solved a computational problem with a DNA algorithm. Efficient generation of combinatorial libraries of this type, obtained by using XPCR, was initially proposed in [14], and further investigated in [16].

In this section we formally prove that the iterated overlap assembly can theoretically generate this library with some restrictions on the words X_i, Y_i. We consider the following library where an additional symbol \$ is inserted between every pair of X_i/Y_i and X_{i+1}/Y_{i+1}:

$$\{\alpha_1\$\alpha_2\$\cdots\alpha_n\$ \mid \alpha_i \in \{X_i, Y_i\} \text{ for } i = 1\ldots,n\}. \tag{4.1}$$

For simplicity, we view \$ as an additional letter that does not appear inside any of the words X_i or Y_i. The purpose of introducing the letters \$ is that each letter \$ has to match the position of another letter \$ during overlap assembly (i.e., no proper suffix of $\alpha_i\$$ is identical to a proper prefix of $\alpha_j\$$). If one prefers to avoid the introduction of this additional letter in the strings (e.g., for practical purposes), it is sufficient to design the set of strings

$$C = \{X_1, \ldots, X_n, Y_1, \ldots, Y_n\}$$

such that either C contains only equal-length words that are *overlap-free* or, less restrictive, C is a *solid code* (i.e., overlap- and infix-free), see e.g., [24]. In this case, the symbols \$ in the library (4.1) become markers (of width 0) which match during overlap assembly because of the design of the set C.

We start by generalizing the definition of $L_{\overline{\odot}} = L \overline{\odot} L$. The *iterated overlap assembly* of a language L, [7], is defined as follows:

$$\mu_0(L) = L \qquad\qquad \mu_{i+1}(L) = \mu_i(L) \overline{\odot} \mu_i(L)$$
$$\mu_*(L) = \bigcup_{i\geq 0} \mu_i(L)$$

In particular $\mu_1(L) = L \overline{\odot} L = L_{\overline{\odot}}$. Since $w \in w \overline{\odot} w$ for any non-empty word w, from the definition it easily follows that $\mu_i(L) \subseteq \mu_{i+1}(L)$ for $L \in \Sigma^+$. It can be shown (using intersections with appropriate regular languages) that Theorems 4.4.4 and 4.4.5 also hold for iterated overlap assembly.

We will now show that we can generate the combinatorial library (4.1) by *(i)* starting with a set of strands

$$\{\alpha_k\$\alpha_{k+1}\$ \mid 1 \le k \le n-1, \alpha_i \in \{X_i, Y_i\} \text{ for } i = 1, \dots, n\},$$

(ii) iteratedly applying overlap assembly until no new strands are produced anymore (Theorem 4.5.5), and *(iii)* extracting the longest strands from the result. We will also show (Theorem 4.5.4) that the number of steps of this process is logarithmic in the size of the input.

Definition A string $x \in L$ is said to be *terminal* with respect to language L if $x \overline{\odot} L = L \overline{\odot} x = \{x\}$.

Definition A set of strings $T(L) \subseteq L$ is said to be *the maximal terminating set* of L if every $w \in T(L)$ is terminal with respect to L and for all $w \in L \backslash T(L)$, w is not terminal with respect to L, that is,

$$T(L) = \{w \in L \mid w \overline{\odot} L = L \overline{\odot} w = \{w\}\}$$

Lemma 4.5.1. *If $t \in T(L)$, then $t \in T(\mu_1(L))$. More generally, if $t \in T(L)$, then $t \in T(\mu_*(L))$*

Proof. We prove the contrapositive: if $t \notin T(L \overline{\odot} L)$, then $t \notin T(L)$ for any $t \in L$ (if $t \notin L$ the statement is obviously true). There exists $w \in \mu_1(L)$ and $u \neq t$ such that either $u \in w \overline{\odot} t$ or $u \in t \overline{\odot} w$. If $w \in L$, then $t \notin T(L)$. Thus, $w \in \mu_1(L) \backslash L$. There are $w_1, w_3 \in \Sigma^*, w_2 \in \Sigma^+$ such that $w = w_1 w_2 w_3 \in w_1 w_2 \overline{\odot} w_2 w_3$ where $w_1 w_2, w_2 w_3 \in L$. If $u \in t \overline{\odot} w$, there are $u_1, u_3 \in \Sigma^*, u_2 \in \Sigma^+$ such that $u = u_1 u_2 u_3$, where $u = u_1 u_2$ and $w = u_2 u_3 = w_1 w_2 w_3$. There are two cases possible: (A) either u_2 is a proper prefix of $w_1 w_2$, or (B) $w_1 w_2$ is a prefix of u_2.

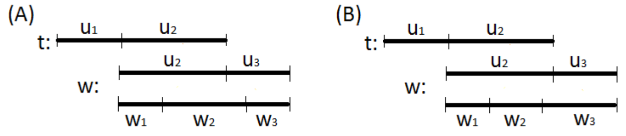

Figure 4.2: Illustration of cases (A) and (B) from the proof of Lemma 4.5.1.

In case (A), there is $u_1w_1w_2 \neq t$ in $t \overline{\odot} w_1w_2 \subseteq t \overline{\odot} L$ and therefore $t \notin T(L)$. In case (B), there is $u \in t \overline{\odot} w_2w_3 \subseteq t \overline{\odot} L$ and therefore $t \notin T(L)$ because $w_2 \neq \varepsilon$. We can similarly prove that $t \notin T(L)$ when $w \in \mu_1(L)$ and $u \neq t$ exists such that $u \in w \overline{\odot} t$. Hence, we prove that $t \in T(L)$ implies $t \in T(\mu_1(L))$. By applying this result recursively, we can similarly prove that if $t \in T(L)$, then $t \in T(\mu_*(L))$. □

Definition We define z_{k_1,k_2} for any $k_1 \leq k_2$ as follows.

$$z_{k_1,k_2} = \{\alpha_{k_1}\$\alpha_{k_1+1}\$\cdots\alpha_{k_2}\$ \mid \alpha_i \in \{X_i,Y_i\}, k_1 \leq i \leq k_2]\}$$

z_{k_1,k_2} is not defined for $k_1 > k_2$.

Informally, z_{k_1,k_2} is the set of words consisting of the catenation of $k_2 - k_1 + 1$ consecutive words α, separated by dollar signs. Note that, with this notation, $z_{1,n}$ represents the required combinatorial library.

Definition We define $Z(m,n)$ for all $m \geq 2$ as equal to the union of all z_{k_1,k_2} such that $1 \leq k_2 - k_1 < m$ for $m \leq n$, and equal to $Z(n,n)$ for $m > n$:

$$Z(m,n) = \begin{cases} \bigcup_{p=1,\ldots,m-1} \bigcup_{k_1=1,\ldots,n-p} z_{k_1,k_1+p} & \text{if } 2 \leq m \leq n \\ Z(n,n) & \text{if } m > n. \end{cases}$$

Informally, $Z(m,n)$ is the set of all strands consisting of at most m consecutive words α (separated by dollar signs), where $2 \leq m \leq n$. With this notation, $Z(2,n)$ represents the initial starting set, and $Z(n,n)$ contains all strands consisting of catenations of consecutive words α, with the minimum number of consecutive words α in such a catenation being 2, and the maximum number being n. Note that $Z(n,n)$ contains the desired library $z_{1,n}$ as a subset.

Lemma 4.5.2. *Let $x = \alpha_{k_1}\$\cdots\alpha_{k_2}\$$ and $y = \beta_{l_1}\$\cdots\beta_{l_2}\$$ be words where $\alpha_i, \beta_i \in \{X_i,Y_i\}$ and $1 \leq k_1,k_2,l_1,l_2 \leq n$. If $k_1 \leq l_1 \leq k_2 \leq l_2$ and $\alpha_i = \beta_i$ for $i = l_1, l_1+1, \ldots, k_2$, then*

$$x \overline{\odot} y = \{\alpha_{k_1}\$\alpha_{k_1+1}\$\cdots\alpha_{k_2}\$\beta_{k_2+1}\$\beta_{k_2+2}\$\cdots\beta_{l_2}\$\}.$$

Otherwise, $x \overline{\odot} y = \emptyset$.

Proof. It is easy to see that

$$\alpha_{k_1}\$\alpha_{k_1+1}\$\cdots\alpha_{k_2}\$\beta_{k_2+1}\$\beta_{k_2+2}\$\cdots\beta_{l_2}\$ \in x \overline{\odot} y$$

if $k_1 \leq l_1 \leq k_2 \leq l_2$ and $\alpha_i = \beta_i$ for $i = l_1, l_1+1, \ldots, k_2$, because

$$\alpha_{l_1}\$\alpha_{l_1+1}\$\cdots\alpha_{k_2}\$ = \beta_{l_1}\$\beta_{l_1+1}\$\cdots\beta_{k_2}\$$$

can serve as overlap of x and y.

The words x and y cannot overlap in any other way since the symbols \$ have to match up in both words and all words $X_1, X_2, \ldots, X_n, Y_1, Y_2, \ldots, Y_n$ are distinct. In particular, when one of the conditions $k_1 \leq l_1 \leq k_2 \leq l_2$ and $\alpha_i = \beta_i$ for $i = l_1, l_1+1, \ldots, k_2$ is not satisfied, the two words x and y cannot form an overlap at all and, therefore, $x \overline{\odot} y = \emptyset$. □

Lemma 4.5.3. *If* $2 \leq m_1, m_2 \leq n$, *then* $Z(m_1, n) \overline{\odot} Z(m_2, n) = Z(m_1 + m_2 - 1, n)$.

Proof. Let $x \in Z(m_1, n)$, $y \in Z(m_2, n)$ and $w \in x \overline{\odot} y$. Clearly, we have $x = \alpha_{k_1}\$\alpha_{k_1+1}\$\cdots\alpha_{k_2}\$$ and $y = \beta_{l_1}\$\beta_{l_1+1}\$\cdots\beta_{l_2}\$$ where $1 \leq k_1, k_2, l_1, l_2 \leq n$, $k_2 - k_1 < m_1$, and $l_2 - l_1 < m_2$. From Lemma 4.5.2 we obtain that $w \in x \overline{\odot} y$ is only possible if $l_1 \leq k_2$ and

$$w = \alpha_{k_1}\$\alpha_{k_1+1}\$\ldots\alpha_{k_2}\$\beta_{k_2+1}\$\beta_{k_2+2}\$\ldots\beta_{l_2}\$.$$

This implies that $l_2 - k_1 \leq l_2 - l_1 + k_2 - k_1 < m_1 + m_2 - 1$; note that we also have $l_2 - k_1 < n$. We conclude $w \in Z(m_1 + m_2 - 1, n)$ and, more general, $Z(m_1, n) \overline{\odot} Z(m_2, n) \subseteq Z(m_1 + m_2 - 1, n)$.

Conversely, consider a word $w = \alpha_k\$\alpha_{k+1}\cdots\alpha_l\$ \in Z(m_1 + m_2 - 1, n)$ where $1 \leq k, l \leq n$, $1 \leq l - k < \min(m_1 + m_2 - 1, n)$, and $\alpha_i \in \{X_i, Y_i\}$. If $l - k < m_1$, then $w \in Z(m_1, n)$ and $y = \alpha_{l-1}\$\alpha_l\$ \in Z(m_2, n)$; this implies that $w \in w \overline{\odot} y \subseteq Z(m_1, n) \overline{\odot} Z(m_2, n)$. Otherwise, we let $j = k + m_1 - 1$ and note that $x = \alpha_k\$\alpha_{k+1}\cdots\alpha_j\$ \in Z(m_1, n)$. Furthermore, because $l - k <$

m_1+m_2-1, we have that $l-j=l-k-m_1+1<m_2$ which implies that $y=\alpha_j\$\alpha_{j+1}\cdots\alpha_l\$\in Z(m_2,n)$. By Lemma 4.5.2, we have $w\in x\overline{\odot}y\subseteq Z(m_1,n)\overline{\odot}Z(m_2,n)$. □

The following theorem shows that, starting from an initial set $Z(2,n)$ we will obtain, after $\lceil\log_2(n-1)\rceil$ or more overlap assemblies, the set $Z(n,n)$ which is a superset of the combinatorial library $z_{1,n}$.

Theorem 4.5.4. *For $L=Z(2,n)$ and $k\geq 0$, we have $\mu_k(L)=Z(2^k+1,n)$. Moreover, $\mu_*(L)=Z(n,n)$.*

Proof. We prove the statement by induction. Clearly, the statement holds for the base case where $k=0$ as $\mu_0(L)=L=Z(2,n)$.

Using the induction hypothesis $\mu_k(L)=Z(2^k+1,n)$ and Lemma 4.5.3, we obtain that

$$\begin{aligned}\mu_{k+1}(L)=\mu_k(L)\overline{\odot}\mu_k(L)&=Z(2^k+1,n)\overline{\odot}Z(2^k+1,n)\\&=Z(2\cdot(2^k+1)-1,n)=Z(2^{k+1}+1,n).\end{aligned}$$

Because $Z(m,n)\subseteq Z(n,n)$ for all $m\in\mathbb{N}$, we obtain the second statement $\mu_*(L)=Z(n,n)$. □

Next, we prove the main result of this section, namely that the maximal terminal set of $\mu_*(L)=\mu_k(L)$ is the desired combinatorial library $z_{1,n}$.

Theorem 4.5.5. *For $L=Z(2,n)$ we have $T(\mu_*(L))=z_{1,n}$.*

Proof. From Theorem 4.5.4, we know that $\mu_*(L)=Z(n,n)$. First, note that for every word $w=\alpha_1\$\alpha_2\$\cdots\alpha_n\$\in z_{1,n}\subseteq Z(n,n)$ there does not exist any word $v\in Z(n,n)$, such that any suffix (resp., prefix) of w (including w itself) is a proper prefix (resp., suffix) of v. Therefore, we must have $w\overline{\odot}Z(n,n)=Z(n,n)\overline{\odot}w=\{w\}$. Thus, $z_{1,n}$ only contains words which are terminal with respect to $\mu_*(L)$.

Next, consider a word

$$w = \alpha_{k_1}\$\alpha_{k_1+1}\$\ldots\alpha_{k_2}\$ \in Z(n,n)\backslash z_{1,n}$$

where $\alpha_i \in \{X_i, Y_i\}$, $1 \le k_1 < k_2 \le n$, and $k_1 > 1$ or $k_2 < n$. If $k_1 > 1$, then it is easy to see that

$$w \neq X_{k_1-1}\$\alpha_{k_1}\$\alpha_{k_1+1}\cdots\alpha_{k_2}\$ \in X_{k_1-1}\$\alpha_{k_1}\$ \overline{\odot}\, w \subseteq Z(n,n)\,\overline{\odot}\, w.$$

Otherwise, $k_2 < n$, and we have

$$w \neq \alpha_{k_1}\$\alpha_{k_1+1}\$\cdots\alpha_{k_2}\$X_{k_2+1}\$ \in w\,\overline{\odot}\,\alpha_{k_2}\$X_{k_2+1}\$ \subseteq w\,\overline{\odot}\, Z(n,n).$$

In either case, w is not terminal with respect to $\mu_*(L)$. We conclude that $T(\mu_*(L)) = z_{1,n}$. $\square$

Observe that the result of iterated overlap assembly applied to the initial set $Z(2,n)$ produces the set $Z(n,n)$ that contains the required library $z_{1,n}$, but it contains also other intermediate strings. One can use various techniques to extract the library $z_{1,n}$ from this solution. For example, gel electrophoresis can be used to separate strands by length, and the longest strands, which are the desired combinatorial library strands, can then be extracted.

4.6 Conclusions

This paper studies properties of the operation of overlap assembly, a formal language operation that models the process of linear overlap assembly of DNA strands: Two DNA strands that partially "overlap", in the sense that the suffix of one is the Watson-Crick complement of a prefix of another, can be concatenated with the aid of a DNA Polymerase enzyme. We obtain closure properties of various language classes under this operation, and discuss various decision problems. We also investigate the iterated overlap assembly and demonstrate that, under some simplifying assumptions, it can be used to generate a DNA combinatorial library.

Acknowledgements

We thank Giuditta Franco for useful suggestions and discussions on experimental aspects of XPCR, as well as Sepinoud Azimi and Florin Manea for pointing out important references.

Bibliography

[1] R. Alur and P. Madhusudan. Visibly pushdown languages. In Proc. *ACM Symposium on Theory of Computing,* STOC, pages 202–211. ACM-Press, 2004.

[2] A. Angeleska, N. Jonoska, M. Saito, and L. F. Landweber. RNA-guided DNA assembly. *Journal of Theoretical Biology*, 248(4):706–720, 2007.

[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] R. S. Braich, N. Chelyapov, C. Johnson, P. W. K. Rothemund, and L. Adleman. Solution of a 20-variable 3-SAT problem on a DNA computer. *Science*, 296(5567):499–502, 2002.

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

[6] E. Chiniforooshan, M. Daley, O. H. Ibarra, L. Kari, and S. Seki. One-reversal counter machines and multihead automata: revisited. *Theoretical Computer Science*, 454:81–87, 2012.

[7] E. Csuhaj-Varjú, I. Petre, and G. Vaszil. Self-assembly of strings and languages. *Theoretical Computer Science*, 374(1-3):74–81, 2007.

[8] A. R. Cukras, D. Faulhammer, R. J. Lipton, and L. F. Landweber. Chess games: a model for RNA based computation. *Biosystems*, 52(1-3):35–45, 1999.

[9] M. Daley, L. Kari, G. Gloor, and R. Siromoney. Circular contextual insertions/deletions with applications to biomolecular computation. In Proc. *String Processing and Information Retrieval,* SPIRE, pages 47–54, 1999.

[10] J. Dassow, C. Martín-Vide, G. Păun, and A. Rodríguez-Patón. Conditional concatenation. *Fundamenta Informaticae*, 44(4):353–372, 2000.

[11] A. Ehrenfeucht, I. Petre, D. M. Prescott, and G. Rozenberg. *Circularity and other invariants of gene assembly in ciliates*, page 8197. World Scientific, 2001.

[12] S. K. Enaganti, L. Kari, and S. Kopecki. A formal language model of DNA polymerase activity. *Fundamenta Informaticae*, 138:179–192, 2015.

[13] D. Faulhammer, A. R. Cukras, R. J. Lipton, and L. F. Landweber. Molecular computation: RNA solutions to chess problems. *Proceedings of the National Academy of Sciences*, 97(4):1385–1389, 2000.

[14] G. Franco. A polymerase based algorithm for SAT. In M. Coppo, E. Lodi, and G. Pinna, editors, *Theoretical Computer Science*, volume 3701 of *LNCS*, pages 237–250. Springer Berlin Heidelberg, 2005.

[15] G. Franco, C. Giagulli, C. Laudanna, and V. Manca. DNA extraction by XPCR. In C. Ferretti, G. Mauri, and C. Zandron, editors, Proc. *DNA Computing,* (DNA 11), volume 3384 of *LNCS*, pages 104–112, 2005.

[16] G. Franco and V. Manca. Algorithmic applications of XPCR. *Natural Computing*, 10(2):805–819, 2011.

[17] G. Franco, V. Manca, C. Giagulli, and C. Laudanna. DNA recombination by XPCR. In A. Carbone and N. A. Pierce, editors, Proc. *DNA Computing,* (DNA 12), volume 3892 of *LNCS*, pages 55–66, 2006.

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

[19] T. Head, D. Pixton, and E. Goode. Splicing systems: regularity and below. In M. Hagiya and A. Ohuchi, editors, *DNA Based Computers: DNA Computing,* DNA 8, volume 2568 of *LNCS*, pages 262–268, 2003.

[20] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Inc., 1978.

[21] O. H. Ibarra. Reversal-bounded multicounter machines and their decision problems. *J. ACM*, 25(1):116–133, 1978.

[22] O. H. Ibarra. Automata with reversal-bounded counters: a survey. In Proc. *Descriptional Complexity of Formal Systems,* DCFS, pages 5–22, 2014.

[23] O. H. Ibarra and S. Seki. Characterizations of bounded semilinear languages by one-way and two-way deterministic machines. *Int. J. Foundations of Computer Science*, 23(6):1291–1305, 2012.

[24] H. Jürgensen and S. Konstantinidis. Codes. In *Handbook of Formal Languages*, pages 511–607. Springer, 1997.

[25] P. D. Kaplan, Q. Ouyang, D. S. Thaler, and A. Libchaber. Parallel overlap assembly for the construction of computational DNA libraries. *Journal of Theoretical Biology*, 188(3):333–341, 1997.

[26] L. Kari, J. Kari, and L. Landweber. Reversible molecular computation in ciliates. In J. Karhumäki, H. Maurer, G. Păun, and G. Rozenberg, editors, *Jewels are Forever*, pages 353–363. Springer Berlin Heidelberg, 1999.

[27] 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 (DNA 18)*, volume 7433 of *LNCS*, pages 98–109, 2012.

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

[29] 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*, pages 329–346, 1999.

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

[31] S. M. Kim. An algorithm for identifying spliced languages. In T. Jiang and D. Lee, editors, Proc. *Computing and Combinatorics Conference,* COCOON, volume 1276 of *LNCS*, pages 403–411, 1997.

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

[33] L. F. Landweber and L. Kari. The evolution of cellular computing: natures solution to a computational problem. *Biosystems*, 52(13):3–13, 1999.

[34] L. Ledesma, D. Manrique, and A. Rodríguez-Patón. A tissue P system and a DNA microfluidic device for solving the shortest common superstring problem. *Soft Computing*, 9(9):679–685, 2005.

[35] V. Manca and G. Franco. Computing by polymerase chain reaction. *Mathematical Biosciences*, 211(2):282–298, 2008.

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

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

[38] F. Manea, V. Mitrana, and J. Sempere. Some remarks on superposition based on Watson-Crick-like complementarity. In V. Diekert and D. Nowotka, editors, *Developments in Language Theory*, volume 5583 of *LNCS*, pages 372–383, 2009.

[39] C. Martín-Vide, G. Păun, J. Pazos, and A. Rodríguez-Patón. Tissue P systems. *Theoretical Computer Science*, 296(2):295–326, 2003.

[40] K. Mehlhorn. Pebbling moutain ranges and its application of DCFL-recognition. In Proc. *Automata, Languages and Programming,* ICALP, pages 422–435. Springer-Verlag, 1980.

[41] M. L. Minsky. Recursive unsolvability of Post's problem of "Tag" and other topics in theory of Turing machines. *The Annals of Mathematics*, 74(3):437–455, 1961.

[42] Q. Ouyang, P. D. Kaplan, S. Liu, and A. Libchaber. DNA solution of the maximal clique problem. *Science*, 278(5337):446–449, 1997.

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

[44] D. M. Prescott, A. Ehrenfeucht, and G. Rozenberg. Molecular operations for DNA processing in hypotrichous ciliates. *European Journal of Protistology*, 37(3):241–260, 2001.

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

[46] G. Păun, G. Rozenberg, and A. Salomaa. *DNA computing: new computing paradigms*. Texts in Theoretical Computer Science. Springer, 2006.

[47] W. P. Stemmer. DNA shuffling by random fragmentation and reassembly: in vitro recombination for molecular evolution. *Proceedings of the National Academy of Sciences*, 91(22):10747–10751, 1994.

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

[49] 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.

Chapter 5

Further remarks on DNA overlap assembly[1]

5.1 Introduction

The word and language operation *overlap assembly* was first introduced by Csuhaj-Varju, Petre, and Vaszil - under the name (self-)assembly - in [5], and later studied in [7], as a formal model of the linear self-assembly of DNA strands. Formally, the overlap assembly is a binary operation which, when applied to two input strings xy and yz (where y is their non-empty *overlap*), produces the output xyz.

The study of overlap assembly as a formal language operation is part of ongoing efforts to provide a formal framework and rigorous treatment of DNA-based information and DNA-based computation. More specifically, this study can be placed in the context of studies of DNA bio-operations enabled by the actions of DNA polymerase enzymes, such as hairpin completion and its inverse operation, hairpin reduction [4, 26, 28, 29], overlapping concatenation [31], and directed extension [8].

The activity of DNA Polymerase presupposes the existence of a DNA single strand, called

[1]A version of this chapter, including an abstract, has been submitted to the Information and Computation journal (S.K. Enaganti, O.H. Ibarra, L. Kari, S. Kopecki. Further remarks on DNA overlap assembly.)

template, and of a second short DNA strand, called *primer*, that is Watson-Crick complementary to the template and binds to it. Given a supply of individual nucleotides, a DNA polymerase then extends the primer, at one of its ends only, by adding individual nucleotides complementary to the template nucleotides, one by one, until the end of the template is reached. In the wet lab, the iteration of this process is used to obtain an exponential replication of DNA strands, in a protocol called *Polymerase Chain Reaction (PCR)*. Experimentally, (parallel) overlap assembly of DNA strands under the action of DNA Polymerase enzyme was used for gene shuffling in, e.g., [35]. In the context of experimental DNA Computing, overlap assembly was used in, e.g., [6,9,23,33] for the formation of combinatorial DNA or RNA libraries. Overlap assembly can also be viewed as modelling a special case of an experimental procedure called cross-pairing PCR, introduced in [11] and studied in, e.g., [10, 12, 13, 27].

This paper is a continuation of the theoretical analysis of overlap assembly, as a formal language operation, that was started in [5] and [7]. Following Section 5.2 comprising definitions, notations and basic properties of overlap assembly, in Section 5.3 we correlate the overlap assembly operation with the superposition operation introduced in [3] and determine closure properties of iterated overlap assembly. A string w is terminal with respect to a language L if $w \in L$ and the result of the overlap assembly between w and L equals $\{w\}$, and the terminal set $T(L)$ contains all words that are terminal with respect to L. In Section 5.4 we investigate the closure properties of terminal sets of complete languages (languages closed under overlap assembly). In Section 5.5 we study three decision problems: deciding the completeness of an arbitrary language, deciding whether a string is terminal with respect to a language, and deciding whether a language is generated by an overlap assembly operation of two other given languages. Section 5.6 contains concluding remarks.

5.2 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 w, respectively. The sets $\mathrm{pref}(w)$, $\mathrm{inf}(w)$, and $\mathrm{suff}(w)$ contain, respectively, all prefixes, infixes, and suffixes of w. A prefix (resp., infix or suffix) x of w is *proper* if $x \neq w$. We employ the following notation: $\mathrm{Pref}(w) = \mathrm{pref}(w) \setminus \{w\}$, $\mathrm{Inf}(w) = \mathrm{inf}(w) \setminus \{w\}$, and $\mathrm{Suff}(w) = \mathrm{suff}(w) \setminus \{w\}$. This notation is naturally extended to languages; for example, $\mathrm{Suff}(L) = \bigcup_{w\in L} \mathrm{Suff}(w)$. The complement of a language $L \subseteq \Sigma^*$ is $L^c = \Sigma^* \setminus L$.

Let $\mathbb{N}$ be the set of non-negative integers and k be a positive integer. A subset Q of $\mathbb{N}^k$ is a *linear set* if there exist vectors $\vec{v}_0, \vec{v}_1, \ldots, \vec{v}_n \in \mathbb{N}^k$ such that $Q = \{\vec{v}_0 + i_1\vec{v}_1 + \cdots + i_n\vec{v}_n \mid i_1, \ldots, i_n \in \mathbb{N}\}$. A finite union of linear sets is called a *semilinear set*.

Let $\Sigma = \{a_1, \ldots, a_k\}$. The *Parikh map* of a language $L \subseteq \Sigma^*$, denoted $\Psi(L)$, is defined as

$$\Psi(L) = \{(|w|_{a_1}, \ldots, |w|_{a_k}) \mid w \in L\},$$

where $|w|_{a_i}$ is the number of a_i's in w.

5.2.1 The overlap assembly

An *involution* is a function $\theta : \Sigma^* \to \Sigma^*$ with the property that θ^2 is the identity. θ is called an *antimorphism* if $\theta(uv) = \theta(v)\theta(u)$. Traditionally, the Watson-Crick complementarity of DNA strands has been modeled as an anti-morphic involution over the DNA alphabet $\Delta = \{A, C, G, T\}$, [18, 24].

Using the convention that a word x over this alphabet represents the DNA single strand x in the 5' to 3' direction, the overlap assembly of a strand uv with a strand $\theta(w)\theta(v)$ first forms a partially double-stranded DNA molecule with v in uv and $\theta(v)$ in $\theta(w)\theta(v)$ attaching to each other, see Figure 1(a). DNA polymerase enzyme will extend the 3' end of uv with the strand

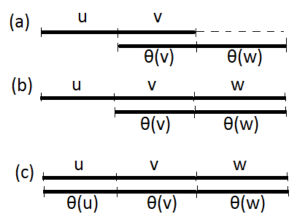

Figure 5.1: (a) The two input DNA single-strands, uv and $\theta(w)\theta(v)$ bind to each other through their complementary segments v and $\theta(v)$, forming a partially double-stranded DNA complex. (b) DNA polymerase extends the 3' end of the strand uv. (c) DNA polymerase extends the 3' end of the other strand. The resulting DNA double strand is considered to be the output of the *overlap assembly* of the two input single strands.

w, see Figure 1(b). Similarly, the 3' end of $\theta(w)\theta(v)$ will extended, resulting in a full double strand whose upper strand is uvw, see Figure 1(c). Formally, the overlap assembly between uv and $\theta(w)\theta(v)$ is uvw. Assuming that all involved DNA strands are initially double-stranded, that is, whenever the strand x is available, its Watson-Crick complement $\theta(x)$ is also available, one can further simplify this model and, given two words x,y over an alphabet Σ, define the *overlap assembly of x with y*, [5], as:

$$x\,\overline{\odot}\,y = \{z \in \Sigma^+ \mid \exists u,w \in \Sigma^*, \exists v \in \Sigma^+ : x = uv, y = vw, z = uvw\}.$$

The definition of overlap assembly can be extended to languages in the natural way.

The *iterated overlap assembly* $\mu_*(L)$ of a language L is defined as:

$$\mu_0(L) = L, \qquad \mu_{i+1}(L) = \mu_i(L)\,\overline{\odot}\,\mu_i(L), \qquad \mu_*(L) = \bigcup_{i\geq 0} \mu_i(L).$$

Since $w \in w\,\overline{\odot}\,w$ for any non-empty word w, it easily follows that $\mu_i(L) \subseteq \mu_{i+1}(L)$ for $L \in \Sigma^+$.

A string $w \in L$ is said to be *terminal* with respect to the language L if $w\,\overline{\odot}\,L = L\,\overline{\odot}\,w = \{w\}$. A set of strings $T(L) \subseteq L$ is said to be the *(maximal) terminal set* of L if every $w \in T(L)$ is terminal with respect to L and for all $w \in L\backslash T(L)$, w is not terminal with respect to L, that is,

$$T(L) = \{w \in L \mid w\,\overline{\odot}\,L = L\,\overline{\odot}\,w = \{w\}\}.$$

A *complete* language is a language closed under overlap assembly, that is, $L \overline{\odot} L = L$. For every language L, the language $\mu_*(L)$ is complete and $\mu_*(L) = L$ if and only if L is complete. The investigation of the terminal set $T(L)$ makes most sense if L is complete, but it is well-defined for all languages L.

5.2.2 Basic properties of the overlap assembly

In this section we present a few basic properties of the (iterated) overlap assembly and complete languages.

The operation of overlap assembly is not associative, as seen in the next example.

Example 5.2.1. The following example shows that overlap assembly is not associative

$$(aba \overline{\odot} a) \overline{\odot} bac = \{abac\}, \qquad\qquad aba \overline{\odot} (a \overline{\odot} bac) = \emptyset.$$

The words in the example are chosen such that a and bac cannot form an overlap; note that, however, $aba \overline{\odot} bac = (aba \overline{\odot} a) \overline{\odot} bac$ in this example.

However, overlap assembly does satisfy a property related to associativity, as seen in the following result.

Lemma 5.2.1. *For languages L_x, L_y and L_z, we have*

$$((L_x \overline{\odot} L_y) \overline{\odot} L_z) \cup (L_x \overline{\odot} L_z) = (L_x \overline{\odot} (L_y \overline{\odot} L_z)) \cup (L_x \overline{\odot} L_z).$$

Proof. Consider a word $w \in ((L_x \overline{\odot} L_y) \overline{\odot} L_z) \cup (L_x \overline{\odot} L_z)$. If $w \in (L_x \overline{\odot} L_z)$, then w clearly belongs to $(L_x \overline{\odot} (L_y \overline{\odot} L_z)) \cup (L_x \overline{\odot} L_z)$. Otherwise, there exist $x \in L_x$, $y \in L_y$ and $z \in L_z$ such that $w = (x \overline{\odot} y) \overline{\odot} z$. Because $w \notin x \overline{\odot} z$, we can factorize $y = uy'v$ such that $w = xy'z$, u is a nonempty suffix of x, and v is a nonempty prefix of z. We conclude $uy'z \in (y \overline{\odot} z)$ and $w = xy'v \in x \overline{\odot} (y \overline{\odot} z)$. The converse inclusion follows by similar arguments. $\square$

The next lemma considers a word $w \in \mu_*(L)$ and its infixes that belong to L. It is not difficult to see that the entire word w is "covered" by overlapping words from L. We formalize this observation by saying that every two consecutive letters in the word w are covered by an infix v of w that belongs to L.

Lemma 5.2.2. *Let L and $w = a_1 a_2 \cdots a_n \in \mu_*(L)$ for letters $a_1, a_2, \ldots, a_n \in \Sigma$. For each integer k with $1 \le k < n$ there exist integers j, ℓ such that $1 \le j \le k < \ell \le n$ and $a_j a_{j+1} \cdots a_\ell \in L$.*

Proof. The statement is trivially true if $|w| = n < 2$ or $w \in \mu_0(L) = L$. By induction, assume that the statement holds for all words in $\mu_i(L)$ and consider $w \in \mu_{i+1}(L) \setminus \mu_i(L)$. We have $w \in x \overline{\odot} y$ for some $x, y \in \mu_i(L)$. If $k < |x|$, we find j, ℓ such that $1 \le j \le k < \ell \le |x|$ and $a_j a_{j+1} \cdots a_\ell \in L$ because x is a proper prefix of w. If $k \ge |x| \ge n - |y| + 1$, we find j, ℓ such that $n - |y| + 1 \le j \le k < \ell \le n$ and $a_j a_{j+1} \cdots a_\ell \in L$ because y is a proper suffix of w. □

Lemma 5.2.2 leads to the following statement, that allows us to identify non-terminal words in a language $\mu_*(L)$.

Lemma 5.2.3. *For a language L and a word $w \in \mu_*(L) \setminus T(\mu_*(L))$, there exist $z \in \mu_*(L) \setminus \{w\}$ and $x \in L$ such that $z \in w \overline{\odot} x$ or $z \in x \overline{\odot} w$.*

Proof. A word $w \in \mu_*(L)$ is not terminal with respect to $\mu_*(L)$ if there exists $y \in \mu_*(L)$ and $v \neq w$ such that $v \in w \overline{\odot} y$ or $v \in y \overline{\odot} w$. Due to symmetry, we only consider the case when $v \in w \overline{\odot} y$. Let $y = y_1 y_2$ such that $v = w y_2$ and y_1 is a suffix of w. We will use Lemma 5.2.2 with the following interpretation: since $y = y_1 y_2 \in \mu_*(L)$, there are $x_1, x_2 \in \Sigma^+$ with $x = x_1 x_2 \in L$ such that x_1 is a suffix of y_1 and x_2 is a prefix of y_2. Let $z = w x_2$ and observe that $z \in w \overline{\odot} x$ and $z \in \mu_*(L) \setminus \{w\}$. □

Finally, let us state a simple observation on complete languages.

Lemma 5.2.4. *For any $L \subseteq \Sigma^*$ and $\# \notin \Sigma$, the languages $\#L$ and $L\#$ are complete.*

Proof. Since every word in $\#L$ contains exactly one letter #, this letter has to match in an overlap assembly. Therefore, $\#w \in \#x \overline{\odot} \#y$ for $x, y, w \in \Sigma^+$ if and only if x is a prefix of y and $w = y$. □

5.2.3 Automata models, augmented with counters

We will use the following notations: NPDA for nondeterministic pushdown automaton; DPDA for deterministic pushdown automaton; NCA for an NPDA that uses only one stack symbol in addition to the bottom of the stack symbol, which is never altered; DCA for deterministic NCA; NFA for nondeterministic finite automaton; DFA for deterministic finite automaton; NLBA for nondeterministic linear-bounded automaton; DLBA for deterministic linear-bounded automaton; NTM for nondeterministic Turing machine; DTM for deterministic Turing machine. As is well-known, NFAs, NPDAs, NLBAs, halting DTMs, and DTMs, accept exactly the regular languages, context-free languages (CFLs), context-sensitive languages (CSLs), recursive languages, and recursively enumerable languages. We refer the reader to [17] for the formal definitions of these devices.

A *counter* is an integer variable that can be incremented by 1, decremented by 1, left unchanged, and tested for zero. It starts at zero and cannot store negative values. Thus, a counter is a pushdown stack on unary alphabet, in addition to the bottom of the stack symbol which is never altered. Note that an NCA (DCA) is equivalent to an NFA (DFA) which is augmented with a counter.

An automaton (NFA, NPDA, NCA, etc.) can be augmented with a finite number of counters, where the "move" of the machine also now depends on the status (zero or non-zero) of the counters, and the move can update the counters. It is well known that a DFA augmented with two counters is equivalent to a DTM [32].

In this paper, we will restrict the augmented counter(s) to be reversal-bounded in the sense that each counter can only reverse (i.e., change mode from non-decreasing to non-increasing and vice-versa) at most r times for some given r. In particular, when $r = 1$, the counter reverses only once, i.e., once it decrements, it can no longer increment. Note that a counter that makes r reversals can be simulated by $\lceil \frac{r+1}{2} \rceil$ 1-reversal counters. Closure and decidable properties of various machines augmented with reversal-bounded counters have been studied in the literature (see, e.g., [19, 20]). We will use the notation NFCM, NPCM, NCM, etc, to denote an NFA, NPDA, NCA, etc., augmented with reversal-bounded counters.

Automata with reversal-bounded counters can "count", as seen in the following example.

Example 5.2.2. $L = \{xx^r \mid x \in \{a,b\}^+, |x|_a = |x_b|\}$ can be accepted by an NPCM M with two 1-reversal counters. Briefly, M operates as follows: It scans the input and uses the push down stack to check that the input is a palindrome (this requires M to "guess" the middle of the string) while using two counters C_1 and C_2 to store the numbers of a's and b's it encounters. Then, at the end of the input, on λ-transitions (i.e., without reading any input symbol), M decrements C_1 and C_2 simultaneously and verifies that they become zero at the same time. Note that the counters are 1-reversal.

A non-deterministic stack automaton (NSA) is a generalization of an NPDA in that during the computation, the machine can enter the stack in a read-only mode. It can only push and pop by returning to the top of the stack [14].

A non-deterministic stack-counter automaton (NSCA) is a special case of an NSA in that the the stack alphabet is unary, except for the bottom of the stack symbol which is never altered and only used for this purpose [15]. So, again, the machine can enter the unary storage in a read-only mode, and it can only increment/decrement the storage by returning to the top of the stack. Note that an NSCA is strictly more powerful than an NCA. For example, the languages $L_1 = \{(a^n\#)^k \mid n,k \geq 1\}$ and $L_2 = \{a^1\#a^2\#\cdots\#a^k \mid k \geq 1\}$ can be accepted by NSCAs (in fact, deterministically for the case of L_2), but cannot be accepted by NCAs, since these languages have non-semilinear Parikh maps but languages accepted by NCAs (in fact, by NPCMs) have semilinear Parikh maps [19].

NSPACE($S(n)$) and NTIME($T(n)$) denote the classes of languages accepted by non-deterministic Turing machines in $S(n)$ space and $T(n)$ time, respectively. DSPACE($S(n)$) and DTIME($T(n)$) are the the corresponding deterministic classes. PTIME denotes the class of languages accepted but deterministic Turing machines in polynomial time.

5.3 The related superposition operation

The superposition operation is a binary operation proposed by Bottoni, Labella, Manca, and Mitrana in [3] to model the actions of DNA Polymerase enzymes. The result of the superposition operation between words $x, y \in \Sigma^+$, denoted by $x \diamond y$, consists of the set of all words $z \in \Sigma^+$ obtained by any of the four following cases ($^-$ denotes the *morphic complement*, that is, $^-$ is a morphism such that $\overline{\overline{u}} = u$ for all words u):

1. If there exist $u, v \in \Sigma^*, w \in \Sigma^+$ such that $x = uw, y = \overline{w}v$, then $z = uw\bar{v} \in x \diamond_1 y$.
2. If there exist $u, v \in \Sigma^*$ such that $x = u\bar{y}v$, then $z = u\bar{y}v \in x \diamond_2 y$.
3. If there exist $u, v \in \Sigma^*, w \in \Sigma^+$ such that $x = wv, y = u\overline{w}$, then $z = \bar{u}wv \in x \diamond_3 y$.
4. If there exist $u, v \in \Sigma^*$ such that $y = u\bar{x}v$, then $z = \bar{u}x\bar{v} \in x \diamond_4 y$.

The superposition operation can be naturally extended to languages. For more on the superposition operation between two words (languages), the reader is referred to [3, 30].

The superposition operation and the overlap assembly are closely related. In particular, when we replace the complement $^-$ by the identity, then case 1 above is identical to overlap assembly, i.e., $x \overline{\odot} y = x \diamond_1 y$; case 3 above is symmetrical to the overlap assembly, i.e., $x \overline{\odot} y = y \diamond_3 x$; furthermore, cases 2 and 4 above give $x \diamond_2 y = y \diamond_4 x = x$ if y is an infix of x. However, in the general case of two languages or when we consider a "real" complement function, the overlap assembly $L_x \overline{\odot} L_y$ does not give the same result as the superposition $L_x \diamond L_y$.

If $^-$ is the identity, then the overlap assembly of a language L with itself gives $\mu_1(L) = L \overline{\odot} L = L \diamond L$ and, moreover, the iterated overlap assembly coincides with the analogously defined[2] iterated superposition $\mu_*(L) = \diamond^*(L)$. In other words, the iterated overlap assembly is a special case of the iterated superposition. Therefore, the (positive) closure results for the iterated superposition, obtained in [3, 30], also hold for the iterated overlap assembly. Indeed,

[2]When $^-$ is not the identity, the iterated superposition has to be defined slightly different than the iterated overlap assembly, because $L \subseteq L \diamond L$ does not necessarily hold. In [3,30] two iterated versions of the superposition are defined which turn out to yield the same language.

the same results were independently obtained in [5], in its study of closure properties of iterated overlap assembly:

Proposition 1 ([5]). *The language classes of regular, context-sensitive, recursive and recursively enumerable languages are closed under iterated overlap assembly.*

The result that the family of context-free languages is not closed under iterated overlap assembly (resp., iterated superposition) was proven in [5] (resp., [3]). The following statement strengthens that result.

Theorem 5.3.1. *There is a language L accepted by a 1-reversal DCA (i.e., a DFA with one counter that makes only 1 reversal) whose iterated overlap assembly, $\mu^*(L)$, cannot be accepted by any NPCM (i.e., an NPDA with reversal-bounded counters).*

Proof. Let $L = \{\$a^i\#\$a^{i+1}\# \mid i \geq 1\}$. Clearly, L can be accepted by a 1-reversal DCA. Suppose $\mu_*(L)$ could be accepted by an NPCM. Then $\mu_*(L) \cap \$a\#(\$a^+\#)^+ = \{\$a^1\#\$a^2\# \cdots \$a^k\#\$a^{k+1}\# \mid k \geq 1\}$ could also be accepted by an NPCM. This is not possible since the Parikh map of this language is not semilinear, but it is known that the Parikh map of any NPCM language is semilinear [19]. □

In contrast to the previous result, for any k and NFCM language L, $\mu_k(L)$ can be accepted by an NFCM. This follows from the fact that $L \overline{\odot} L$ is an NFCM language if L is an NFCM language [7].

In [3], the notion of *maximal (adult) language* is defined, which is analogous to the terminal set of a language. The authors consider *maximal (adult) words* with respect to the iterated superposition of some language L: A word x is a maximal word with respect to $\diamond^*(L)$ if $x \in \diamond^*(L)$ and $x \diamond (\diamond^*(L)) \subseteq \{x\}$. Also, $max \diamond^* (L)$ is defined as the set of all maximal words with respect to $\diamond^*(L)$. We immediately obtain that $T(\mu_*(L))$ is the special case of $max \diamond^* (L)$ where the complement $^-$ is replaced by the identity. From the results in [3,30] we obtain the following result for terminal sets of complete, regular languages:

Proposition 2. *The terminal set $T(L)$ of a complete, regular language L is regular. In particular, the terminal set $T(\mu_*(L))$ is regular if L is regular.*

It is known from [3] that there exists a context-sensitive L language such that the maximal language $max \diamond^* (L)$ is undecidable. This result can be strengthened as follows.

Theorem 5.3.2. *There exists a (complete) language $L \in \mathrm{DSPACE}(\log n)$, such that $T(L)$ and $T(\mu_*(L))$ are undecidable.*

Proof. Let M be a deterministic Turing machine with input alphabet Σ and let $\$, \# \notin \Sigma$. Note that the languages $L_1 = \$\Sigma^*\#$ and

$$L_2 = \{\$w\#^n \mid w \in \Sigma^*, n \geq 1, \text{ and } M \text{ accepts } w \text{ using at most } \log(n) \text{ space}\}$$

can be decided in deterministic log-space. Observe that the language $L = L_1 \cup L_2$ is complete (Lemma 5.2.4), hence $T(L) = T(\mu^*(L))$; no word in L_2 belongs to $T(L)$; and a word $\$w\# \in L_1$ belongs to $T(L)$ if and only if M does not accept w. Because M may accept an undecidable language, we cannot decide $T(L)$ in general. □

A closure property of superposition with respect to the language class PTIME has been stated in [30]. Thus, we have:

Proposition 3. *The class* PTIME *is closed under iterated overlap assembly.*

5.4 Iterated overlap assembly and terminal sets

In this section we further explore the iterated overlap assembly and terminal sets. In the previous section we have seen that the terminal set of a complete context-sensitive language can be undecidable (Theorem 5.3.2). In this section we will show that the terminal set of a complete context-free language is always context-sensitive (Theorem 5.4.2). We also show that, for a context-free language L, the language $T(\mu_*(L))$ is context-sensitive (Theorem 5.4.3). We establish space and time complexities for deciding $T(L)$ when L is complete and given via certain automata models (Theorem 5.4.4 and Corollaries 5.4.5–5.4.8). Lastly, we establish a relation between Schützenburger constants and the iterated overlap assembly (Theorem 5.4.9).

We start with an observation about terminal sets of complete languages.

Theorem 5.4.1. *The terminal set of a complete language L is given by*

$$T(L) = L \setminus (\mathrm{Pref}(L) \cup \mathrm{Suff}(L))$$

Proof. First, we prove $T(L) \subseteq L \setminus (\mathrm{Pref}(L) \cup \mathrm{Suff}(L))$ by contradiction: suppose there were $w \in T(L) \cap \mathrm{Pref}(L) \subseteq L$. There is $x \in L$ such that w is a proper prefix of x. Therefore, $x \in w \overline{\odot} x \subseteq w \overline{\odot} L$ which contradicts that $w \in T(L)$. A similar case can be made when $w \in T(L) \cap \mathrm{Suff}(L)$.

Now, let $w \in L \setminus (\mathrm{Pref}(L) \cup \mathrm{Suff}(L))$ and suppose that $w \notin T(L)$. There is $x \neq w$ such that $x \in w \overline{\odot} L$ or $x \in L \overline{\odot} w$; by symmetry, we assume $x \in w \overline{\odot} L$. Note that $x \in L$ because L is complete. Since w is a proper prefix of $x \in L$ we obtain a contradiction. □

Note that Proposition 2 now follows as a corollary of Theorem 5.4.1, since regularity is preserved under the used operations. Next, we consider terminal sets of context-free languages.

Theorem 5.4.2. *The terminal set of a complete, context-free language L is not necessarily a context-free language, but is always a context-sensitive language.*

Proof. We first prove that $T(L)$ for a complete, context-free language L may not be context-free using a counter-example. Let

$$L = \{\#a^i b^j c^k \mid i, j, k \geq 1, k \leq i \vee k \leq j\}.$$

Note that L is clearly context-free and complete (using Lemma 5.2.4), but the terminal set $T(L)$ is not context-free:

$$T(L) = \{\#a^i b^j c^k \mid i, j, k \geq 1, k = \max\{i, j\}\}.$$

If L is a context-free language, then the language $M = \mathrm{Pref}(L) \cup \mathrm{Suff}(L)$ is context-free as well. Because the family of context-sensitive languages is closed under intersection [17] and complementation [22, 36], we have that $T(L) = L \cap M^c$ is context-sensitive. $\square$

Later, in Corollary 5.4.7, we will see that $T(L)$ is, in fact, in DSPACE($\log^2 n$) and also in DTIME($n^{2.373}$).

So far, we have seen that the iterated overlap assembly of a context-free language is context-sensitive (Theorem 5.4.2), and that the terminal set of a context-sensitive language can be undecidable (Theorem 5.3.2). Next we prove that, for context-free language L, the language $T(\mu_*(L))$ is context-sensitive.

Theorem 5.4.3. *The terminal set of $\mu_*(L)$ is context-sensitive if L is context-free.*

Proof. In order to decide $w \in T(\mu_*(L))$ we decide the two properties

1. $w \in \mu_*(L)$ and
2. $(\mathrm{pref}(w) \cap \mathrm{Suff}(L) \cap \Sigma^+) \cup (\mathrm{suff}(w) \cap \mathrm{Pref}(L) \cap \Sigma^+) = \emptyset$.

Both properties can be decided in linear space when L is context-free; see Proposition 1 for property 1. Furthermore, it is clear that properties 1 and 2 are necessary for w to belong to $T(\mu_*(L))$.

In order to show that the conditions are sufficient, consider $w \notin T(\mu_*(L))$. If $w \notin \mu_*(L)$, then condition 1 is violated. Otherwise, there exist $z \in \mu_*(L) \setminus \{w\}$ and $x \in L$ such that $z \in w \overline{\odot} x$ or $z \in x \overline{\odot} w$, by Lemma 5.2.3. If $z \in w \overline{\odot} x$, then $\mathrm{suff}(w) \cap \mathrm{Pref}(L) \cap \Sigma^+ \neq \emptyset$; and if $z \in x \overline{\odot} w$, then $\mathrm{pref}(w) \cap \mathrm{Suff}(L) \cap \Sigma^+ \neq \emptyset$ — hence, property 2 is violated. $\square$

Next, we investigate the terminal set $T(L)$ for various complete languages L. For convenience, we assume that all (one-way) machines have a right end marker in their read-only input tape. For non-deterministic machines, this assumption can be made without loss of generality, since such a machine can "guess" the end of the input and simulate the computation on the end marker using λ-moves at the end of the input.

Theorem 5.4.4. *If L is a complete language accepted by an NSCA, then $T(L)$ is in* NSPACE$(\log n)$ *and is also in* PTIME.

Proof. Let L be accepted by an NSCA M. We claim that Pref(L) and Suff(L) can be accepted by NSCAs. We construct an NSCA M_1 accepting Pref(L). M_1, when given input x, will accept x if there is some non-empty y such that xy is accepted by M. M_1 operates as follows: It simulates M on x faithfully. Then after processing x, M_1, on λ-moves, guesses some suffix-string y symbol-by-symbol and continues simulating the computation of M on y and accepts if M accepts. Similarly, an NSCA M_2 accepting Suff(L) can be constructed, but in this case, given input x, M_2, on λ-moves, guesses some non-empty prefix-string y and simulates M. After guessing and processing y, M_2 reads x and continues simulating M and accepts if M accepts. Clearly, from M_1 and M_2, we can also construct an NSCA M_3 accepting $L_3 = \text{Pref}(L) \cup \text{Suff}(L)$.

It is known that every NSCA can be converted to an equivalent quasi-real time NSCA, i.e., there is a d such that during the computation, the number of consecutive λ moves on the input is bounded by d (hence the NSCA runs in linear time) [15]. We can then convert M_3 to an equivalent quasi-real time NSCA M_4. It follows that the stack-counter values during the accepting computation is linear on the length of the input. Clearly, the stack-counter can be simulated by two ordinary counters whose values would also be linear in the length of the input. Hence the stack-counter of M_4 can be stored in $\log n$ space on a read/write tape. It follows that $L_3 = \text{Pref}(L) \cup \text{Suff}(L)$ is in NSPACE$(\log n)$. Now the complement L_3^c of L_3 is also in NSPACE$(\log n)$ [22, 36]. Since NSPACE$(\log n)$ is clearly closed under intersection, by Theorem 5.4.1, $T(L) = L \setminus (\text{Pref}(L) \cup \text{Suff}(L)) = L \cap L_3^c$ is also in NSPACE$(\log n)$. That $T(L)$ is in PTIME follows from the fact that NSPACE$(\log n) \subseteq$ PTIME. □

We can use a similar construction as in the proof of Theorem 5.4.4 to obtain the following results.

Corollary 5.4.5. *If L is a complete language accepted by an NCA, then $T(L)$ is in* NSPACE$(\log n)$ *and* DTIME(n^2).

Proof. $T(L)$ in NSPACE$(\log n)$ follows from Theorem 5.4.4 since NCA is a special case of NSCA. The time complexity follows from the proof of Theorem 5.4.4 and the fact that every language accepted by an NCA is in DTIME(n^2) [16] and that this class is closed under complementation and intersection. □

Corollary 5.4.6. *If L is a complete language accepted by an NFCM, then $T(L)$ is in* NSPACE$(\log n)$ *and also in* PTIME.

Proof. If L is accepted by an NFCM M, then, as in the proof of Theorem 5.4.4, we can construct an NFCM M_3 accepting $L_3 = \text{Pref}(L) \cup \text{Suff}(L)$. It is known that for any NFCM, there is a fixed constant d such that any string of length n accepted by the NFCM can be accepted in dn steps, i.e., the values of the counters are at most dn [1]. It follows that any NFCM language is in NSPACE$(\log n)$. Then, as in the proof of Theorem 5.4.4, $T(L) = L \cap L_3^c$ is in NSPACE$(\log n)$ and, hence, also in PTIME. □

The next corollary strengthens Theorem 5.4.2.

Corollary 5.4.7. *If L is a complete language accepted by an NPDA (i.e., L is a complete context-free language), then $T(L)$ is in* DSPACE$(\log^2 n)$ *and* DTIME$(n^{2.373})$ *(= complexity of matrix multiplication).*

Proof. This follows by similar constructions as in Theorem 5.4.4 using the fact that every language accepted by an NPDA is in DSPACE$(\log^2 n)$ and also in DTIME$(n^{2.373})$ (= complexity of matrix multiplication [37]), and the fact that these classes are closed under complementation and intersection. □

Similarly, since the family of linear context-free languages is in DTIME(n^2) [25], we have:

Corollary 5.4.8. *If L is a complete linear context-free language (i.e., accepted by a 1-reversal NPDA), then $T(L)$ is in* DTIME(n^2).

Lastly, we consider the relation between Schützenburger constants [34] and the iterated overlap assembly. A word $w \in \Sigma^+$ is a *(Schützenberger) constant* for L if $w \in \inf(L)$ and for all words $u_1, u_2, v_1, v_2 \in \Sigma^*$, we have that

$$u_1 w v_1 \in L \text{ and } u_2 w v_2 \in L \implies u_1 w v_2 \in L.$$

The existence of constants in a language seems to have a close connection to languages that are generated by some biologically inspired systems; for example, every splicing language has a constant [2].

Theorem 5.4.9. *Every word $w \in \Sigma^+$ in $\mu_*(L) \setminus \mathrm{Inf}(L)$ is a constant for $\mu_*(L)$. If, in addition, w also satisfies $w \in \inf(T(\mu_*(L)))$, then w is a constant for $T(\mu_*(L))$ as well.*

Proof. Let $w \in \mu_*(L) \setminus \mathrm{Inf}(L)$ and let $u_1, v_1, u_2, v_2 \in \Sigma^*$ such that $u_1 w v_1 \in \mu_*(L)$ and $u_2 w v_2 \in \mu_*(L)$. Let us show that $u_1 w \in \mu_*(L)$. Let x_1 be the longest suffix of $u_1 w$ that belongs to $\mu_*(L)$. If u_1 is a proper prefix of x_1, then $u_1 w \in x_1 \overline{\odot} w \subseteq \mu_*(L)$. Otherwise ($x_1$ is a prefix of u_1), let x_2 such that $x_1 x_2 = u_1 w v_1$. Lemma 5.2.2 implies that there are $y_1, y_2 \in \Sigma^+$ with $y_1 y_2 \in L$ such that y_1 is a suffix of x_1 and y_2 is a prefix of x_2. Since w cannot be an infix of y_2, we obtain that $x_1 y_2 \in x_1 \overline{\odot} y_1 y_2 \subseteq \mu_*(L)$ is a prefix of $u_1 w$; this contradicts the choice of x_1 which is supposed to be the longest prefix with that property. By a symmetric argument, we can show that $w v_2 \in \mu_*(L)$, and therefore, $u_1 w v_2 \in u_1 w \overline{\odot} w v_2 \subseteq \mu_*(L)$. We conclude that w is a constant for $\mu_*(L)$.

Now, consider the case when $u_1 w v_1 \in T(\mu_*(L))$ and $u_2 w v_2 \in T(\mu_*(L))$ and, hence, $w \in \inf(T(\mu_*(L)))$. As before, $u_1 w v_2 \in \mu_*(L)$. In order to obtain a contradiction, suppose that $u_1 w v_2 \notin T(\mu_*(L))$. By Lemma 5.2.3 there exist $z \in \mu_*(L) \setminus \{w\}$ and $x \in L$ such that $z \in u_1 w v_2 \overline{\odot} x$ or $z \in x \overline{\odot} u_1 w v_2$. Due to symmetry, we only consider the case when $z \in u_1 w v_2 \overline{\odot} x$. Let y be the nonempty suffix of x such that $z = u_1 w v_2 y$. Because x cannot have w as proper infix, we have $u_2 w u v_2 y \in u_2 w v_2 \overline{\odot} x$ which contradicts the premise $u_2 w v_2 \in T(\mu_*(L))$. $\square$

5.5 Decision problems

In this section we consider three decision problems: whether a language is complete (Subsection 5.5.1), whether a string is terminal with respect to a language (Subsection 5.5.2), and whether the overlap assembly of two given languages equals a given third one (Subsection 5.5.3).

5.5.1 Deciding the completeness of a language

The problem of deciding if a given language is complete was studied in [5] for language classes in Chomsky hierarchy. In this subsection we narrow the gap between the language classes whose completeness is decidable and those for which it is undecidable. Recall first a result from [5]:

Proposition 4 ([5])**.**

1. *It is decidable if any given regular language is complete.*
2. *It is undecidable if any given context-free(resp., context-sensitive, recursively enumerable) language is complete.*

The following shows that Proposition 4, part 1 holds for DFCMs (i.e., DFAs augmented with reversal bounded counters).

Theorem 5.5.1. *It is decidable, given a DFCM M, if $L(M)$ is complete.*

Proof. Given a DFCM M accepting L, we construct an NFCM M' accepting $L' = L \overline{\odot} L$ as follows:

M', when given input z, guesses a partition $z = uvw$ for some u, v, w with $v \neq \lambda$, and checks that uv is in L by running M on uv, and vw is in L by running another copy of M on vw, and accepts z if and only if M accepts uv and vw. Note that M' uses two sets of counters of M to simulate the two copies of M. Clearly, $L(M') = L \overline{\odot} L$. It follows that $L \overline{\odot} L = L$ if and only if $L(M') \subseteq L$, and if and only if $L(M') \cap L^c = \emptyset$. Since the family of DFCM languages is effectively closed under complementation, we can construct from M a DFCM accepting L^c [19]. The result follows, since we can construct, given two NFCMs, an NFCM accepting their intersection language, and emptiness of NFCMs is decidable [19]. $\square$

In contrast to Theorem 5.5.1, for the case of NFCM we have the following result which strengthens Proposition 4, part 2:

Theorem 5.5.2. *It is undecidable, given a 1-reversal NCA (i.e., an NFA augmented with one counter which makes only 1 reversal) M, whether $L(M)$ is complete.*

Proof. We reduce the problem to the undecidability of the halting problem for deterministic Turing machines (DTMs) on an initially blank tape.

Let Z be single-tape DTM. Without loss of generality, we assume that if Z halts on an initially blank tape, it makes at least two moves. We construct a 1-reversal NCA M which accepts the language:

> $L(M) = \{w \mid w \neq ID_1\#ID_2\cdots\#ID_k$, where $k \geq 3, ID_1$ is the initial configuration of Z on an initially blank tape, ID_k is the (unique) halting configuration of Z if it halts, and ID_{i+1} is the valid successor of $ID_i\}$.

Let Σ be the alphabet over which $L(M)$ is defined. Clearly, $L(M) = \Sigma^*$ (which is complete) if Z does not halt on blank tape. However, if Z halts on blank tape, $L(M) = \Sigma^* \setminus \{x\}$ for exactly one string x of the form $ID_1\#ID_2\#\cdots\#ID_k$, $k \geq 3$, and it is not complete because: $ID_1\#ID_2$ is in $L(M)$ (since it is not x) and $ID_2\#\cdots\#ID_k$ is also in $L(M)$ (since it is not x). Hence, $x = ID_1\#ID_2\#\cdots\#ID_k$ is in $L(M)\overline{\odot}L(M)$, but it is not in $L(M)$. It follows that $L(M)$ is complete if and only if Z does not halt on blank tape, which is undecidable. $\square$

5.5.2 Deciding the terminality of strings

We now investigate the problem of deciding whether a given string is terminal with respect to a language. The following result gives sufficient conditions for the decidability of whether a string w is terminal with respect to a language.

Theorem 5.5.3. *Let $\mathscr{M}$ be a class of machines, and let $\mathscr{L}$ be the corresponding class of accepted languages, satisfying:*

1. *if L is in $\mathscr{L}$, then for any string w, $w \overline{\odot} L$ and $L \overline{\odot} w$ are also in $\mathscr{L}$;*

2. *$\mathscr{L}$ is closed under intersection with regular sets;*

3. *$\mathscr{L}$ has a decidable emptiness problem;*

and items 1 and 2 are effective. Then it is decidable, given a machine M in $\mathscr{M}$ and a string w in $L(M)$, if w is terminal with respect to $L(M)$.

Proof. Let L be a language accepted by a machine $M \in \mathscr{M}$ and w be a string in L. Then, by item 1, we can construct machines in $\mathscr{M}$ accepting $L_1 = w \overline{\odot} L$ and $L_2 = L \overline{\odot} w$. To check that $L_1 = \{w\}$, we do the following: since $\mathscr{L}$ is closed under intersection with regular sets (item 2) and has a decidable emptiness problem (item 3), we check that $L_1 \cap \{w\}^c = \emptyset$ (note that $w \in L_1$ is always true). Similarly, we can check that $L_2 = \{w\}$. □

Almost all classes of one-way nondeterministic machines satisfy condition 1 in Theorem 5.5.3. Indeed, given M accepting L and $w \in L$, we construct a machine M' accepting $w \overline{\odot} L$ which, on a given input z, guesses a decomposition of z into uvx and checks that that $uv = w$ and vx is accepted by M. Similarly, we can construct a machine M'' to accept $L \overline{\odot} w$.

As examples, the classes of languages accepted by NPCMs and NSAs satisfy condition 1 of Theorem 5.5.3, while condition 2 is also clearly satisfied. Since emptiness for NPCMs and NSAs is decidable [14, 19], we have:

Corollary 5.5.4. *It is decidable, given an NPCM (resp., NSA) M and a string w in $L(M)$, whether w is terminal with respect to $L(M)$.*

Next we show that condition 3 in Theorem 5.5.3 is a necessary condition. We say that a class of languages $\mathscr{L}$ is closed under *distinct-symbol concatenation* if, given $L \in \mathscr{L}$ and a symbol \$, not in the alphabet of L, \$$L$ and L\$ are in $\mathscr{L}$.

Theorem 5.5.5. *Let $\mathscr{M}$ be a class of machines, and $\mathscr{L}$ be the corresponding class of accepted languages. Assume that $\mathscr{L}$ is effectively closed under distinct-symbol concatenation and union with a singleton language. If $\mathscr{L}$ has an undecidable emptiness problem, then it is undecidable, given a language L in $\mathscr{L}$ and a string w in L, whether w is terminal with respect to L.*

Proof. Let M_1 be a machine in $\mathscr{M}$ accepting a language $L_1 \subseteq \Sigma^*$. Let $\%, \#, \$$ be new symbols not in Σ. Consider the string $w = \%\#$. Construct a machine M in $\mathscr{M}$ accepting the language $L = \{\%\#\} \cup \{\%\#x\$ \mid x \in L_1\}$. Clearly, $\%\# \overline{\odot} L = L \overline{\odot} \%\# = \{\%\#\}$ if and only if $L_1 = \emptyset$. We cannot decide if w is terminal, since emptiness for $\mathscr{L}$ is undecidable. □

An example of a class $\mathscr{L}$ such as the one in Theorem 5.5.5 is the class of languages accepted by real-time DFAs augmented with two unrestricted counters (real-time deterministic 2-counter machines). Real-time here means that the machines have no λ-moves. It is known that it is undecidable, given a deterministic machine Z which has *no input tape* but with two counters that are initially zero, whether it will halt [32]. We construct from Z, a real-time deterministic 2-counter machine M which, when given a unary string a^n, simulates Z's counters while reading an input symbol on each move, and accepts if and only if Z halts after n steps. Hence, $L(M) = \emptyset$ if and only if Z does not halt. It follows that the emptiness problem for real-time deterministic 2-counter machines is undecidable. Clearly, the assumptions in Theorem 5.5.5 are satisfied. Hence, we have:

Corollary 5.5.6. *It is undecidable, given a real-time deterministic 2-counter machine M and a string w in $L(M)$, whether w is terminal with respect to $L(M)$.*

As above, we can also construct a (one-way) real-time deterministic $\log n$ space-bounded DTM to simulate Z. Hence, the emptiness problem for these machines is also undecidable. Thus, we have:

Corollary 5.5.7. *It is undecidable, given a real-time deterministic $\log n$ space-bounded DTM M and a string w in $L(M)$, whether w is terminal with respect to $L(M)$.*

Unlike NPCMs, it can be shown that an NSA when augmented with even only two reversal-bounded counters, call this NSCM(2), has an undecidable emptiness problem. The proof of this result (using the techniques in [19]) is a reduction to the undecidability of Hilbert's Tenth Problem. Hence, we have:

Corollary 5.5.8. *It is undecidable, given an NSCM(2) M and a string w in $L(M)$, whether w is terminal with respect to $L(M)$.*

5.5.3 Deciding the given decomposition of a language

Finally, we consider the problem of deciding, given languages L, L_1, L_2, whether $L = L_1 \overline{\odot} L_2$.

Theorem 5.5.9.

1. *It is undecidable, given a language L accepted by a 1-reversal NCA and regular languages L_1 and L_2, whether $L = L_1 \overline{\odot} L_2$.*
2. *It is undecidable, given a regular language L and languages L_1 and L_2 accepted by 1-reversal DPDAs (resp., DCAs), whether $L = L_1 \overline{\odot} L_2$.*

Proof. For part 1, let $L \subseteq \Sigma^+$ be accepted by a 1-reversal NCA and let $L_1 = L_2 = \Sigma^+$. Hence, $L_1 \overline{\odot} L_2 = \Sigma^+$. The result follows, since it is undecidable whether the language accepted by a 1-reversal NCA is equal to Σ^+ (as seen in the proof of Theorem 5.5.2).

For part 2, let $L_1', L_2' \subseteq \Sigma^+$ be accepted by 1-reversal DPDAs (resp., DCAs). Let $\#, \$$ be new symbols not in Σ. Let $L = \{\$\$\}$, $L_1 = \# L_1' \$ \cup \{\$\$\}$, and $L_2 = \# L_2' \$ \cup \{\$\$\}$. Clearly, L_1 and L_2 can also be accepted by 1-reversal DPDAs (resp., DCAs). Then $L = L_1 \overline{\odot} L_2$ if and only if $L_1' \cap L_2' = \emptyset$. The result now follows since it is undecidable if the intersection of two languages accepted by 1-reversal DPDAs (resp., DCAs) is empty [17, 19]. □

In contrast to Theorem 5.5.9, part 2, when L is accepted by a deterministic machine we have the following result.

Theorem 5.5.10. *It is decidable, given a language L accepted by a DFCM (resp., DPCM) and regular languages L_1 and L_2, whether $L = L_1 \overline{\odot} L_2$.*

Proof. Clearly, $L_3 = L_1 \overline{\odot} L_2$ is regular. Now $L = L_1 \overline{\odot} L_2$ if and only if $L \cap L_3^c = \emptyset$, and $L^c \cap L_3 = \emptyset$. The result follows since the class of DFCM (resp., DPCM) languages is closed under intersection with regular sets and complementation, and has a decidable emptiness problem [19, 21]. □

Finally, in contrast to Theorem 5.5.9, when the problem concerns "containment", we have:

Theorem 5.5.11. *1. It is decidable, given languages L_1 and L_2 accepted by NFCMs and a language L accepted by a DPCM, whether $L_1 \overline{\odot} L_2 \subseteq L$.*

2. It is decidable, given a language L accepted by an NPCM and regular languages L_1 and L_2, whether $L \subseteq L_1 \overline{\odot} L_2$.

Proof. The claims follow from the following known results:

1. The family of DPCM (resp., regular) languages is closed under complementation [21].
2. The family of NFCM (resp., regular) languages is closed under the overlap operation $\overline{\odot}$ [7].
3. The family of NPCM (resp., DCM) languages is closed under intersection with NFCM languages [19].
4. The emptiness problem for NPCMs (resp., NFAs) is decidable [19].

□

5.6 Concluding remarks

This paper continues the exploration, started in [5] and [7], of the properties of the overlap assembly operation. In particular, it strengthens the results given in [5] regarding the closure of language classes under iterated overlap assembly and the decidability of the completeness of a language. It also enhances the results regarding closure properties of terminating sets of languages (which are almost equivalent to *maximal (adult) languages* in [3, 30]). Finally, it investigates the problem of deciding whether a given string is terminal with respect to a language, and the problem of deciding if a given language can be generated by an overlap assembly operation of two given others. Further directions of research include investigations of decision problems such as those studied in Section 5.5.3 for various other language classes, and finding an efficient algorithm that, given a language L, outputs a pair of languages (if they exist) whose overlap assembly equals L.

Bibliography

[1] B. S. Baker and R. V. Book. Reversal-bounded multipushdown machines. *Journal of Computer and System Sciences*, 8(3):315–332, 1974.

[2] P. Bonizzoni and N. Jonoska. Existence of constants in regular splicing languages. *Information and Computation*, 242:340–353, 2015.

[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 Proc. *Transgressive Computing,* TC, pages 216–228, 2006.

[5] E. Csuhaj-Varjú, I. Petre, and G. Vaszil. Self-assembly of strings and languages. *Theoretical Computer Science*, 374(1-3):74–81, 2007.

[6] A. R. Cukras, D. Faulhammer, R. J. Lipton, and L. F. Landweber. Chess games: a model for RNA based computation. *Biosystems*, 52(1-3):35–45, 1999.

[7] S. K. Enaganti, O. H. Ibarra, L. Kari, and S. Kopecki. On the overlap assembly of strings and languages. Submitted.

[8] S. K. Enaganti, L. Kari, and S. Kopecki. A formal language model of DNA polymerase activity. *Fundamenta Informaticae*, 138:179–192, 2015.

[9] D. Faulhammer, A. R. Cukras, R. J. Lipton, and L. F. Landweber. Molecular computation: RNA solutions to chess problems. *Proceedings of the National Academy of Sciences*, 97(4):1385–1389, 2000.

[10] G. Franco. A polymerase based algorithm for SAT. In M. Coppo, E. Lodi, and G. Pinna, editors, *Theoretical Computer Science*, volume 3701 of *LNCS*, pages 237–250. Springer Berlin Heidelberg, 2005.

[11] G. Franco, C. Giagulli, C. Laudanna, and V. Manca. DNA extraction by XPCR. In C. Ferretti, G. Mauri, and C. Zandron, editors, Proc. *DNA Computing,* (DNA 11), volume 3384 of *LNCS*, pages 104–112, 2005.

[12] G. Franco and V. Manca. Algorithmic applications of XPCR. *Natural Computing*, 10(2):805–819, 2011.

[13] G. Franco, V. Manca, C. Giagulli, and C. Laudanna. DNA recombination by XPCR. In A. Carbone and N. A. Pierce, editors, Proc. *DNA Computing,* (DNA 12), volume 3892 of *LNCS*, pages 55–66, 2006.

[14] S. Ginsburg, S. A. Greibach, and M. A. Harrison. One-way stack automata. *J. ACM*, 14(2):389–418, 1967.

[15] S. Ginsburg and G. F. Rose. The equivalence of stack-counter acceptors and quasi-realtime stack-counter acceptors. *Journal of Computer and System Sciences*, 8(2):243–269, 1974.

[16] S. A. Greibach. A note on the recognition of one counter languages. *ITA*, 9(2):5–12, 1975.

[17] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Inc., 1978.

[18] S. Hussini, L. Kari, and S. Konstantinidis. Coding properties of DNA languages. In N. Jonoska and N. C. Seeman, editors, Proc. *DNA Computing,* (DNA 7), volume 2340 of *LNCS*, pages 57–69, 2002.

[19] O. H. Ibarra. Reversal-bounded multicounter machines and their decision problems. *J. ACM*, 25(1):116–133, 1978.

[20] O. H. Ibarra. Automata with reversal-bounded counters: a survey. In Proc. *Descriptional Complexity of Formal Systems,* DCFS, pages 5–22, 2014.

[21] O. H. Ibarra and H.-C. Yen. On the containment and equivalence problems for two-way transducers. *Theoretical Computer Science*, 429(0):155–163, 2012.

[22] N. Immerman. Nondeterministic space is closed under complementation. *SIAM J. Comput.*, 17(5):935–938, 1988.

[23] P. D. Kaplan, Q. Ouyang, D. S. Thaler, and A. Libchaber. Parallel overlap assembly for the construction of computational DNA libraries. *Journal of Theoretical Biology*, 188(3):333–341, 1997.

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

[25] T. Kasami. A note on computing time for recognition of languages generated by linear grammars. *Information and Control*, 10(2):209–214, 1967.

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

[27] V. Manca and G. Franco. Computing by polymerase chain reaction. *Mathematical Biosciences*, 211(2):282–298, 2008.

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

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

[30] F. Manea, V. Mitrana, and J. Sempere. Some remarks on superposition based on Watson-Crick-like complementarity. In V. Diekert and D. Nowotka, editors, *Developments in Language Theory*, volume 5583 of *LNCS*, pages 372–383, 2009.

[31] C. Martín-Vide, G. Păun, J. Pazos, and A. Rodríguez-Patón. Tissue P systems. *Theoretical Computer Science*, 296(2):295–326, 2003.

[32] M. L. Minsky. Recursive unsolvability of Post's problem of "Tag" and other topics in theory of Turing machines. *The Annals of Mathematics*, 74(3):437–455, 1961.

[33] Q. Ouyang, P. D. Kaplan, S. Liu, and A. Libchaber. DNA solution of the maximal clique problem. *Science*, 278(5337):446–449, 1997.

[34] M.-P. Schützenberger. Sur certaines opérations de fermeture dans les langages rationnels. In *Symposia Mathematica*, volume 15, pages 245–253, 1975.

[35] W. P. Stemmer. DNA shuffling by random fragmentation and reassembly: in vitro recombination for molecular evolution. *Proceedings of the National Academy of Sciences*, 91(22):10747–10751, 1994.

[36] R. Szelepcsényi. The method of forced enumeration for nondeterministic automata. *Acta Informatica*, 26(3):279–284, 1988.

[37] V. V. Williams. Multiplying matrices faster than Coppersmith-Winograd. In Proc. *ACM Symposium on Theory of Computing,* STOC, pages 887–898, 2012.

Chapter 6

Conclusions

This thesis investigates two formal operations that model the action of the DNA Polymerase enzyme on DNA strands, and studies their properties. First, an overview of results on some well-studied formal language systems in the literature, based on operations inspired by enzymatic reactions on DNA is presented in Chapter 2. Although these bio-inspired operations are based on various enzymes such as restriction enzymes, ligases, polymerases etc., the focus of the original contributions of the current thesis is exclusively on modelling of the actions of the DNA polymerase enzyme.

The operation of "directed extension" is proposed, extended to languages and its closure properties for various classes of languages are studied in Chapter 3. The language equations involving the operation are studied, along with the definition of the inverse of the operation. Some relevant questions such as finding necessary and sufficient conditions for the existence of an inverse operation are answered, and a solution to finding an optimal inverse is given.

The operation of "overlap assembly" that was originally proposed by Csuhaj-varju, Petre and Vaszil is studied in Chapter 4. After settling closure properties of various language classes under this operation, some decision problems are solved. A theoretical analysis of how the iterated version of the operation can be useful in creating a DNA combinatorial library, known to have practical applications, is presented.

The study of the overlap assembly operation is continued in Chapter 5. This operation is compared with the related superposition operation, and it is shown how the positive closure properties of the iterated version of overlap assembly follow from their counterparts for the iterated superposition. Next, closure and related properties of the terminating sets are studied, and some decision problems are solved.

The operations studied in this thesis are exclusively inspired by the actions of the DNA polymerase enzyme. As a future work, it may be worthwhile to formulate formal operations that can model the actions of other enzymes such as restriction endonucleases, along with polymerases. As another future direction of study, it would be interesting to examine complexity problems for both operations described here. An example is finding lower and upper bounds for the complexity of the result language when the operation is applied between two initial languages of known complexity. Some complexity results of the iterated version of overlap assembly may also be interesting to address. It is possible that optimal algorithms to solve some well-known string problems, such as the shortest common super-string problem, may become easier to design and analyze using such results.

Chapter 7

Addendum

Since this thesis is formatted as integrated-article, all the technical chapters should contain exactly the same content of the published articles and no change is allowed. Therefore, we list the modifications implemented according to the comments provided by the thesis examiners as follows.

Implementation of the comments

- **page 50, lines 5-6:** "models the action of DNA Polymerase enzyme, an enzyme that plays a major role" is changed to "models the activity of DNA polymerase enzymes, enzymes that play a major role".

- **page 50, line 11 and page 51, lines 2, 3, 5, 10:** Grouped multiple citations in each of these locations.

- **page 51, lines 11-12:** "the action of the DNA Polymerase on DNA strands" is changed to "the extension activity of DNA polymerase enzymes on DNA strands".

- **page 54, equation after line 2:** $w \in \Sigma^+$ in the equation is replaced by $w \in \Sigma^*$.

- **page 55, line 2:** Added the sentence "Similarly, if $\mathscr{X}$ is REG and $\mathscr{Y}$ is LIN (CF), then the result $L_x \oplus L_y$ is in LIN (CF)."

- **page 64, line 6 after the equation:** "of the DNA Polymerase enzyme" is replaced by "of DNA polymerase enzymes".

Appendix A

Closure properties of overlap assembly

A.1 Notations

Throughout this appendix, all languages are considered to be defined over a fixed alphabet Σ. The sets $\mathrm{pref}(w)$, $\mathrm{inf}(w)$, and $\mathrm{suff}(w)$ contain, respectively, all prefixes, infixes, and suffixes of w. A prefix (resp., infix or suffix) x of w is *proper* if $x \neq w$. We employ the following notation: $\mathrm{Pref}(w) = \mathrm{pref}(w) \setminus \{w\}$, $\mathrm{Inf}(w) = \mathrm{inf}(w) \setminus \{w\}$, and $\mathrm{Suff}(w) = \mathrm{suff}(w) \setminus \{w\}$. This notation is naturally extended to languages; for example, $\mathrm{Suff}(L) = \bigcup_{w \in L} \mathrm{Suff}(w)$.

We use $\centerdot$ to represent catenation. Formally, $x \centerdot y = xy$ for all $x, y \in \Sigma^*$.

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. The complexity class $\mathrm{NSPACE}(f(n))$ (resp. $\mathrm{DSPACE}(f(n))$) is the set of decision problems that can be solved by a non-deterministic (resp. deterministic) Turing machine, M, using space $O(f(n))$, where $f(n)$ is the maximum number of tape cells that M scans on any input of length n. The complexity notations of O and Ω are used as usually defined in the literature.

A.2 Results

In this section we study closure properties of various language classes under overlap assembly. The next lemma expresses the overlap assembly operation in terms of concatenation, intersection, prefix and (infinite) union.

Lemma A.2.1. *If L_1 and L_2 are two languages over Σ, then*

$$L_1 \mathbin{\overline{\odot}} L_2 = \bigcup_{x \in L_1, y \in L_2} [x \cdot \mathrm{Suff}(y) \cap \mathrm{Pref}(x) \cdot y]$$

Proof. First, let $z \in L_1 \mathbin{\overline{\odot}} L_2$. Since $z = uvw$ where $uv \in L_1$ and $vw \in L_2$, there exists $x = uv \in L_1$ and $y = vw \in L_2$ such that $z \in x \mathbin{\overline{\odot}} y$. Since $w \in \mathrm{Suff}(y)$ and $u \in \mathrm{Pref}(x)$, we obtain that $z \in x \cdot \mathrm{Suff}(y)$ and $z \in \mathrm{Pref}(x) \cdot y$ which implies the result.

Conversely, let $z \in x \cdot \mathrm{Suff}(y) \cap \mathrm{Pref}(x) \cdot y$ for some $x \in L_1$ and $y \in L_2$. Let $u \in \mathrm{Pref}(x)$ such that $z = uy$ and let v be a string such that $uv = x$. Let $w \in \mathrm{Suff}(y)$ such that $z = xw$ and let v' be a string such that $v'w = y$. We thus have $z = uv'w = uvw$ which can be true only when $v = v'$. We then have $z = uvw$ where $uv = x \in L_1$ and $vw = y \in L_2$. Therefore, $z \in (L_1 \mathbin{\overline{\odot}} L_2)$.

□

Clearly, we can re-write the expression of $L_1 \mathbin{\overline{\odot}} L_2$ in Lemma A.2.1, as $\mathrm{Suff}(y)$ is common to both terms in the intersection:

$$L_1 \mathbin{\overline{\odot}} L_2 = \bigcup_{x \in L_1, y \in L_2} [x \cdot \Sigma^* \cap \mathrm{Pref}(x) \cdot y] = \bigcup_{x \in L_1} [x \cdot \Sigma^* \cap \mathrm{Pref}(x) \cdot L_2] \tag{A.1}$$

Symmetrically , it also follows that:

$$L_1 \mathbin{\overline{\odot}} L_2 = \bigcup_{x \in L_1, y \in L_2} [x \cdot \mathrm{Suff}(y) \cap \Sigma^* \cdot y] = \bigcup_{y \in L_2} [L_1 \cdot \mathrm{Suff}(y) \cap \Sigma^* \cdot y] \tag{A.2}$$

The following corollaries follow from the above expressions.

Corollary A.2.2. *The class of finite languages is closed under overlap assembly. If L_1 (resp. L_2) is a finite language, the result of $L_1 \overline{\odot} L_2$ will remain in the same class of Chomsky hierarchy as that of L_2 (resp. L_1).*

Proof. Let L_1 and L_2 be finite languages. By Lemma A.2.1, the overlap assembly

$$L_1 \overline{\odot} L_2 = \bigcup_{x \in L_1, y \in L_2} [x \centerdot \mathrm{Suff}(y) \cap \mathrm{Pref}(x) \centerdot y]$$

is a finite union of finite sets. Hence, the class of finite languages is closed under overlap assembly.

Let L_1 be a finite language and L_2 be a language in one of the classes of Chomsky hierarchy. From Equation (A.1), we have that

$$L_1 \overline{\odot} L_2 = \bigcup_{x \in L_1} [x \centerdot \Sigma^* \cap \mathrm{Pref}(x) \centerdot L_2].$$

The result of $L_1 \overline{\odot} L_2$ will be in the same class of that of L_2 because all classes of languages in the Chomsky hierarchy are closed under concatenation with a finite set, intersection with regular languages and finite union.

The same argument applies in case of L_2 being a finite language, by using Equation (A.2).

□

Corollary A.2.3. *The class of recursively enumerable languages is closed under overlap assembly. If one of either L_1 or L_2 is a recursively enumerable language, then $L_1 \overline{\odot} L_2$ is not necessarily a context-sensitive language.*

Proof. Consider two languages L_1 and L_2, both of whom are recursively enumerable. From Lemma A.2.1, we have

$$L_1 \overline{\odot} L_2 = \bigcup_{x \in L_1, y \in L_2} [x \centerdot \mathrm{Suff}(y) \cap \mathrm{Pref}(x) \centerdot y].$$

For any $x \in L_1$ and $y \in L_2$, the set $[x \centerdot \mathrm{Suff}(y) \cap \mathrm{Pref}(x) \centerdot y]$ is finite. The union of such sets over a countable (even if infinite) set will result in a countable set, and hence will be recursively enumerable. Hence, the set of all recursively enumerable languages is closed under overlap assembly.

Now we will prove that even if one of L_1 and L_2 is recursively enumerable and the other is finite, the result of $L_1 \overline{\odot} L_2$ can be recursively enumerable. Consider two languages L_1 and L_2 as follows:

$$L_1 = \{\#\$\}, \qquad\qquad L_2 = \{\$w \mid w \in L\}$$

where L is a recursively enumerable but not context-sensitive language whose alphabet does not contain the symbols # and \$. Clearly, the result of $L_1 \overline{\odot} L_2$ is $\{\#\$w \mid w \in L\}$ which is recursively enumerable but not context-sensitive.

Similarly, we can prove that if $L_1 = \{w\$ \mid w \in L\}$ and $L_2 = \{\$\#\}$ for a recursively enumerable but not context-sensitive language L, with no symbols # and \$, the result of $L_1 \overline{\odot} L_2$ will be $\{w\$\# \mid w \in L\}$ which is recursively enumerable but not context-sensitive.

□

Theorem A.2.4. *The family of regular languages is closed under overlap assembly.*

Proof. Let M_1 defined as $M_1 = \{S_1, \Sigma, \delta_1, s_0, \{s_f\}\}$ be a finite state automaton that accepts the regular language L_1 with the state set $S_1 = \{s_0, s_1, s_2, ..., s_n\}$ with initial state s_0 and final state s_f. We define $L_{1_{i,j}}$ to be the set of all strings that are generated between states s_i and s_j formally defined as $L_{1_{i,j}} = \{w \mid \delta_1^*(s_i, w) = s_j\}$. If L_1 and L_2 are regular languages, the result of the operation between L_1 and L_2 can be written as

$$L_1 \overline{\odot} L_2 = \bigcup_{\forall i: L_{1_{i,f}} \neq \emptyset} L_{1_{0,i}} \centerdot ([(L_{1_{i,f}} \backslash \{\lambda\}) \centerdot \Sigma^*] \cap L_2) \tag{A.3}$$

We will prove the expression in both the directions. Let $z \in L_1 \overline{\odot} L_2$. From Equation (A.1) we have,

$$z \in \bigcup_{x \in L_1, y \in L_2} (x \centerdot \Sigma^* \cap \mathrm{Pref}(x) \centerdot y)$$

which means $z \in x \centerdot \Sigma^*$ and $z \in \mathrm{Pref}(x) \centerdot y$ for some $x \in L_1$ and $y \in L_2$. Let $w \in \Sigma^*$ such that $z = xw$. Let $u \in \mathrm{Pref}(x)$ which would mean for some $v \neq \lambda$, we have $x = uv$. Hence, $z = uvw$ for some $w \in \Sigma^*$ where $uv = x$ for some $x \in L_1$ and $z = uy$ for $y \in L_2$. Since $uv = x \in L_1$, there is $i \leq n$ such that $u \in L1_{0,i}$ and $v \in L1_{i,f}$. Therefore, z belongs to $L1_{0,i} \centerdot (L1_{i,f} \backslash \{\lambda\}) \centerdot \Sigma^*$ and z belongs to $L1_{0,i} \centerdot L_2$. Hence, $z \in L1_{0,i} \centerdot ((L1_{i,f} \backslash \{\lambda\}) \centerdot \Sigma^* \cap L_2)$ which gives us the result.

Conversely, there exists z such that $z = uvw$ with $u \in L1_{0,i}$ for some $i \leq n$, $v \in L1_{i,f}$, $v \neq \lambda$, $w \in \Sigma^*$ and $y \in L_2$ such that $z = u \centerdot v \centerdot w$ and $z = u \centerdot y$. Clearly, $x = uv \in L_1$ and $y = vw \in L_2$. Therefore, $z \in x \overline{\odot} y \subseteq L_1 \overline{\odot} L_2$.

Since each of the sets $L1_{i,j}$ is regular and L_2 is regular, and regularity is preserved under concatenation, intersection and finite union, the language $L_1 \overline{\odot} L_2$ is regular. □

We give a corollary for the situation when exactly one of L_1 or L_2 is regular and the other is context-free or context-sensitive.

Corollary A.2.5. *(i) If L_1 is a regular language and L_2 is a context-free or context-sensitive language, then $L_1 \overline{\odot} L_2$ will be a context-free or context-sensitive language respectively. (ii) If L_2 is a regular language and L_1 is a context-free or context-sensitive language, then $L_1 \overline{\odot} L_2$ will be a context-free or context-sensitive language respectively.*

Proof. (i) Consider the Equation (A.3) which is still valid when L_1 is a regular language and L_2 is not. Let L_2 be a context-free or context-sensitive language.

We know that the class of context-free languages is closed under concatenation, intersection with regular language and finite union. The result of $L_1 \overline{\odot} L_2$ according to the expression in Equation (A.3) is context-free. The same argument applies for context-sensitive languages.

(ii) If L_2 is a regular language, we define $S_2 = \{s_0, s_1, s_2, ..., s_n\}$ to be the set of all states of the finite state automaton M_2 defined as $M_2 = \{S_2, \Sigma, \delta_2, s_0, \{s_f\}\}$ that accepts L_2 with s_0 being the initial state and s_f being the final state of M_2. Let $L2_{i,j}$ represent set of all strings generated by M_2 between states s_i and s_j formally defined as $L_{2_{i,j}} = \{w \mid \delta_2^*(s_i, w) = s_j\}$. We can derive an expression for $L_1 \,\overline{\odot}\, L_2$ similar to Equation (A.3) and use Equation A.2 to show that:

$$L_1 \,\overline{\odot}\, L_2 = \bigcup_{\forall i : L_{2_{0,i}} \neq \emptyset} (L_1 \cap [\Sigma^* \centerdot (L_{2_{0,i}} \backslash \{\lambda\})]) \centerdot L_{2_{i,f}}$$

We know that the class of context-free languages is closed under concatenation, intersection with regular language and finite union. Hence, the result of $L_1 \,\overline{\odot}\, L_2$ is also context-free. The same argument applies to context-sensitive languages. □

Theorem A.2.6. *Let L_1 and L_2 be context-free (resp. linear) languages. Then the language $L_1 \,\overline{\odot}\, L_2$ is context-sensitive, but not necessarily context-free (resp. linear).*

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

$$L_1 = \{a^n\$b^n\# \mid n \geq 1\}, \qquad L_2 = \{\$b^n\#c^n \mid n \geq 1\}.$$

We can easily see that the result of overlap assembly of L_1 with L_2 yields the context-sensitive but not context-free language

$$L_1 \,\overline{\odot}\, L_2 = \{a^n\$b^n\#c^n \mid n \geq 1\}.$$

The fact that $L_1 \,\overline{\odot}\, L_2$ will be in CS for all possible context-free (resp. linear) languages L_1 and L_2 is established in Theorem A.2.7. □

The next result shows that $L_1 \,\overline{\odot}\, L_2$ is context-sensitive for $L_1 \in$ CF and $L_2 \in$ CF.

Theorem A.2.7. *Let $f(n) \in \Omega(\log n)$ be a monotone function. The classes* NSPACE$(f(n))$ *and* DSPACE$(f(n))$ *are each closed under overlap assembly.*

Proof. First, consider two NSPACE$(f(n))$ languages L_1 and L_2 where $f(n) \in \Omega(\log n)$ is a monotone function. Let M_1 and M_2 be the Turing machines that decide the language L_1 and L_2, respectively, in NSPACE$(f(n))$. We define two procedures:

- $Len(Input)$ returns the length of the input.

- $Simulate(M,I,J)$ simulates the Turing machine M with the input in between (and including) markers I and J of the input and returns "accept"or "reject"according to the output of the simulation.

Let M_z be the Turing machine that non-deterministically guesses two positive integers I,J such that $1 \le I \le J \le Len(Input)$ and accepts only if $Simulate(M_1,1,J)$ and $Simulate(M_2,I,$ $Len(Input))$ both return "accept". Since an input word w belongs to $L_1 \overline{\odot} L_2$ if and only if there is a prefix x and a suffix w of y such that x and y overlap, the Turing machine M_z decides $L_1 \overline{\odot} L_2$. Note that M_1 and M_2 can be simulated on prefixes and suffixes of the input word w using at most $Z(|w|)$ space, because Z is monotone. Additionally, the markers I and J require $\log(|w|)$ space in binary encoding. We conclude the algorithm works in NSPACE$(f(n))$.

Now, consider that L_1 and L_2 are decided by deterministic Turing machines M_1 and M_2, respectively, in DSPACE$(f(n))$. Algorithm 2 determines the longest prefix x of the input w which belongs to L_1, and then, attempts to find the longest suffix y of w that belongs to L_2 and overlaps with x; it accepts if and only if such a suffix y exists. Clearly, this algorithm works in DSPACE$(f(n))$ and decides $L_1 \overline{\odot} L_2$.

Algorithm 2

```
$J := Len(Input)$;
while $J \ge 1$ and $Simulate(M_1,1,J)$ = "reject" do
    $J := J - 1$;
end while
$I := 1$;
while $I \le J$ and $Simulate(M_2,I,Length(Input))$ = "reject" do
    $I := I + 1$;
end while

if $I \le J$ then
    return "accept";
else
    return "reject";
end if
```

□

Corollary A.2.8. *The family of recursive languages is closed under overlap assembly.*

Proof. The proof of theorem A.2.7 applies with the restriction on space taken by the working-tape being lifted. □

In Table A.1 we summarize the results from this section. For two language classes $\mathscr{X}$ and $\mathscr{Y}$, it shows the language class $\mathscr{Z}$ from the Chomsky hierarchy such that for all $L_1 \in \mathscr{X}$ and $L_2 \in \mathscr{Y}$ we have $L_1 \overline{\odot} L_2 \in \mathscr{Z}$. This shows that all entries in Table A.1 can also be considered “lower bounds” for the language class $\mathscr{Z}$.

$L_1 \backslash L_2$	FIN	REG	CF	CS	RE
FIN	FIN Corr A.2.2	REG Corr A.2.2	CF Corr A.2.2	CS Corr A.2.2	RE Corr A.2.2, A.2.3
REG	REG Corr A.2.2	REG Thm A.2.4	CF Corr A.2.5	CS Corr A.2.5	RE Corr A.2.3
CF	CF Corr A.2.2	CF Corr A.2.5	CS Thm A.2.6	CS Thm A.2.7	RE Corr A.2.3
CS	CS Corr A.2.2	CS Corr A.2.5	CS Thm A.2.7	CS Thm A.2.7	RE Corr A.2.3
RE	RE Corr A.2.2, A.2.3	RE Corr A.2.3	RE Corr A.2.3	RE Corr A.2.3	RE Corr A.2.3

Table A.1: Summary of closure properties: each entry shows which language class $L_1 \overline{\odot} L_2$ belongs to if L_1 is from the corresponding language class in the left column and L_2 is from the corresponding language class in the top row.

Appendix B

Copyright releases

The contents of Chapter 3 were published in the journal “Fundamenta Informaticae” by IOS press. I requested them to grant me permission to reuse the article for my thesis and I got their kind permission from them through the following email.

```
From: Carry Koolbergen <C.Koolbergen@iospress.nl>
Date: Tue 2015-07-07 8:47 AM

To:  Srujan Kumar Enaganti;



Dear Srujan Kumar Enaganti,

We hereby grant you permission to reproduce the below mentioned material in print
and electronic format at no charge subject to the following conditions:

1. If any part of the material to be used (for example, figures) has appeared
in our publication with credit or acknowledgement to another source, permission
must also be sought from that source.  If such permission is not obtained then
```

that material may not be included in your publication/copies.

2. Suitable acknowledgement to the source must be made, either as a footnote or in a reference list at the end of your publication, as follows:

‘‘Reprinted from Publication title, Vol number, Author(s), Title of article, Pages No., Copyright (Year), with permission from IOS Press’’.

3. This permission is granted for non-exclusive world English rights only. For other languages please reapply separately for each one required.

4. Reproduction of this material is confined to the purpose for which permission is hereby given.

Yours sincerely

Carry Koolbergen (Mrs.)
Contracts, Rights & Permissions Coordinator
Not in the office on Wednesdays

IOS Press BV
Nieuwe Hemweg 6B
1013 BG Amsterdam
The Netherlands

Tel.: +31 (0)20 687 0022

```
Fax: +31 (0)20 687 0019

Email: [c.koolbergen@iospress.nl]c.koolbergen@iospress.nl /

[publisher@iospress.nl]publisher@iospress.nl


URL: [www.iospress.nl]www.iospress.nl

Twitter: @IOSPress_STM

G+: IospressSTM

Facebook: publisheriospress


P Please consider the environment before printing this email.


Van: Srujan Kumar Enaganti [mailto:senagant@uwo.ca]

Verzonden: maandag 6 juli 2015 19:41

Aan: Carry Koolbergen

Onderwerp: Copyright Access



Hello,


I am an author of one of the papers you published in the journal

Fundamenta Informaticae. The link to the paper is:

http://ip.ios.semcs.net/articles/fundamenta-informaticae/fi138-1-2-14


Can I use the paper to publish my PhD thesis through my University?


Please let me know if you retain the copyright rights.
```

Thanks very much.

Warm regards,

--

Srujan Kumar Enaganti

Curriculum Vitae

Name:	Srujan Kumar Enaganti
Education and Degrees:	2002 - 2006 B.Tech. Indian Institute of Technology Guwahati Guwahati, India
	2008 - 2010 M.Sc. University of British Columbia Vancouver, BC, Canada
	2011 - 2015 Ph.D. University of Western Ontario London, ON, Canada
Related Work Experience:	Junior Research Associate Infosys Technologies Limited 2006 - 2008
	Teaching Assistant The University of British Columbia 2008 - 2010
	Research Engineer University of Pau and Pays de l'Adour 2010 - 2011
	Teaching Assistant The University of Western Ontario 2011 - 2015

Publications:

1. Srujan Kumar Enaganti, Lila Kari, Steffen Kopecki: A Formal Language Model of DNA Polymerase Enzymatic Activity. *Fundamenta Informaticae* 138(1-2): 179-192 (2015)
2. Srujan Kumar Enaganti, Anish Damodaran, Anirban Chakrabarti: A Framework for Analysis of Legacy Code Migration to Grid Environment. *CoreGRID* 2007: 215-224

Publications under review:

1. Srujan Kumar Enaganti, Oscar H. Ibarra, Lila Kari, Steffen Kopecki: On the overlap assembly of strings and languages.(Submitted)
2. Srujan Kumar Enaganti, Oscar H. Ibarra, Lila Kari, Steffen Kopecki: Further remarks on DNA overlap assembly.(Submitted)