EPTCS 407

Proceedings of the

# 14th International Workshop on Non-Classical Models of Automata and Applications

**Göttingen, Germany, 12-13 August 2024**

Edited by: Florin Manea and Giovanni Pighizzini

Published: 11th September 2024
DOI: 10.4204/EPTCS.407
ISSN: 2075-2180
Open Publishing Association

# Table of Contents

# Preface

Florin Manea
University of Göttingen, Germany
florin.manea@cs.uni-goettingen.de

Giovanni Pighizzini
Università degli Studi di Milano
pighizzini@di.unimi.it

The Fourteenth International Workshop on Non-Classical Models of Automata and Applications (NCMA 2024) was held in Göttingen, Germany, on August 12 and 13, 2024, at the historic Georg-Augustus-Universität, organized by the Theoretical Computer Science research group of the respective university. The NCMA workshop series was established in 2009 as an annual event for researchers working on non-classical and classical models of automata, grammars or related devices. Such models are investigated both as theoretical models and as formal models for applications from various points of view. The goal of the NCMA workshop series is to exchange and develop novel ideas in order to gain deeper and interdisciplinary coverage of this particular area that may foster new insights and substantial progress.

The previous NCMA workshops took place in Wrocław, Poland (2009), Jena, Germany (2010), Milano, Italy (2011), Fribourg, Switzerland (2012), Umeå, Sweden (2013), Kassel, Germany (2014), Porto, Portugal (2015), Debrecen, Hungary (2016), Prague, Czech Republic (2017), Koăice, Slovakia (2018), Valencia, Spain (2019). Due to the Covid-19 pandemic there was no NCMA workshop in 2020 and 2021. After that, the series returned to Debrecen (2022) and then continued in Famagusta, North Cyprus (2023).

The Fourteenth edition, in Göttingen, Germany, was co-located with the 28th International Conference on Developments in Language Theory (DLT 2024, 12-16 August).

The invited lectures at NCMA 2024 have been the following:

- Martin Kutrib (Gießen, Germany): Cellular Automata: From Black-and-White to High Gloss Color (joint invited lecture with DLT 2024)
- Robert Mercaş (Loughborough, UK): Complexities of One-way Jumping Finite Automata

The 13 regular contributions have been selected out of 17 submissions by a total of 38 authors from 10 different countries by the following members of the Program Committee:

- Marcella Anselmo (Salerno, Italy)
- Péter Battyányi (Debrecen, Hungary)
- Martin Berglund (Umeå, Sweden)
- Erzsébet Csuhaj-Varjú (Budapest, Hungary)
- Joel Day (Loughborough, UK)
- Pamela Fleischmann (Kiel, Germany)
- Tore Koß(Göttingen, Germany)
- Zbyněk Křivka (Brno, Czech Republic)
- Andreas Malcher (Gießen, Germany)
- Florin Manea (Göttingen, Germany, chair)

- Victor Mitrana (Madrid, Spain)
- Giovanni Pighizzini (Milano, Italy, chair)
- Dana Pardubska (Bratislava, Slovak Republic)
- Luca Prigioniero (Loughborough, UK)
- Stefan Siemer (Göttingen, Germany)
- Bianca Truthe (Gießen, Germany)
- Brink Van Der Merwe (Stellenbosch, South Africa)
- Petra Wolf (Bordeaux, France)

A special issue of the Journal of Automata, Languages and Combinatorics containing extended versions of selected contributions to NCMA 2024 will also be edited after the workshop. The extended papers will undergo the standard refereeing process of the journal.

We are grateful to the two invited speakers, to all authors who submitted a paper to NCMA 2024, to all members of the Program Committee, their colleagues who helped evaluating the submissions, and to the members of the Theoretical Computer Science research group of the University of Göttingen who were involved in the local organization of NCMA 2024.

August 2024

Florin Manea
Giovanni Pighizzini

# Complexities of One-way Jumping Finite Automata

Szilárd Zsolt Fazekas
Akita University, Department of Mathematical Science and Electrical-Electronic-Computer Engineering
szilard.fazekas@ie.akita-u.ac.jp

Robert Mercaş  Luca Prigioniero
Loughborough University, Department of Computer Science
R.G.Mercas@lboro.ac.uk  L.Prigioniero@lboro.ac.uk

Words are an integral part of computer science, and their classical sequential processing is reflected by machines accepting various classes of languages of the Chomsky hierarchy. However, since already the 70's there has been keen interest, when considering words or languages of words, on presentations of these sequences in the form of nonconsecutive symbols, i.e., [17, 13]. Within last thirty years, non-classical models of automata have captured more and more attention, with the introduction of several such models, i.e., restarting automata [11], jumping automata [14], input revolving automata [4] and automata with translucent letters [16].

One-way jumping finite automata (OWJFA) were introduced [5] as a non-sequential processing model of deterministic finite automata (DFA) that is more restrictive, with respect to the extra operations, than the general jumping automata introduced in [14]. The former two coincide in fact in the case of complete DFAs, since the extra capabilities of the OWJFA stem precisely from the missing transitions on any state in the classical DFA. At the same time, due to their restrictive definition of 'jumping' that is only possible in the previously described case of a sequential read, they represent a natural restriction of the model defined in [14]. To this end, they also represent a restriction of the (right) revolving automata [4], where the revolving operation maintains the state it starts in (see [5]).

In a nutshell, OWJFAs are specified in exactly the same way as DFAs while allowing partial transition functions, see Fig. 1. The only behavioural difference is when arriving at a tape symbol for which the current state has no defined outgoing transition. In the classical case such inputs are rejected, in the jumping mode these are irrelevant since we can jump anywhere on the tape, while in the revolving case the only possibility is for a revolving operation to be applied, should one such operation be specified for the current state and symbol. In the one-way jumping mode these symbols are skipped temporarily, but must be processed later, i.e., the relative order of the skipped symbols is maintained, with the automaton moving back to the beginning after each pass (called *sweeps*, see Fig. 2), and seeing only the symbols previously skipped. Therefore one can also view this model as a type of DFA with a circular input tape,

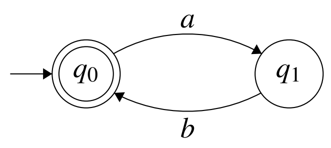


Figure 1: OWJFA $\mathscr{A}$ accepting the language $L(\mathscr{A}) = \{w \in \{a,b\}^* \mid |w|_a = |w|_b\}$

| *position* : | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| input | **a** | **a** | **a** | **a** | **b** | **b** | **a** | **b** | **b** | **b** |
| after sweep 1 | $\varepsilon$ | **a** | **a** | **a** | $\varepsilon$ | **b** | $\varepsilon$ | $\varepsilon$ | **b** | **b** |
| after sweep 2 | $\varepsilon$ | $\varepsilon$ | **a** | **a** | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | **b** | **b** |
| after sweep 3 | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | **a** | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | **b** |
| after sweep 4 | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ | $\varepsilon$ |

Figure 2: Computation for *aaaabbabbb* by OWJFA $\mathscr{A}$

Giovanni Pighizzini, Florin Manea (Eds.): NCMA 2024
EPTCS 407, 2024, pp. 1–4, doi:10.4204/EPTCS.407.1

always jumping clockwise, reading and consuming the nearest letter for which it has a defined transition from the current state.

While [5, 1] investigate various properties of the accepted language class, the decidability questions that arise from these have been settled [2]. Considering the one-way jumping processing mode, authors also investigated more powerful machines corresponding to the classical models, i.e., nondeterministic finite automata [3, 6], two-way finite automata [7], pushdown automata and linear bounded automata [6]. While the language classes defined by the models have no nontrivial closure properties under usual language operations, the accepting power and decidability issues raised some intriguing problems.

The above machine models are more powerful when the one-way jumping mode is present in all cases, except for linear bounded automata. While this is rather natural and showing it follows classical techniques, the challenge was to figure just how powerful the new processing mode is, even in the simplest of cases, when DFAs are considered. Since a complete transition function renders a DFA, i.e., no symbols are skipped, the class of languages accepted by DFA in one-way jumping mode trivially includes all regular languages. The language class defined by OWJFAs is incomparable with the context-free class, but included in the context-sensitive class and in DTIME($n^2$) [1]. The separation results make use of combinations of a handful of regular languages together with a very simple type of non-regular languages which contain words having letter counts in a certain ratio, e.g., the frequently used $L_{ab} = \{w \in \{a,b\}^* \mid w \text{ contains as many } a\text{'s as } b\text{'s}\}$ accepted by the machine $\mathscr{A}$ in Fig. 1. While this was enough to establish virtually all separations of interest, it left a significant gap in our understanding of the model when it comes to acceptance of non-regular languages that do not establish linear relationships among letter counts.

Moreover, of high interest, in most cases, is the study of the effect that the non-sequential way of processing inputs has on the various complexity aspects. In terms of computation complexity theory, [9] considers the impact of the needed jumps or sweeps when looking at the computation of a machine. The aim was to arrive at a decision procedure on whether a given machine accepts a regular language. Part of this is the introduction of sweep complexity, a measure for the number of times, in the worst case, that such machines have to return to the beginning of their input after having skipped some of the symbols. The consideration was that sweep count can represent a measure of non-regular resources used by a machine, posing the natural question of how much of this resource is needed to be able to accept non-regular languages? The idea of sweep complexity appears in other contexts, too, and an interesting and thorough investigation of a similar flavour established infinite hierarchies in terms of sweep count for iterated uniform finite transducers [12]. Even more recently, the notion of jump complexity appears in the context of automata with translucent letters [15].

It was known that constant sweep complexity does not increase the accepting power of the machines [10], and in [8] the authors refute the conjecture that, in fact, any automaton with higher than constant sweep complexity accepts a non-regular language. Moreover, the same work shows that, in general, the sweep complexity of an automaton does not distinguish between accepting regular and non-regular languages, and provides a separation result for asymptotic classes defined by this complexity measure. Thus language classes defined through asymptotic complexity form a true hierarchy, i.e., there are languages which can be accepted by a machine with $\mathscr{O}(f(n))$ sweep complexity but not by any with $o(f(n))$ sweep complexity, for various functions $f(n)$.

One of the main benefits that OWJFAs seem to exhibit, in comparison with similar classical ones, is the size of the machine when considering the accepted languages. While already from [5] it was established that there is no unique minimal machine describing a certain language, when talking about regular languages, we were able to show some descriptional complexity results with respect to these machines. In particular, we showed that while there might still be a, tight, exponential blow-up in the number of states needed to represent an NFA with the help of an OWJFA, contrary to the classical case, there are situations where such an exponential blow-up exists in the other direction as well, even for 3-letter alphabets. For the deterministic machines, since OWJFAs are presented in the form of DFAs, only one direction is meaningful, and we were able to show that we get an exponential blow-up when for a regular language accepted by OWJFAs we want a description in the form of a DFA.

Based on all of the previous discussion there are several questions that arise when considering models with such operations. These range from the analysis of the hierarchy in the case of computational complexity with respect to the number of sweeps that words from a language require, to the a descriptional complexity analysis in the case of nondeteministic machines.

## References

[1] Simon Beier & Markus Holzer (2019): *Properties of right one-way jumping finite automata*. Theoretical Computer Science 798, pp. 78 – 94, doi:10.1016/j.tcs.2019.03.044.

[2] Simon Beier & Markus Holzer (2020): *Decidability of Right One-Way Jumping Finite Automata*. International Journal of Foundations of Computer Science 31(06), pp. 805–825, doi:10.1142/S0129054120410063.

[3] Simon Beier & Markus Holzer (2022): *Nondeterministic right one-way jumping finite automata*. Information and Computation 284, p. 104687, doi:10.1016/j.ic.2021.104687.

[4] Henning Bordihn, Markus Holzer & Martin Kutrib (2005): *Revolving-Input Finite Automata*. In Clelia de Felice & Antonio Restivo, editors: *DLT 2005*, *LNCS* 3572, pp. 168–179, doi:10.1007/11505877_15.

[5] Hiroyuki Chigahara, Szilárd Zsolt Fazekas & Akihiro Yamamura (2016): *One-Way Jumping Finite Automata*. International Journal of Foundations of Computer Science 27(3), pp. 391–405, doi:10.1142/S0129054116400165.

[6] Szilárd Zsolt Fazekas, Kaito Hoshi & Akihiro Yamamura (2021): *The effect of jumping modes on various automata models*. Natural Computing, doi:10.1007/s11047-021-09844-4.

[7] Szilárd Zsolt Fazekas, Kaito Hoshi & Akihiro Yamamura (2021): *Two-way deterministic automata with jumping mode*. Theoretical Computer Science 864, pp. 92–102, doi:10.1016/j.tcs.2021.02.030.

[8] Szilárd Zsolt Fazekas & Robert Mercaş (2023): *Sweep Complexity Revisited*. In Benedek Nagy, editor: *CIAA*, *LNCS* 14151, Springer, pp. 116–127, doi:10.1007/978-3-031-40247-0_8.

[9] Szilárd Zsolt Fazekas, Robert Mercaş & Olivia Wu (2022): *Complexities for Jumps and Sweeps*. Journal of Automata, Languages and Combinatorics 27(1-3), pp. 131–149, doi:10.25596/jalc-2022-131.

[10] Szilárd Zsolt Fazekas & Akihiro Yamamura (2016): *On regular languages accepted by one-way jumping finite automata*. In: NCMA, short papers, pp. 7–14.

[11] Petr Jančar, František Mráz, Martin Plátek & Jörg Vogel (1995): *Restarting automata*. In Horst Reichel, editor: Fundamentals of Computation Theory, pp. 283–292, doi:10.1007/3-540-60249-6_60.

[12] Martin Kutrib, Andreas Malcher, Carlo Mereghetti & Beatrice Palano (2022): *Descriptional complexity of iterated uniform finite-state transducers*. Information and Computation 284, p. 104691, doi:10.1016/j.ic.2021.104691.

[13] David Maier (1978): *The Complexity of Some Problems on Subsequences and Supersequences*. Journal of the ACM 25(2), pp. 322–336, doi:10.1145/322063.322075.

[14] Alexander Meduna & Petr Zemek (2012): *Jumping Finite Automata*. International Journal of Foundations of Computer Science 23(7), pp. 1555–1578, doi:10.1142/S0129054112500244.

[15] Victor Mitrana, Andrei Paun, Mihaela Paun & José-Ramón Sánchez-Couso (2024): *Jump complexity of finite automata with translucent letters*. Theoretical Computer Science 992, p. 114450, doi:10.1016/J.TCS.2024.114450.

[16] Benedek Nagy & Friedrich Otto (2011): *Finite-state acceptors with translucent letters*. In: *ICAART 2011*, pp. 3–13.

[17] Imre Simon (1972): *Hierarchies of events with dot-depth one - Ph.D. thesis*. University of Waterloo.

# A New Notion of Regularity: Finite State Automata Accepting Graphs

Yvo Ad Meeres
Department of Theoretical Computer Science
University of Bremen
Bremen, Germany
yvo.meeres@mailbox.org

Analogous to regular string and tree languages, regular languages of directed acyclic graphs (DAGs) are defined in the literature. Although called regular, those DAG-languages are more powerful and, consequently, standard problems have a higher complexity than in the string case. Top-down as well as bottom-up deterministic DAG languages are subclasses of the regular DAG languages. We refine this hierarchy by providing a weaker subclass of the deterministic DAG languages. For a DAG grammar generating a language in this new DAG language class, or, equivalently, a DAG-automaton recognizing it, a classical deterministic finite state automaton (DFA) can be constructed. As the main result, we provide a characterization of this class.

The motivation behind this is the transfer of techniques for regular string languages to graphs. Trivially, our restricted DAG language class is closed under union and intersection. This permits the application of minimization and hyper-minimization algorithms known for DFAs. This alternative notion of regularity coins at the existence of a DFA for recognizing a DAG language.

## 1 Introduction

Many research fields either struggle with the complexity of processing graphs – for example fields like high performance computing [19] or neurocomputing [6], to mention just a few – or by encoding their graph problems as strings, see e.g. [16, 11]. The well-researched class of regular string languages, recognized by finite state automata (FSAs), exhibits a fruitful balance between expressiveness and efficiency concerning standard algorithmic problems. The problem is, that these algorithms are only applicable to strings. One approach would be to provide efficient graph algorithms for specific problems, as in [25] for the membership problem circumventing Braess's Paradox [7] or in [4] providing a faster algorithm for the very specific problem of the maximum independent set on interval filament graphs. Instead of the cumbersome approach to tackle all these specific problems one by one, our meta-approach suggests porting all efficient algorithms known for string processing to graph processing in one sweep. To port a wide range of well-known efficient algorithms (based on FSAs) from strings to graphs, this article introduces FSAs recognizing sets of directed acyclic graphs (DAGs) instead of sets of strings. Such sets are called DAG languages. We consider *vertex-labeled DAGs* with unlabeled edges but label those edges for accepting a DAG. A classical FSA accepts a string by reading it symbol by symbol. Our proposed FSA accepts a DAG by reading top-down vertex by vertex instead. A symbol read by the automaton encodes a whole vertex, consisting of its vertex label and its ordered and labeled in- and outgoing edges. While reading the vertices top-down, the outgoing edges in a DAG are labeled according to the automaton's specification while the ingoing edges, labeled beforehand, have to match. The FSA's states keep track of those ingoing edge labels whose target vertices are not yet read, thus whose outgoing edges are unlabeled. The class of DAG languages accepted by such an FSA is a proper subset of the top-down

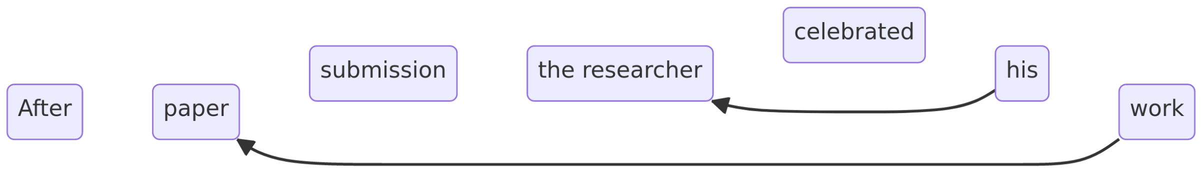


Figure 1: Classical NLP parsing is blind to coreferences within sentences, since trees cannot represent these edges within the parse tree. Graphs, on the contrary, are capable of showing coreferences between e.g. words of parsed sentences. For the above sentence, its parse trees could neither model the obvious possessive relation between the possessive pronoun *his* and *the researcher* nor the semantic kind of equivalence relation requiring world knowledge between the *paper* and the *work*. But, a semantic graph like an `AMR` DAG [29] could. The capabilities of semantic graphs are illustrated in [13] as well as for a complex sentence in [12] by means of the representation of a sentence as an `AMR` DAG.

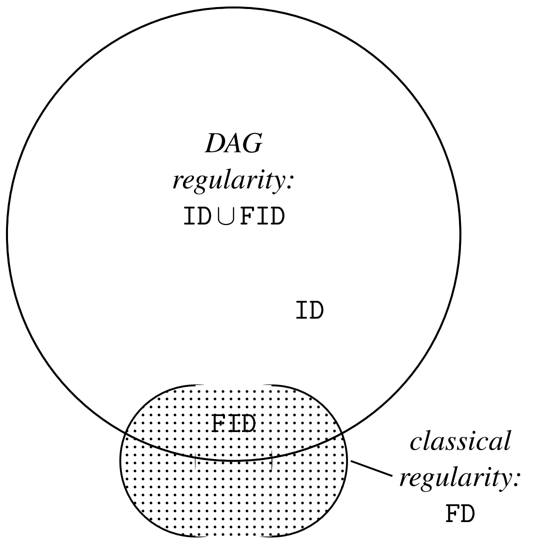
deterministic regular DAG languages defined in the literature [5]. This class, in turn, is a proper subset of the regular DAG languages [5].

In literature, the notion of regularity concerning DAG languages differs from that applied to string languages. Regular DAG automata recognize regular DAG languages [5]. These automata are one of the formalisms [12] proposed in the literature to model semantics by using Abstract Meaning Representation (`AMR`) [3]. DAG automata were originally introduced by Kamimura and Slutzki [21, 22]. A promising alternative formalism, not considered in this paper, is the hyperedge replacement graph grammar [20]. Classical natural language parsing turns a sentence into a parse tree while semantic parsing, such as `AMR` parsers [9, 10, 30], can model coreferences between e.g. the words of a sentence. Fig. 1 shows two such relations which turn a parse tree into a DAG. Such semantic relations, expressible in `AMR`, specify that the work was conducted by the mentioned researcher (*researcher* $\leftarrow$ *his*) and writing the paper is the researcher's daily work (*work* $\rightarrow$ *paper*). Semantic parsing provides vital contributions to improve natural language processing (NLP). The anecdotes about AI chat bots inventing facts feed wishes for NLP improvements by ensuring semantic consistency. This consistency is desirable also for sophisticated spell and grammar checking and machine translations.

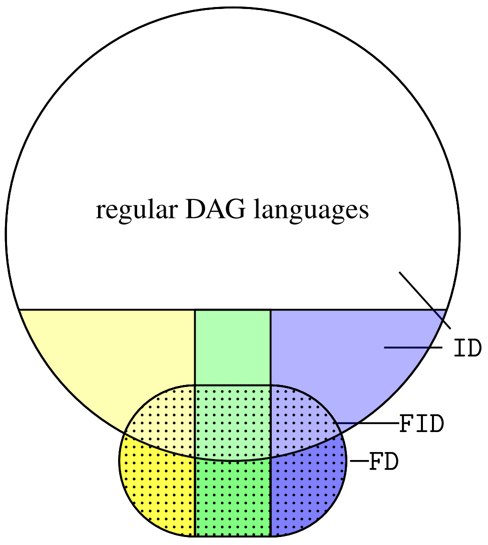
The membership problem for regular DAG automata surprisingly being NP-complete [8, 18], the uniform and even the non-uniform one[1], finding strategies for identifying efficient semantic parsing algorithms remains the core problem. By determinizing, either top-down or bottom-up, the membership problem becomes tractable. However, even with restrictions like determinism or planarity [21, 32], parsing problems can easily become too complex. For instance, Vasiljeva et al. were surprised that for certain probabilistic DAG automata non-trivial probability distributions are necessary to assign weights [32]. Since the notions of regularity differ for the string and DAG case, we propose the *new notion of regularity*: use the string case regularity for DAGs in order to obtain better algorithmic properties. The regularity notion for DAGs, presented in literature, seems to be too powerful to provide efficient algorithms. Viewing DAG languages only then as regular when they can be recognized by an FSA, provides deep insight into the structural properties of DAG languages.

Although the mildly context sensitive upper bound for natural language parsing classifies parsing as lying between context-free and context-sensitive formalisms, finite state descriptions of languages are of

[1]The uniform membership problem asks for a given automaton and a given graph whether the graph is an element of the given language; the non-uniform membership problem asks for a fixed automaton and a given graph whether the automaton accepts the graph, making the membership problem potentially easier.



(a) Classical and DAG regularity



(b) DAG language classes

Figure 2: *Overview over the language classes*
*In both Venn diagrams, the circle denotes the the regular DAG languages whereas the oval denotes the language class* `FD`*. The intersection between the two is the class* `FID` *which is both closed under edge swap as well as under DFA-construction. The dotted part, the oval, corresponds to* `FD`*. The non-dotted part corresponds to* `ID`*.*
*(a) Classically, the term* regularity *refers to FSAs and thus to string languages. This does not match the notion of regularity for DAG languages. The two notions match only for languages in* `FID`*.*
*(b) Top-down determinism and bottom-up determinism are colored in yellow and blue. Consequently, green stands for languages which are both top-down as well as bottom-up deterministic. In the right Venn diagram, whereas all colored fields are deterministic, the nondeterministic part corresponds to the white part The class* `ID` *comprises those regular DAG languages which are not in* `FID` *(and consequently not in* `FD`*), and which are either (top-down / bottom-up) deterministic or non-deterministic.*

major importance [1]. For the sake of efficiency, this field often seeks to digest also mildly context sensitive structures with finite state methods [27, 23, 28], an approach conceivable also for DAG digestion. The conference *Finite State Methods in Natural Language Processing* (FSMNLP) concentrates on this lowest level of the Chomsky hierarchy. Many NLP tools, like `apertium`, `HFST` or `GiellaLT` [17, 31, 26] operate with finite state descriptions.

All this said, our contributions, see them illustrated in Fig. 2, can be stated as

- the idea of using a classical FSA to recognize a DAG language
- the notion of a *meta-state*, a multiset of edge labels, serving as the states of the FSA
- separation of the regular top-down deterministic DAG languages into those recognizable (`FID`) and those not recognizable (`ID`) via an FSA by means of the meta-state technique
- restricting a DAG automaton by meta-states, resulting in the class `FD` comprising `FID` but not being a subset of the regular DAG languages
- characterization of the newly defined classes (*main result*).

Providing an FSA for a DAG language immediately opens up a wide range of results for DAGs formerly applicable only to strings. The folklore algorithm of FSA minimization can be applied, just as algorithms for lossy FSA compression, called hyper-minimization [2, 24], where *hyper* stands for the tolerance of finitely many errors. Morphological transducers already prove hyper-minimizations being useful for NLP [14].

Also from a structural point of view a deeper understanding of DAG language classes and a suitable overall hierarchy would be a beautiful result for theoretical computer science. As mentioned in the beginning, many research fields require efficient graph algorithms and therefore could potentially profit from these very limited DAG languages since they are parsable as efficiently as regular string languages. Even though their expressiveness is quite limited, implying that we cannot encode arbitrary graph languages, for a variety of important problems, the limited expressiveness will suffice, and algorithms can be ported directly from the string case.

## 2 Preliminaries

The set of non-negative integers is denoted by $\mathbb{N}$. For $n \in \mathbb{N}$ we define $[n] = \{1, \ldots, n\}$. For a set $A$, we denote its cardinality by $|A|$. A finite set $A$ is called an *alphabet*, an element $a \in A$ is a *symbol*, a *string* is the concatenation of symbols, the set of all finite strings over $A$ is denoted by $A^*$ and a, not necessarily proper, subset of $A^*$ is called a *language*. The empty string of length 0 is denoted by $\lambda$. The length of a string $w \in A^*$ is denoted by $|w|$ and $[w]$ denotes the smallest set $A$ such that $w \in A^*$. The concatenation of two strings $a$ and $b$ is written as its juxtaposition $ab$. For a string $w_1 w_2 \ldots w_i \ldots w_n$ of length $n$ over $A$, the *position* of a symbol $w_i \in A$ is $i \in [n]$. The canonical extensions of a function $f\colon X \to Y$ to the power set of $X$ and to $X^*$ are denoted by $f$ as well. Thus, $f(\{x_1, \ldots, x_n\}) = \{f(x_1), \ldots, f(x_n)\}$ and $f(x_1 \cdots x_n) = f(x_1) \cdots f(x_n)$ for all $x_1, \ldots, x_n \in X$. For a set $Y$ a (locally finite) multiset over $Y$ is a function $\mu\colon Y \to \mathbb{N}$. For brevity, we give a specific multiset by a string notation $y_1^{\mu(y_1)} \cdots y_n^{\mu(y_n)}$ for $y_1 \ldots y_n \in Y$. The size of $\mu$ is $|\mu| = \sum_{y \in Y} \mu(y)$. (Formally, $|\mu| = \infty$ if $\mu(y) \geq 1$ for infinitely many $y$, but this case will not be relevant for this paper, i.e., all multisets appearing here will be finite.) We denote the set of all multisets over $Y$ by $\mathbb{N}^Y$. For a function $f\colon X \to Y$, we let $f_M\colon 2^X \to \mathbb{N}^Y$ be the mapping such that, for every $X' \subseteq X$, $f_M(X')$ is the multiset of images of elements of $X'$ under $f$. Thus, formally, $f_M(X')(y) = |\{x \in X' \mid f(x) = y\}|$ for every $y \in Y$.

This article studies languages of vertex-labeled, directed multigraphs without loops and with ordered unlabeled edges (called *graphs*, see Def. 2.1) which are acyclic (called *DAGs*, see Def. 2.4). Edges will be labeled only temporarily by a grammar (c.f. Def. 2.5), its equivalent automaton or classical finite state automaton (see Section 7).

**Definition 2.1** (Graph). *A* graph *over $\Gamma$ is a tuple $G = (V, E, \ell, in, out)$ with $\Gamma$, $V$ and $E$ being disjoint finite sets, the sets of* vertex labels, vertices *and* edges, *respectively. The vertices are labeled by $\ell\colon V \to \Gamma$. For an* edge *$e \in E$ between the vertices $(v, w) \in V \times V$,* directed *from $v$ to $w$, with $v \neq w$, the* source *$v$ is referenced by $src(e)$ and the* target *$w$ by $tar(e)$. By $in, out\colon V \to E^*$ we assign to each vertex $v \in V$ its* incoming *and* outgoing *edges such that $src(e) = v \Leftrightarrow e \in [out(v)]$ and $tar(e) = v \Leftrightarrow e \in [in(v)]$. These edges are ordered as specified by the strings $in(v)$ and $out(v)$. The* empty graph *$\varnothing$ is the graph whose set of vertices is empty. A vertex is called a* root *or a* leaf *if $in(v)$ or $out(v)$ are empty, respectively. The disjoint union of graphs, meaning disjoint sets of vertices and edges, is denoted by the operator* &.

**Definition 2.2** (Path). *A* path *in a graph $G = (V, E, \ell, in, out)$ is a nonempty sequence of edges $e_1, \ldots, e_n$, $e_i \in E$ for $i \in [n]$, yielding a unique alternating sequence $v_0 e_1 v_1 \cdots e_n v_n$ with vertices $v_0, \ldots, v_n \in V$ such that $\{src(e_i), tar(e_i)\} = \{v_{i-1}, v_i\}$ for all $i \in [n]$. Such a path is a* cycle *if $v_0 = v_n$. A* path between *$s$ and*

*$t$ is a path with $s \in \{v_0, e_1\}$ and $t \in \{v_n, e_n\}$. A path is* directed *if for $i \in [n]$ either $\forall i : tar(e_i) = v_i$ or $\forall i : tar(e_i) = v_{i-1}$ and we call it a* path from $s$ to $t$ *if it is a directed path between $s$ and $t$ with $\forall i : tar(e_i) = v_i$. The* length *of a path is the number of its edges, written as $|e_1, \ldots, e_n| = n$. The graph $G$ is said to be* connected *if there is a path between each pair of vertices. In a path specification, we may denote the vertices and edges $a \in V \cup E$ by their respective label $\ell(a)$.*

**Definition 2.3** (Chord Path)**.** *A* chord path *of a cycle shares its end vertices with its corresponding cycle, but none of its edges [33, 15]. Given a graph $G = (V, E, \ell, in, out)$, let the path $c = e_1, \ldots, e_n$ with $e_i \in E$ for $i \in [n]$, be a cycle yielding $v_n e_1 \cdots e_n v_n$. A* chord path *of the cycle $c$ is a path $e'_1, \ldots, e'_m$ with $e'_j \in E$ for $j \in [m]$ yielding $v'_0 e'_1 v'_1 \cdots e'_n v'_m$ with vertices $v'_0, \ldots, v'_m \in V$ such that $v'_0 \neq v'_m$ and $\{v'_0, v'_m\} = \{v_0, \ldots, v_n\} \cap \{v'_0, \ldots, v'_m\}$ but $\{e_i \mid i \in [n]\}$ and $\{e'_j \mid j \in [m]\}$ being disjoint sets.*

**Definition 2.4** (DAG, complete DAG, prefix-DAG)**.** *A* directed acyclic graph (over $\Gamma$)*, abbreviated as* DAG*, is a graph over $\Gamma$ that does not contain any directed cycle. The set of all connected and nonempty DAGs over $\Sigma$ is denoted by $\mathscr{D}_\Sigma$. A connected DAG $G = (V, E, \ell, in, out)$ is called a* string DAG *iff $|in(v)| \leq 1$ and $|out(v)| \leq 1$ for all vertices $v \in V$. Throughout this paper, $\Sigma$ and $N$ being disjoint sets, $\Sigma$ will denote an* alphabet of terminals*, namely, the* vertex labels*, whereas $N$ is our* alphabet of nonterminals *used for labeling vertices and edges temporarily. We call a DAG over $\Sigma$ a* complete DAG*. A DAG over $\Sigma \cup N$ is called a* prefix-DAG*, a* proper prefix-DAG *if at least one vertex is labeled by a nonterminal $n \in N$.*

**Definition 2.5** (Regular DAG grammar, $L(\mathscr{G})$, $L(\mathscr{G})^{\&}$ [13])**.** *A* regular DAG grammar[2] *is a triple $\mathscr{G} = (N, \Sigma, R)$. Each* rule *$r \in R$ is of the form $\alpha \twoheadrightarrow \textcircled{\sigma} \twoheadrightarrow \beta$ where $\sigma \in \Sigma$ while the* head *$\alpha$ and the* tail *$\beta$ are elements of $N^*$. For the prefix DAGs $G$ and $G'$, there exists a* derivation step *$G \Rightarrow_r G'$ using a rule $r$ if $G$ contains pairwise distinct vertices $v_1, \ldots, v_k$ such that $\ell(v_1 \cdots v_k) = \alpha$. In that case, $G'$ is obtained from $G$ by*

- *adding the vertex $v$ with its label $\ell(v) = \sigma$*
- *by letting the edges, formerly pointing to $v_1, \ldots, v_k$, now point to $v$, thus $tar(in(v_i)) = v$,*
- *deleting the temporary vertices $v_1, \ldots, v_k$ and, in turn,*
- *adding the temporary vertices $w_1, \ldots, w_j$ labeled by their nonterminals $\ell(w_1 \cdots w_j) = \beta$*
- *by connecting them to the graph with the edges $(v, w_1), \ldots (v, w_j)$.*

*A* derivation *is a sequence of prefix DAGs[3] $G_0 \Rightarrow_{r_1} G_1 \Rightarrow_{r_2} \cdots \Rightarrow_{r_n} G_n$, also denoted by $G_0 \Rightarrow_{r_1 \cdots r_n} G_n$. The set of all these $G_n$ that are complete is denoted by $L(\mathscr{G})^{\&}$. The* DAG language generated by $\mathscr{G}$ *is $L(\mathscr{G}) = \{G \in \mathscr{D}_\Sigma \mid \varnothing \Rightarrow^*_R G\}$, the set of connected and complete DAGs which the grammar can derive, where $\Rightarrow^*_R$ denotes the transitive and reflexive closure of $\Rightarrow_R = \bigcup_{r \in R} \Rightarrow_r$. As usual, a rule is said to be* useless *if none of the derivations for DAGs in $L(\mathscr{G})$ comprises this rule and* useful *if it does. The DAG grammar $\mathscr{G}$ is* deterministic *if, for every pair $\alpha \twoheadrightarrow \textcircled{\sigma} \twoheadrightarrow$ in $N^* \times \Sigma$, there exists at most one $\beta \in N^*$ such that $(\alpha \twoheadrightarrow \textcircled{\sigma} \twoheadrightarrow \beta) \in R$. The DAG language generated by a deterministic DAG grammar, as well as a DAG automaton recognizing it, is called* top-down deterministic*. By reversing the orientation of the edges, we obtain its* dual language*. A language $L(\mathscr{G})$ of a deterministic DAG grammar is called* bottom-up deterministic *if its dual language is generated by $\mathscr{G}$.*

*The class RDL of* regular DAG languages *consists of all DAG languages generated by a regular DAG grammar (equivalently recognized by a regular DAG automaton). The class $RDL^{det}$ of* regular

[2]Since regular DAG grammars are equivalent to regular DAG automata, an illustrative example of how the DAGs are handeled can be found in [12].

[3]We extend the notation $\Rightarrow_{r_1 \cdots r_n}$ to $\Rightarrow_E$, where $E$ is an (extended) regular expression over rules: if $L(E)$ denotes the language of sequences of rules denoted by $E$, then $\Rightarrow_E = \bigcup_{r_1 \cdots r_n \in L(E)} \Rightarrow_{r_1 \cdots r_n}$.

deterministic DAG languages *consists of all DAG languages and dual languages generated by regular deterministic DAG grammars (equivalently recognized by a regular deterministic DAG automaton).*

In a derivation of a DAG $G$, at the time a new edge $e$ is generated, its newly created target vertex $v$ is labeled by a nonterminal, say $q$. At that time $v$ "dangles" at the end of $e$ without further incoming or outgoing edges. Later rule applications will take $v$, merge it with other vertices and label the resulting vertex with its final symbol taken from $\Sigma$ according to the rule used. The edge $e$, however, remains untouched. We may represent a derivation of $G$ up to reordering of derivation steps by the DAG $G$ itself together with a labeling of edges by nonterminals. Then, $e$ would be labeled with $q$. We call this the derivation DAG of $G$.

**Definition 2.6** (Derivation DAG, $\lfloor D \rfloor$)**.** *Let $G_0 \Rightarrow_{r_1 \cdots r_n} G_n$ with $r_1, \dots r_n \in R$ be a derivation of a DAG $G_n = G = (V,E,\ell,in,out)$ generated by a DAG grammar $\mathscr{G} = (N,\Sigma,R)$. Then, the corresponding* derivation DAG *of $G$ is the tuple $D = (V,E,\ell,in,out)$, where $\ell \colon E \cup V \to \Sigma \cup N$ is extended to edges by: for every edge $e \in E$, $\ell(e)$ is the unique nonterminal $q \in N$ such that, for some $i \in [n]$, $e$ is an edge of $G_i$ with $\ell(tar_{G_i}(e)) = q$. Let $\lfloor D \rfloor$ denote the DAG $G_n$, obtained from $D$ by restricting $\ell$ to $V$, denoted by $\ell|_V$.*

A derivation DAG is not necessarily connected, thus $\lfloor D \rfloor \in L(\mathscr{G})^{\&}$ if $\varnothing \Rightarrow_R^* D$ but $\lfloor D \rfloor \in L(\mathscr{G})$ only if $D$, or equivalently $\lfloor D \rfloor$, is connected. It should be noted that the set of all derivation DAGs of (DAGs in) $L(\mathscr{G})$ is easily characterized: For every such derivation DAG $D = (V,E,\ell,in,out)$ the DAG $G = (V,E,\ell|_V,in,out)$ is an element of $\mathscr{D}_\Sigma)$ and for every vertex $v \in V$ there is a rule $\alpha \twoheadrightarrow \textcircled{$\sigma$} \twoheadrightarrow \beta$ such that $\alpha = \ell(in(v))$ and $\beta = \ell(out(v))$. Thus, the derivation DAGs of $\mathscr{G}$ coincide with the runs of the DAG automata in [5], and $L(\mathscr{G})$ is the set of all DAGs $\lfloor D \rfloor$ such that $D$ is a derivation DAG of a DAG generated by $\mathscr{G}$. Moreover, a regular DAG grammar $\mathscr{G}$ without useless rules is deterministic if and only if every DAG in $L(\mathscr{G})$ has exactly one derivation DAG.

**Definition 2.7** (Rule Path and Cycle)**.** Marking *a symbol $q$ (at a position $i \in [n]$ of a string of length $n$) by a* mark $\bar{}$ *means replacing it with $\bar{q}$. We mark rules with the* entry mark $\check{}$ *and the* exit mark $\hat{}$*; if it is not specified which of those two marks is used, we use $\bar{}$. A* marked rule *$\bar{r} = (\bar{\alpha} \twoheadrightarrow \textcircled{$\sigma$} \twoheadrightarrow \bar{\beta})$ is obtained from a rule $r = (\alpha \twoheadrightarrow \textcircled{$\sigma$} \twoheadrightarrow \beta)$ by marking two nonterminals at two distinct positions $i, j \in [|\alpha\beta|]$ in $\alpha\beta$, one with the entry, one with the exit mark; in a* weakly *marked rule only at one position with either entry or exit mark. Such a* marked nonterminal *is referenced by its tuple $(\bar{q}, \bar{r})$ where $\bar{r} = (\bar{\alpha} \twoheadrightarrow \textcircled{$\sigma$} \twoheadrightarrow \bar{\beta})$ is the (weakly) marked rule in which $q$ is replaced with $\bar{q}$. A rule pair $(\bar{r}_i, \bar{r}_j)$ for the (weakly) marked rules $\bar{r}_i$ and $\bar{r}_j$* agrees on *the marked nonterminals $(\hat{q}, \bar{r}_i)$ and $(\check{q}, \bar{r}_j)$ if $q$ is marked once in a head and once in a tail in order to obtain $(\hat{q}, \bar{r}_i)$ and $(\check{q}, \bar{r}_j)$. Two weakly marked rules always agree – regardless of their marked nonterminals. A* rule sequence *is a nonempty sequence of (weakly) marked rules $\bar{r}_1, \dots, \bar{r}_n$ for which all rule pairs $(r_i, r_{i+1 \bmod n})$ with $i \in [n]$ agree and every marked nonterminal in the rule sequence is exactly once agreed on. A rule sequence of marked rules $\bar{r}_1, \dots, \bar{r}_n$ is a* rule cycle*, with $\bar{r}_1$ and $\bar{r}_n$ being weakly marked it is a* rule path*. A rule path* between *$s$ and $t$ is a rule path with marked nonterminals $\bar{q}_1, \dots, \bar{q}_{n-1}$ which yields a path $\sigma_1, q_1, \dots, q_{n-1}, \sigma_n$ in a derivation DAG between $s$ and $t$ for $s \in \{q_1, \sigma_1\}$ and $t \in \{q_n, \sigma_n\}$.*

Figures 5 and 6a show examples of rule sequences. Observe that these definitions permit two types of rule sequences. A rule sequence yields a directed path in a graph if and only if both mark types, exit and entry, do not occur both in heads and tails. We call this a *directed* rule sequence. If heads (and consequently tails) comprise both types of marks, the resulting path in the graph will be undirected. This is called an *undirected* rule sequence.

**Observation 2.8** (Yielding Cycle)**.** *Obviously, a directed rule cycle cannot yield a directed cycle in a DAG, since DAGs are acyclic. Therefore, only undirected rule cycles can yield a cycle in a DAG. Directed rule cycles yield paths only, no cycles, in a DAG, undirected ones yield both paths and undirected cycles.*

**Theorem 2.9** (Infinite Language [5]). *The DAG language generated by a DAG grammar $\mathscr{G} = (N, \Sigma, R)$ without useless rules is infinite iff $R$ contains a rule cycle.*

**Definition 2.10** (Swap). *Let $G = (V, E, \ell, in, out)$ be a DAG. Two edges $e_0, e_1 \in E$ are* independent *if there is no directed path between $e_0$ and $e_1$. In this case, the* edge swap of $e_0$ and $e_1$ *is defined and yields the DAG $G[e_0 \bowtie e_1] = (V, E, \ell, swap \circ in, out)$ given by the bijection $swap\colon E \to E$ defined as $swap(e_i) = e_{1-i}$ for $i \in \{0,1\}$ and $swap(e) = e$ for $e \notin \{e_0, e_1\}$. For $k \in \mathbb{N}$, let $G_0, G_1, \ldots, G_k$ be $k+1$ disjoint isomorphic copies of $G$, and for $i \in \{0, 1, \ldots, k\}$, let $e_i$ and $e'_i$ be the copies of $e$ and $e'$ in $G_i$, respectively. Then the graph $G(e \bowtie e')^k$ for $k \in \mathbb{N}$ is defined as*

$$G(e \bowtie e')^0 = G_0 \quad \text{and} \quad G(e \bowtie e')^k = (G(e \bowtie e')^{k-1} \,\&\, G_k)[e'_{k-1} \bowtie e_k].$$

Swapping two edges means that the tips of the arrows, the edge targets, are exchanged with one another. This swapping operation is, of course only allowed, if the result of the swap is still a DAG. Swapping edges yielding a directed cycle is not defined. The operation is central for regular DAG languages since, after swapping two edges in a DAG that have the same label in one of its derivation DAGs, the swapped result is still part of the language.

**Lemma 2.11** (Swap Preserves Generation [5]). *Let $\mathscr{G} = (N, \Sigma, R)$ be a regular DAG grammar and $D = (V, E, \ell, in, out)$ a derivation DAG with $\lfloor D \rfloor \in L(\mathscr{G})^{\&}$. Then, if $\ell(e_0) = \ell(e_1)$ for $e_0, e_1 \in E$, the edge swap of $e_0$ and $e_1$ in $D$, in case it is defined, yields a DAG generated by $\mathscr{G}$, thus $\lfloor D[e_0 \bowtie e_1] \rfloor \in L(\mathscr{G})^{\&}$.*

Deterministic DAG grammars as defined above are equivalent to the top-down deterministic DAG automata in [5] and every derivation DAG of a grammar corresponds one-to-one to a run of the corresponding DAG automaton. Therefore, all results for top-down deterministic DAG automata carry over to deterministic DAG grammars. Refer to [5] for the notation concerning DAG automata. In particular, this holds for the following theorem, in which a regular DAG grammar is called minimal if it does not contain useless rules and there is no regular DAG grammar with fewer nonterminals generating the same language.

**Theorem 2.12** (Minimal Grammar [5]). *For every deterministic DAG grammar $\mathscr{G}$, a minimal deterministic DAG grammar $\mathscr{G}'$ with $L(\mathscr{G}') = L(\mathscr{G})$ can be computed in polynomial time. This DAG grammar is unique: every minimal deterministic DAG grammar that accepts $L(\mathscr{G})$ is identical to $\mathscr{G}'$ up to a bijective renaming of its nonterminals.*

## 3 Meta-State

Classical finite state automata recognize string languages by a finite memory, its set of states. In every step the automaton memorizes exactly one state. On the contrary, while generating a DAG, every derivation step of a DAG grammar has to recall several nonterminals and not just one. If we summarize those nonterminals to one meta-state per derivation step, can we then accept DAGs with a finite state automaton? If so, which kind of DAG languages can the finite state automaton recognize?

**Definition 3.1** (Meta-state). *A multiset over nonterminals, i.e. an element of $\mathbb{N}^N$, is called a* meta-state. *The symbol $\mathscr{Q}$ is used for sets of meta-states, thus $\mathscr{Q} \in 2^{\mathbb{N}^N}$. For a DAG $G = (V, E, \ell, in, out)$ over $\Sigma \cup N$ we let $\underline{G}$ denote the meta-state $\ell_M(\{v \in V \mid \ell(v) \in N\})$, the multiset of all labels which are nonterminals.*[4]

Observe that it only depends on the meta-state of a graph, not on the graph as a whole, how the derivation can proceed. For this we define the notion of a graph being useful.

[4]Note that, according to the *Notation* section, the function $\ell_M$ returns a *multiset* of labels.

**Definition 3.2** (Useful)**.** *We say that a prefix DAG $G$ is* useful *with respect to a given grammar $\mathscr{G}$, if this grammar $\mathscr{G}$ derives $\varnothing \Rightarrow_R^* G \Rightarrow_R^* G'$ with the DAG $G'$ being complete and* language-useful *if $G'$ is complete and connected, thus equivalently, if $G$ occurs in a derivation of a DAG $G' \in L(\mathscr{G})$.*

Note that a prefix DAG with respect to a grammar is useful if and only if all its connected components are language-useful. Which properties depend on the meta-state only?

**Lemma 3.3** (Meta-state dependent)**.** *Let $G, G'$ be prefix DAGs with $\underline{G} = \underline{G'}$ derived by a DAG grammar $\mathscr{G} = (N, \Sigma, R)$. Then, $\mathscr{G}$ can apply the rule $r \in R$ as the next derivation step $G \Rightarrow_r H$ iff $G' \Rightarrow_r H'$. Similarly, $G$ is useful if and only if $G'$ is useful. However, if $G$ is language-useful this only implies that $G'$ is useful.*

Interestingly, a minimal grammar can finalize every derivation to a complete DAG.

**Lemma 3.4** (Useful Prefix DAG)**.** *Let $G$ be a prefix DAG and $\mathscr{G} = (N, \Sigma, R)$ be a DAG grammar without useless rules. If $\varnothing \Rightarrow_R^* G$, then $G$ is* useful with respect to *$\mathscr{G}$, i.e. there exists a derivation $\varnothing \Rightarrow_R^* G \Rightarrow_R^* G'$ for a complete DAG $G'$. If $G'$ is connected it is* language-useful.

After having defined the notion of a meta-state, let us use them for derivations. The first naive idea is to use all meta-states which occur in all the derivations for complete graphs. We call that set $\mathscr{Q}_{\mathbf{0}}$ because a derivation of a DAG in the language starts and ends with zero states. Apart from $\mathscr{Q}_{\mathbf{0}}$ also another set of meta-states is of interest. While $\mathscr{Q}_{\mathbf{0}}(\mathscr{G})$ incorporates all meta-states occurring during all derivations of DAGs in $L(\mathscr{G})$, not all of these meta-states may be needed to generate the language $L(\mathscr{G})$. This observation gives rise to a smaller set of meta-states $\mathscr{Q}_{\min}$.

**Definition 3.5** ($\mathscr{Q}_{\mathbf{0}}$ and $\mathscr{Q}_{\min}$)**.** *Let $\mathscr{G}$ be DAG grammar. The set of all meta-states that occur in derivations of DAGs in $L(\mathscr{G})$ is denoted by $\mathscr{Q}_{\mathbf{0}}(\mathscr{G})$ with*

$$\mathscr{Q}_{\mathbf{0}}(\mathscr{G}) = \{\underline{G} \mid G \textit{ is a DAG which is language-useful with respect to } \mathscr{G}.\}.$$

*A minimal set of meta-states is denoted by $\mathscr{Q}_{min}$. And, $\mathscr{Q}_{min}(\mathscr{G})$ denotes any set of meta-states such that*

1. *every DAG $G_n \in L(\mathscr{G})$ has a derivation $\varnothing \Rightarrow_{r_1} G_1 \Rightarrow_{r_2} \dots \Rightarrow_{r_n} G_n$ such that $\underline{G_1}, \dots, \underline{G_n} \in \mathscr{Q}_{min}(\mathscr{G})$, and*
2. *there is no set of meta-states of smaller cardinality with this property.*

Thus, $\mathscr{Q}_{\min}(\mathscr{G})$ is a minimal set of meta-states sufficient to generate a DAG language, while $\mathscr{Q}_{\mathbf{0}}(\mathscr{G})$ incorporates also meta-states that could be dispensed. The set $\mathscr{Q}_{\min}(\mathscr{G})$ is not necessarily unique since often several derivations exist for one DAG, and furthermore a permutation of derivation steps may result in different meta-states. In general, we are interested in $|\mathscr{Q}_{\min}(\mathscr{G})|$, and in particular its finiteness, rather than in the set itself.

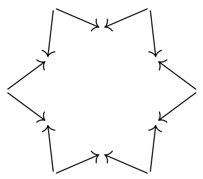
The subsequent example illustrates the existence of DAG grammars for which $\mathscr{Q}_{\min}(\mathscr{G})$ is finite while $\mathscr{Q}_{\mathbf{0}}(\mathscr{G})$ is not.

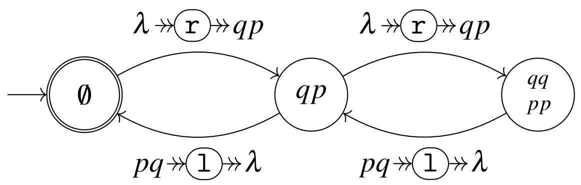
**Example 3.6** (DAG language of stars)**.** *Consider a DAG grammar $\mathscr{G}_{star}$ with the rules $r = \lambda \twoheadrightarrow \textcircled{\texttt{r}} \twoheadrightarrow qp$ and $l = pq \twoheadrightarrow \textcircled{\texttt{l}} \twoheadrightarrow \lambda$. Let $G \in L(\mathscr{G}_{star})$, cf. Fig. 3a. First, $\varnothing \Rightarrow_{r^n} G_{root} \Rightarrow_{l^n} G$ for $n \geq 1$ generates a graph $G_{root}$ consisting of $n$ roots labeled* `r`*. Subsequently, $l$ fuses pairs of nonterminal vertices into a single leaf labeled* `l`*. Collecting the meta-states that occur in these derivations or in an arbitrary derivation $\varnothing \Rightarrow_R^* G$ both result in $\mathscr{Q}_{\mathbf{0}}(\mathscr{G}_{star}) = \{p^n q^n \mid n \in \mathbb{N}\}$, since every rule either consumes or produces both a pair of nonterminals $q$ and $p$. However, by first generating a single root, then alternating between $r$ and $l$, and finally applying $l$ once more, $\mathscr{G}_{star}$ offers the derivations $\varnothing \Rightarrow_{r(rl)^*l} G$ whose largest meta-state is $p^2q^2$. Hence, $\mathscr{Q}_{min}(\mathscr{G}_{star}) = \{pq, p^2q^2\}$.*

The previous example gives rise to the following observation.

**Observation 3.7.** *DAG Grammars $\mathscr{G}$ with finite $\mathscr{Q}_{min}(\mathscr{G})$ but infinite $\mathscr{Q}_{\mathbf{0}}(\mathscr{G})$ exist.*

This brings us to a further investigation of the finiteness of $\mathscr{Q}_{\mathbf{0}}$ and $\mathscr{Q}_{\min}$.


(a) DAG in $L(\mathcal{G}_{star})$



$\lambda \twoheadrightarrow$ r $\twoheadrightarrow qp$
$\lambda \twoheadrightarrow$ r $\twoheadrightarrow qp$
$\emptyset$
$qp$
$qq$ $pp$
$pq \twoheadrightarrow$ l $\twoheadrightarrow \lambda$
$pq \twoheadrightarrow$ l $\twoheadrightarrow \lambda$

(b) FSA for $\mathcal{G}_{star}$

Figure 3: The grammar $\mathcal{G}_{star}$ gives rise to an FSA (b) that accepts DAGs like (a) (*labels are omitted*).

## 4 Finite Number of Meta-States

The previous section showed that languages generated with finite $\mathcal{Q}_{\min}$ indeed exist. This section investigates which types of DAG languages can be generated with a finite number of meta-states induced by the rules of a minimal deterministic grammar. First, the newly identified language class deserves a name. We call it *finite induced meta-state DAG language*. The term *induced* is chosen since the grammar $\mathcal{G}$ induces this set $\mathcal{Q}$ by a suitable (c.f. Lemma 4.4) or all (c.f. Lemma 4.3) derivation DAGs.

**Definition 4.1** (Finite induced meta-state DAG language (`FID`))**.** *A language recognized by a minimal deterministic grammar $\mathcal{G}$ with finite $\mathcal{Q}_{min}(\mathcal{G})$ is called a* finite induced meta-state DAG language. *The language class comprising finite induced meta-state DAG languages is denoted by* `FID`.

In the following, we look into different categories for finite sets of meta-states. For each category we check three types, first languages, second paths and finally rule cycles.

**Lemma 4.2** (Finite $L(\mathcal{G})$ – finite $\mathcal{Q}_{\mathbf{0}}$ and $\mathcal{Q}_{\min}$)**.** *Let $\mathcal{G} = (N,\Sigma,R)$ be a DAG grammar.*

1. *If $L(\mathcal{G})$ is finite, the sets $\mathcal{Q}_{\mathbf{0}}(\mathcal{G})$ and $\mathcal{Q}_{min}(\mathcal{G})$ are finite as well.*
2. *For a rule path $\Pi$ of finite length within R, whose marked rules comprise only marked nonterminals except for the weak marked rules, finitely many meta-states suffice for all derivation sequences which generate the corresponding path $\pi$ of $\Pi$ in a DAG.*

*Proof.* Since $N$ is finite and $L(\mathcal{G})$ is finite, only a finite number of derivations for all DAGs in $L$ exists. Obviously, this combination yields a finite number of meta-states for $\mathcal{Q}_{\mathbf{0}}(\mathcal{G})$ and consequently also for $\mathcal{Q}_{\min}(\mathcal{G})$. In the second statement, the finite rule path $\Pi$ gives rise to a finite derivation $G \Rightarrow_{\Pi} G'$ for all possible prefix DAGs $G$ of $G'$, since, due to all nonterminals marked except start and end, $\Pi$ does not need to be interleaved with rules. A finite derivation yields finitely many meta-states, both for $\mathcal{Q}_{\min}$ as well as for $\mathcal{Q}_{\mathbf{0}}$. $\square$

Infinite languages $L(\mathcal{G})$ do not necessarily induce an infinite $\mathcal{Q}_{\min}$ and not even an infinite $\mathcal{Q}_{\mathbf{0}}$. First, we look at those cases, where they indeed induce finite sets of meta-states only.

**Lemma 4.3** (Strings – finite $\mathcal{Q}_{\mathbf{0}}$ and $\mathcal{Q}_{\min}$)**.** *Let $\mathcal{G} = (N,\Sigma,R)$ be a DAG grammar.*

1. *If $L(\mathcal{G})$ is a string DAG language the sets $\mathcal{Q}_{\mathbf{0}}(\mathcal{G})$ and $\mathcal{Q}_{min}(\mathcal{G})$ are finite*
2. *For a, possibly arbitrary long, directed rule path, $\Pi$ in R, whose marked rules comprise only marked nonterminals except for the weak marked rules, finitely many meta-states suffice for all derivation sequences which generate the corresponding path $\pi$ by $\Pi$ in a DAG.*

3. *For a directed rule cycle $c$ whose marked rules comprise only marked nonterminals, finitely many meta-states suffice for all derivation sequences which generate the corresponding path $\pi$ by $c$ in a DAG.*

*Proof.* Since string languages consume and produce exactly one nonterminal in every derivation step except for the first and last steps producing and consuming one nonterminal only, respectively, $N$ equals $\mathscr{Q}_{\mathbf{0}}(\mathscr{G})$.[5] Obviously, this means that $\mathscr{Q}_{\mathbf{0}}(\mathscr{G})$ is finite, since $N$ is finite.

This carries over to subgraphs which are string DAGs. Strings as subgraphs are derived in the statements (2) and (3). Let $\bar{r}$ denote a marked rule with only marked nonterminals of the form $\check{q} \twoheadrightarrow \textcircled{\sigma} \twoheadrightarrow \hat{p}$. Such a rule $\bar{r}$ does not alter the size of the meta-states, thus $|\underline{G}| = |\underline{G'}|$, in its derivation step $G \Rightarrow_r G'$. For a directed rule cycle $c$ with only marked nonterminals holds the same just as for an infinite rule path $\Pi$ in $R$, since they consist of a rule sequence with rules of type $\bar{r}$. Such a rule path $\Pi$ comprises only finitely many marked rules since $R$ is finite. Thus, an arbitrary long rule path corresponds to a rule cycle. Consequently, $\mathscr{Q}_{\mathbf{0}}$ is finite for both (2) and (3).

In all three cases $\mathscr{Q}_{\mathbf{0}}$ is finite, yielding a finite $\mathscr{Q}_{\text{min}}$. □

In summary, for strings of any kind, string languages or strings as subgraphs, also of unbounded length, both $\mathscr{Q}_{\text{min}}$ and $\mathscr{Q}_{\mathbf{0}}$ stay finite. This was to be expected since regular string languages are accepted by finite state automata with a finite set of states. What happens beyond string DAGs?

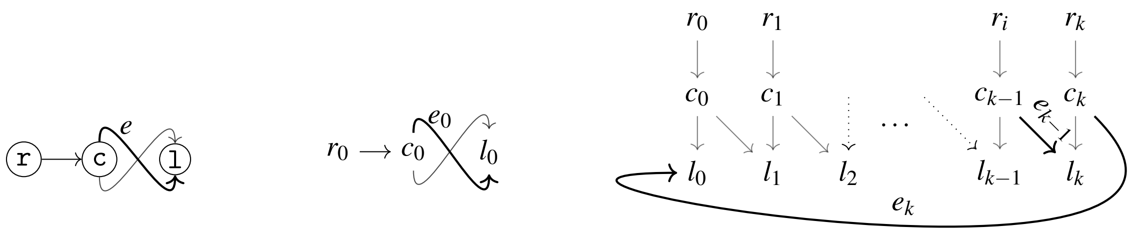
**Lemma 4.4** (Finite $\mathscr{Q}_{\text{min}}$ – infinite $\mathscr{Q}_{\mathbf{0}}$). *Let $\mathscr{G} = (N, \Sigma, R)$ be a minimal deterministic DAG grammar*

1. *In the case where $\mathscr{Q}_{min}(\mathscr{G})$ is finite, it is possible that $\mathscr{Q}_{\mathbf{0}}(\mathscr{G})$ is infinite.*
2. *For a possibly arbitrary long, undirected rule path, $\Pi$ in $R$, whose marked rules comprise only marked nonterminals except for the weak marked rules, finitely many meta-states suffice for deriving all complete DAGs incorporating the corresponding path $\pi$ generated by $\Pi$, however, some derivations for each such complete DAG ask for an infinite set of meta-states.*
3. *For an undirected rule cycle $c$ in $R$, whose marked rules comprise only marked nonterminals, finitely many meta-states suffice for deriving all complete DAGs incorporating the corresponding path $\pi$ of unbounded length generated by $c$. However, some derivations for each such complete DAG ask for an infinite set of meta-states.*

*Proof.* The first statement is equivalent to Observation 3.7. Both Statements (2) and (3) may yield a path $\pi$ of unbounded length. Such a path $\pi$ gives rise to a derivation which does not increase the cardinality of the meta-state, such that $\mathscr{Q}_{\text{min}}$ with respect to $\Pi$ and $c$ is finite. This is possible by using rules that add nonterminals to the current meta-state only when they are needed, just like for the DAG language of stars in Example 3.6. But for undirected rule paths and cycles, we can rearrange the rules in the derivations in order to first generate all roots (again, c.f. Example 3.6). Since both $\Pi$ and $c$ can make the path $\pi$ unbounded, the size of the meta-states will grow without bound, yielding an infinite $\mathscr{Q}_{\mathbf{0}}$. □

In the next section, we will turn to languages which cannot be generated with a finite set of meta-states. Prior to this, we look at the unexpected fact that although a label does not occur in any rule cycle, its number of occurrences could be unbounded.

[5]Since we use strings as a notation for multisets, this special set of multisets equals a normal set.



(a) $G$ (b) $G_0 = G$ (c) $G_k[e_{k-1} \bowtie e_k] = G_{k-1} \,\&\, G_k$

Figure 4: The decorated one-pointed star $G$ is shown in (a). This equals the star $G_0$ in (b), where in addition to the vertex label, the label's index allows us to reference each vertex uniquely by its number of copies. Note thus, that in (b), as well as in (c), the index is not part of the vertex label. To draw the graph itself in those two pictures, the indices would be stripped off. Note, (c) illustrates the swapping of $k+1$ disjoint isomorphic copies to a $k+1$-pointed star $G(e \bowtie e)^k$.

**Lemma 4.5** (Unbounded without Rule Cycle)**.** *There exists a DAG grammar $\mathscr{G} = (N, \Sigma, R)$ and a vertex or edge label* u $\in \Sigma \cup N$ *such that the number of occurrences of* u *in a (derivation) DAG generated by $\mathscr{G}$ is unbounded, although* u *does not occur in any rule cycle.*

The following lemma summarizes when the number of label occurrences is not bounded.

**Lemma 4.6** (Unbounded Label Occurrence)**.** *Let $\mathscr{G} = (N, \Sigma, R)$ be a minimal deterministic DAG grammar and* u $\in \Sigma \cup N$ *a label of a vertex or an edge. The number of occurrences of* u *in graphs $G \in L(\mathscr{G})$, or, for edge labels, in their corresponding derivation DAGs D, is unbounded, iff one of the two following conditions is fulfilled:*

*a) Label* u *occurs in some rule cycle of $\mathscr{G}$.*

*b) There exist both a rule cycle c in which an unmarked $q \in N$ occurs as well as a rule path $\Pi$ between this nonterminal q and the label* u.

## 5 Infinite Number of Meta-States

Unfortunately, from the algorithmic viewpoint at least, there exist infinite DAG languages where a finite number of meta-states is not sufficient for generating them in a top-down deterministic manner. We call this language class the infinite meta-state DAG languages, abbreviated as ID.

**Definition 5.1** (Infinite meta-state DAG language (ID))**.** *A language $L(\mathscr{G})$ of a minimal deterministic DAG grammar $\mathscr{G}$ is called an* infinite meta-state DAG language *if $\mathscr{Q}_{min}(\mathscr{G})$ is infinite. The language class comprised of all infinite meta-state DAG languages is denoted* ID.

**Lemma 5.2** (Infinite $\mathscr{Q}_{\min}$ – infinite $\mathscr{Q}_{\mathbf{0}}$)**.** *Infinite meta-state DAG languages exist.*

*Proof.* Let us consider the following DAG grammar for binary trees $\mathscr{G}_{tree} = (\{q\}, \{\mathtt{r}, \mathtt{m}, \mathtt{L}\}, R)$. As a tree, every $G \in L(\mathscr{G}_{tree})$ has only one root. Therefore, every derivation for the tree grammar $\mathscr{G}_{tree}$ applies $r_r = \lambda \twoheadrightarrow\!\text{(r)}\!\twoheadrightarrow qq$ only once: $\varnothing \Rightarrow_{r_r} G_{\text{root}} \Rightarrow_{R \setminus \{r_r\}} G$. The leaf rule $r_l = q \twoheadrightarrow\!\text{(l)}\!\twoheadrightarrow \lambda$ cannot be part of a rule cycle since this requires a minimum of two nonterminals. The rule $r_m = q \twoheadrightarrow\!\text{(m)}\!\twoheadrightarrow qq$ is the only rule that occurs in a rule cycle.

Every derivation step using $r_m$ increases the cardinality of the meta-state by one due to consuming one nonterminal and producing two nonterminals. If $r_m$ is the only rule in a rule cycle and $r_m$ increments the meta-state in every derivation step $\Rightarrow_{r_m}$, do we end up with an infinite set $\mathcal{Q}_{\min}$? Luckily, Lemma 4.5 tells us that we can repeatedly apply derivation steps also with rules not being part of a rule cycle. Lemma 4.6 confirms that this is the case for the rule $r_l$, since we have an unmarked $q$ in all marked rule combinations of $r_m$ for a rule cycle and we have a rule path $q$ to `l` only with the rule $r_l$. The number of leaves, labeled by `l`, is not bounded. As a sole rule in $R$ the leaf rule decreases the cardinality of the meta-state. It is folklore that the amount of leaf vertices is one more than non-leaf vertices. This sounds thus promising to end up for a finite set $\mathcal{Q}_{\min}$. The derivations $\varnothing \Rightarrow_{r_r} G_{\text{root}} \Rightarrow_{r_l} G'_{\text{root}} \Rightarrow_{(r_m r_l)^*} G' \Rightarrow_{r_l} G$ generate DAGs in $L(\mathcal{G})$. Those derivations include only two meta-states $\underline{G}_{\text{root}} = q^2$ and $\underline{G}_{\text{root}} = \underline{G'} = q$. Unfortunately, this does not yield fully balanced trees.

If we consider the fully balanced binary trees of depth $n$, where every leaf is the $n$th vertex in a path from the root to the leaf, $\mathcal{G}$ has to generate at least $n-2$ vertices labeled `m` or $n-1$ vertices not labeled `l` until it can apply the rule $r_l$. This, in turn, means that $n-2$ derivation steps increment the meta-state. The generation of a complete binary tree results in a meta-state $q^n$. For generating the language of binary trees of all depths, $\mathcal{Q} = \{q^i \mid i \in [n]\}$. But this set is minimal, since no other derivations exist, and infinite which shows that $\mathcal{Q}_{\min}(\mathcal{G}_{tree})$ is infinite. Note, $\mathcal{G}_{tree}$ is a deterministic and minimal grammar. Thus, minimal deterministic DAG grammars with finite $\mathcal{Q}_{\min}$ exist. □

After proving the existence of languages in the language class `ID`, we present languages in this class. The following theorem about certain tree languages follows directly from Lemma 5.2.

**Theorem 5.3** (Trees in `ID`)**.** *A minimal deterministic DAG grammar $\mathcal{G} = (N, \Sigma, R)$ generating fully balanced trees of unbounded size, $L(\mathcal{G})$ is in* `ID`.

*Proof.* For $L(\mathcal{G})$ comprising fully balanced trees with one root, $L(\mathcal{G}) \in$ `ID` was shown to be true in Lemma 5.2. For trees "upside down", one leaf and many roots, the argumentation is similar. Instead every derivation step $\Rightarrow_{r_m}$ for a path from a root to a leaf adds a state to produce the ensuing meta-state. In doing so, it forces dependencies to be derived. The top-down generation of $\mathcal{G}$ prevents the isolated derivation of a root-to-leaf path. A vertex at depth $n$ requires $n$ roots. Therefore, the size of the meta-states depends on $n$, yielding infinitely many meta-states. Thus, the dependencies are responsible for the infinite class $\mathcal{Q}_{\min}(\mathcal{G})$. □

Trees are fine, but what about graphs? We observed (Obs. 2.8) that only undirected rule cycles can yield cycles in DAGs. Directed trees are generated by directed rule cycles. So let's consider languages with proper graphs, thus generated by undirected rule cycles.

**Lemma 5.4** (DAG Cycles in `ID`)**.** *Let $\mathcal{G}$ be a minimal deterministic DAG grammar. Then $L(\mathcal{G}) \in$ `ID` if some DAG $G \in L(\mathcal{G})$ contains a cycle of arbitrary length that exhibits a chord path.*

*Proof.* As Lemma 5.2 tells us, generating a cycle of arbitrary length causes $\mathcal{Q}_{\mathbf{0}}(\mathcal{G})$ to be infinite, but not necessarily $\mathcal{Q}_{\min}(\mathcal{G})$. Hence, the infinite size of $\mathcal{Q}_{\min}(\mathcal{G})$ hides in the rule cycle's chord path.

In a DAG, a cycle of arbitrary length $m \in \mathbb{N}$ requires the repeated application of an undirected rule cycle (Obs. 2.8) in an unbounded number of iteration steps, say, in $n \in \mathbb{N}$ iteration steps, since the set of rules in $\mathcal{G}$ is finite and, in order to obtain a cycle of arbitrary length $m$, as stated, by the pigeonhole principle, we need an unbounded number $n$ of applications of the required rule cycle.

Let $c$ be this undirected rule cycle that is required. With the constant $|c|$ denoting the number of rules $c$ consists of, the length of the DAG's undirected cycle is $m = |c| \cdot n$.

Consider the DAG $G$ generated by $2n$ rule cycle iterations of $c$, yielding a cycle of length $2 \cdot |c| \cdot n$, as detailed in the following.

As stated, the cycle in $G$ has a chord path. However, if one application of $c$ generated the stated chord path, a graph generated by many applications of $c$, could, if 'wired' appropriately, contain many such chord paths instead of just one. The grammar $\mathscr{G}$ generated $G$ by $2n$ applications of $c$. Consequently, $\mathscr{G}$ can generate $G$ in a way that it exhibits $2n$ such chord paths, thus, with identical edge labels as the stated chord path exhibits. The identical labeling is key for the proof, since this will allow 'wiring' the chords differently.

By definition (Def. 2.7), a rule cycle includes marked rules with a minimum of two nonterminals with two of them marked. However, obviously, the end vertices of a chord path have a vertex degree of at least three [34]. Due to that, the marked rules generating the chord path need at least three nonterminals and that means one unmarked nonterminal. Thus, a chord path requires an unmarked nonterminal. An unmarked nonterminal needs to be *saved* in a meta-state. Let us call this nonterminal $s$. Assume that $\mathscr{G}$ opens and closes the chord paths one after another while generating $G$. Then $\mathscr{G}$ could possibly reuse the meta-state in which we stored $s$: when it opens the chord path, it stores $s$ in the ensuing meta-state, when it closes the chord path, it does not need to save $s$ anymore. Then, $\mathscr{G}$ repeats this step for the next chord path, reusing the meta-state in which it stored $s$.

Yet, Lemma 2.11 allows the swapping of the $2n$ chord paths. In order to obtain one of those graphs in $L(\mathscr{G})$ requiring an infinite $\mathscr{Q}_{\min}(\mathscr{G})$, let us swap the chord paths in $G$ (technically more precise: certain edges of the chord paths). Let $e_i$, labeled with the afore mentioned $\ell(e_i) = s$, be one of the ends of the chord path number $i$ for $i \in [2n]$. Swapping $G[e_1 \bowtie e_n]$, $G[e_2 \bowtie e_{n-1}]$, $\ldots$, thus $G[e_k \bowtie e_{n-k+1}]$ for $k \in [n/2]$ results in the first $n$ chord paths bridging a distance of length $|c| \cdot n$ of its DAG cycle.

Please note: Swapping the second half of the chord paths is unnecessary, but can be performed in order to obtain a symmetric, beautiful graph. If we would, however, swap the $2n$'th chord path with the first, the distance would be one length of $c$, since the chords are arranged on a cycle. First and $2n$'th chord paths are neighbors, thus. That is the reason for choosing $2n$ instead of $n$ cycle iterations of $c$. The first and the $n$'th chord paths are a maximum distance apart with respect to the cycle generated by $2n$ applications of $c$.

After these swapping operations, the chords are nested: $\mathscr{G}$ opens the first chord path before the second but closes the second before the first, and so on, according to the FILO (first in, last out) principle. This constitutes the proof: by opening $n/2$ chord paths, all identically labeled with $s$, there exists a meta-state in $\mathscr{Q}_{\min}(\mathscr{G})$ that has to store all those $s^{n/2}$ labels. Then, $\mathscr{G}$ has to remember that all these edges of the chord paths are dangling. Thus, the number of meta-states depends on the number of cycle iterations $2n$, which completes the proof, since $n \in \mathbb{N}$. □

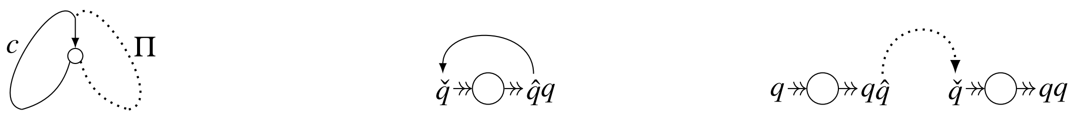
## 6 Characterization

Now we contemplate the newly found language classes which divide the class of top-down deterministic regular DAG languages $RDL^{\text{det}}$, whose languages are generated by deterministic DAG grammars, into disjoint subclasses.

**Observation 6.1** ($\mathtt{FID} \cup \mathtt{ID} = RDL^{\text{det}}$)**.** *By the definitions of* `FID` *and* `ID`*, it is true that* `FID` *and* `ID` *are disjoint and that* $\mathtt{FID} \cup \mathtt{ID} = RDL^{det}$.

The following lemma characterizes the language class `ID` by the rules of a grammar. Since `ID` and `FID` are disjoint, indirectly it is also a characterization of `FID`.

**Theorem 6.2** (Characterization of `ID`)**.** *For a minimal deterministic DAG grammar* $\mathscr{G} = (N, \Sigma, R)$ *its set of meta-states* $\mathscr{Q}_{min}(\mathscr{G})$ *is infinite iff there exist a rule cycle c, a rule path* $\Pi$ *and not necessarily distinct nonterminals* $q, p \in N$ *satisfying the following conditions:*



(a) Rule cycles' graph (b) Rule cycle $c$ – left branch (c) Rule path $\Pi$ – right branch

Figure 5: Rule cycles and their graphs for the binary tree language given by a cycle and its chord

- *The nonterminals $q$ and $p$ occur in $c$ as unmarked $q \in N$ and marked $\bar{p} \in \bar{N}$.* $(c)$
- *The rule path $\Pi$ lies between $q$ and $p$.* $(\Pi)$
- *The nonterminal $q$ is in $c$ in a head iff it is in $\Pi$'s weak rule in the head.* $(c\Pi)$

*Proof.* Before proving the stated bi-implication, observe that the above conditions result in two distinct rule cycles. It is easy to see that $\Pi$ acts as a chord path for the rule cycle $c$, and consequently, provides an alternative rule cycle. Fig. 5 demonstrates an example. For the rule cycle $c$ condition $(c)$ and for the rule path $\Pi$ condition $(\Pi)$ urge the suitable nonterminals for gluing $\Pi$ as a chord to $c$. The requirements stated in $(c\Pi)$, concerning the nonterminals' positions in head or tail ensure the right orientation of the edge labeled $q$. And, this alternative rule cycle provided by $\Pi$ is the crux of the matter. Since the grammar has always two options to continue the derivation, it always has to remember the dangling edges of the alternative it did not decide to generate next. Thus, in every cycle iteration, regardless of the decision how to proceed, a new meta-state is needed since the number of dangling edges grows in an unbounded fashion.

We start with assuming that $\mathscr{G}$'s rules have the stated form and then prove that $\mathscr{Q}_{\min}(\mathscr{G})$ is infinite. The proof by construction relies on two techniques. We construct a graph $G_n$ for which the number of meta-states in $\mathscr{Q}_{\min}(\mathscr{G})$ needed to generate it depends on the number of rule cycle iterations $n \in \mathbb{N}$. The first technique is the (directed or undirected) path whose length depends on $n$: Our grammar $\mathscr{G}$ can generate the above stated DAG $G_n$ since a rule cycle can yield a path of length $k \cdot n$ where $k$ is the number of edges generated by the marked nonterminals in one cycle iteration. A path as such is connected, and our path is a subgraph of a connected prefix DAG, generated top-down by $\mathscr{G}$. As such, it is a connected prefix DAG which, according to Lemma 3.4, $\mathscr{G}$ can complete to a language-useful DAG. The second technique is to disregard all the edges and vertices not mentioned in the conditions stated about the rules of the grammar. Obviously, they are irrelevant since, due to the swapping operation (Theorem 2.11), they would only *increase* the size and number of the meta-states.[6] Like that, $\mathscr{G}$ causes $\mathscr{Q}_{\min}$ to be infinite by deriving $G_n$ with a path of length $n$ as a subgraph via:

- *directed* rule cycles and therefore fully balanced binary trees as subgraphs (Lemma 5.3)
- *undirected* rule cycles and therefore proper DAGs as shown in Lemma 5.4.

Consequently, $\mathscr{G}$ generates a DAG $G_n$ whose size and number of meta-states depend on the number of rule cycle iterations which are unbounded. Like that, $\mathscr{Q}_{\min}(\mathscr{G})$ is infinite.

The direction vice versa assumes an infinite $\mathscr{Q}_{\min}(\mathscr{G})$ and arguments by negating the conditions one by one: The existence of the stated rule cycle and rule path, as well as the requirements $(c)$, $(\Pi)$ and $(c\Pi)$ have to meet are iteratively shown to hold by negating them and concluding a contradiction.

- Assume no $\Pi$ exists. By their definition, grammars without rule paths do not generate DAGs. With $\mathscr{Q}_{\min}(\mathscr{G})$ being infinite, a rule path must exist. Assume no $c$ exists. Languages without a

[6]Note that, with this second technique applied, $k$ – the number of edges generated by the marked nonterminals in one cycle iteration – equals to one. We abstract away how many edges the cycle exactly generates. Relevant to distinguish between finite and infinite $\mathscr{Q}_{\min}$ is not the constant factor $k$, but only the number of iterations $n$.

rule cycle are finite (Lemma 2.9). Finite languages have a finite $\mathscr{Q}_{\min}$ (Lemma 4.2), contradicting our assumption that no rule cycle exists. Infinite languages are necessarily generated by cycles (Lemma 2.9). Thus, both rule path and cycle, called $\Pi$ and $c$, exist.

- Assume condition $(c)$ does not hold. By definition, no rule cycles without marked nonterminals exist, thus $\bar{p}$ must occur in $c$. A rule cycle without an unmarked nonterminal, here $q$, yields finitely many meta-states (Lemma 4.4, 3.). Condition $(c)$ holds.
- Without condition $(\Pi)$, thus without $\Pi$ being connected with its both ends $q$ and $p$ to $c$, there would be no chord path[7]. First, assume that there would be no rule path with $q$ as one of its ends. However, it is immediate that some rule path is indeed connected to $c$: Since $q$ occurs unmarked in $c$, a rule path $\Pi$ connected by $q$ necessarily exists, yielding – without the chord – an infinite set of meta-states, indeed. But, it is $\mathscr{Q}_{\mathbf{0}}(\mathscr{G})$ which is infinite, not $\mathscr{Q}_{\min}(\mathscr{G})$ – by Lemma 4.4 again – in case $\Pi$ does not lead back to $c$ via $\bar{p}$, our second assumption when negating $(\Pi)$. This contradiction shows that a path $\Pi$ between $q$ and $p$ must exist for an infinite $\mathscr{Q}_{\min}$.
- Negating condition $(c\Pi)$ means allowing $q$ as unmarked in a head of $c$ while it is in a tail of $\Pi$'s two weak rules or vice versa. This would induce the wrong orientation of $q$ so that $\Pi$ would not be a chord. And, we already know from the previous point that $\Pi$ has to be a chord.

We cannot sacrifice any condition without contradicting our assumption of an infinite $\mathscr{Q}_{\min}(\mathscr{G})$ which proves the second direction and by that completes the proof. □

What if we restrict a grammar $\mathscr{G}$ with a set of meta-states $\mathscr{Q}$ instead of deriving this set out of the grammar? In that case the grammar's language possibly changes. Whereas when just extracting $\mathscr{Q}_{\min}(\mathscr{G})$, the grammar's language is not altered.

**Definition 6.3** (Finite meta-state DAG language (FD))**.** *A minimal deterministic grammar* $\mathscr{G} = (N, \Sigma, R)$ *generates a* finite meta-state language $L^{\mathscr{Q}}(\mathscr{G})$ *where a finite set* $\mathscr{Q} \subseteq \mathbb{N}^{Q}$ *is given to restrict which rules in* $R$ *can be used. The derivation step* $G_1 \Rightarrow G_2$ *is only allowed if the meta-state* $\underline{G_2} \in \mathscr{Q}$*.*

Languages in the classes `FID` and `FD` can use their finite sets of meta-states, $\mathscr{Q}_{\min}$ and $\mathscr{Q}$, respectively, to construct a classical finite state automaton to recognize themselves.

## 7 Classical Finite State Automata for DAG Languages

This section describes how classical finite state automata come into play when recognizing certain DAG languages. An `FD` language can be recognized by a classical deterministic finite state automaton (DFA). As an example, see Figure 3b which shows the automaton for accepting DAGs of $L(\mathscr{G}_{star})$ as defined in Example 3.6. The top-down reading process induces merely a partial order on the vertices. The deterministic automaton thus reads a DAG in a partly nondeterministic fashion.

A DFA is a five-tuple $A = (Q, \Gamma, \delta, q_0, q_\$)$ with the finite sets $Q$ and $\Gamma$ being the states and the alphabet, resp., $q_0, q_\$ \in Q$ being the start and final state, resp., and $\delta : (Q \times \Gamma) \to Q$ being the transition function, extended inductively to strings $\delta(Q \times \Gamma^*) \to Q$ by applying $\delta$ symbol-wise. We omit entries of $\delta$ if they do not lead to an accept state and thus consider only partial DFAs.

Every DAG grammar $\mathscr{G} = (N, \Sigma, R)$ with $L(\mathscr{G})$ being a finite meta-state DAG language gives rise to a DFA $M = (R, \mathscr{Q}_{\min}(\mathscr{G}), \delta, \emptyset, \emptyset)$ such that for all $\mathbb{q} \in \mathscr{Q}_{\min}$ the transition function yields the following if $\alpha \subseteq \mathbb{q}$ then

$$\delta(\mathbb{q}, (\alpha \twoheadrightarrow \textcircled{\sigma} \twoheadrightarrow \beta)) = (\mathbb{q} \setminus \alpha) \cup \beta.$$

[7]Recall the introduction of the proof for the explanation of $\Pi$ being a chord path.

Although $M$ meets the requirements of a DFA, we call its states in $\mathscr{Q}_{\min}$ meta-states instead to avoid confusion. While reading a string with a DFA is as easy as reading it symbol by symbol, reading a DAG is somewhat more complicated since the vertices do not exhibit an obvious total order. Therefore, $M$ reads the vertices partly nondeterministically. Let $G = (V, E, \ell, in, out)$ be a DAG in $\mathscr{D}_\Sigma$. Such a DAG has no labels assigned to its edges. We denote this by $\ell(e) = \#$. Since we are restricted to top-down procedures, reading a vertex means assigning states to the outgoing edges: Consequently, $M$ may read a vertex only if all of its ingoing vertices have been assigned labels, thus $\# \notin [\ell(in)]$. If $\ell(v)$ matches the $\sigma$ of a rule $r = (\alpha \twoheadrightarrow \textcircled{\sigma} \twoheadrightarrow \beta)$ such that $\delta(\mathtt{q}, r)$ yields a new state $\mathtt{q}'$ of $M$, $M$ can read $v$ and assigns the labels $\beta$ to its outgoing edges: $\ell(out(v)) = \beta$.[8]

By reading the vertices top-down, $M$ will accept a DAG if it nondeterministically chooses the right order of vertices. The order in which the vertices are read is already restricted by imposing the requirement to read top-down. But, we can improve upon this by fine-tuning the order in which the roots are read. When $M$ cannot read any non-root, then there exists no vertex with all its ingoing edges labeled. Instead of choosing an arbitrary root next – since in a top-down a root has no prerequisites and can always be read top-down – $M$ can choose a root which is needed next. Which root $v_{\text{root}} \in V$ do we need next? One where the DAG $G$ has a path from $v_{\text{root}}$ to a vertex $v_{\text{next}}$ whose ingoing vertices are labeled as well as not labeled.

We defined the language class `FD` at the end of the last section (c.f. Definition 6.3). Now that we know how a grammar $\mathscr{G}$ in combination with a finite set of meta-states $\mathscr{Q}(\mathscr{G})$ can determine a DFA for the DAG language $L(\mathscr{G})$, let us come back to the idea of restricting a grammar with a set of meta-states not derived from $\mathscr{G}$. Since languages easily become `ID` languages, the rules which are allowed to remain in the class of `FID` limit the expressiveness of languages. The motivation behind restricting via a finite $\mathscr{Q}$ strives to increase the expressiveness of a language while keeping the number of meta-states to recognize or generate it finite in order to profit from transferring the good algorithmic properties of regular string languages. We use the language which comprises DAGs looking like a rainbow to illustrate restricting a given `ID` language to become an `FD` language recognizable by an FSA, see Figure 6.

**Theorem 7.1** (FD $\nsubseteq$ $RDL^{\text{det}}$)**.** *The class of finite meta-state DAG languages is not a subset of the class of top-down deterministic regular DAG languages $RDL^{det}$.*

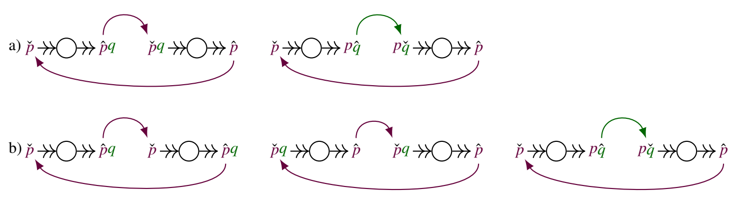
*Proof.* Given is a minimal deterministic DAG grammar $\mathscr{G} = (N, \Sigma, R)$ and a finite set of meta-states $\mathscr{Q}$. If FD $\nsubseteq$ $RDL^{\text{det}}$ holds, then there exists a DAG language $L^{\mathscr{Q}}(\mathscr{G})$ that is not in the class of $RDL^{\text{det}}$. Our $\mathscr{G}$ is a deterministic DAG grammar and as such generates a language $L(\mathscr{G}) \in RDL^{\text{det}}$.

We try to construct a language not in $RDL^{\text{det}}$ by limiting $\mathscr{G}$'s language to a finite one. Suppose, $\mathscr{G}$ generates an infinite language $L(\mathscr{G})$ and $\mathscr{Q}$ prevents derivations of any rule cycle in $R$. According to Lemma 2.9, then, $L^{\mathscr{Q}}(\mathscr{G})$ will be finite. However, Lemma 4.2 tells us that any finite language can be generated by a deterministic DAG grammar $\mathscr{G}' \neq \mathscr{G}$ such that $L(\mathscr{G}') = L^{\mathscr{Q}}(\mathscr{G})$. This attempt did not work out.

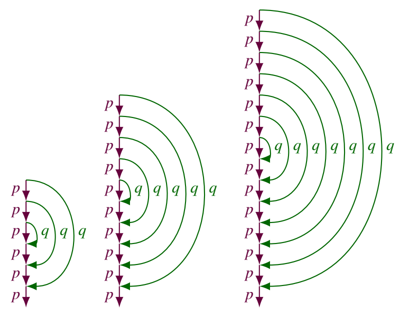
Our next attempt is the grammar $\mathscr{G}_{\text{bow}}$ with its infinite language $L(\mathscr{G}_{\text{bow}}) \in$ `ID`. Its rule set comprises $(\lambda \twoheadrightarrow \textcircled{\texttt{r}} \twoheadrightarrow pq), (p \twoheadrightarrow \textcircled{\texttt{o}} \twoheadrightarrow pq), (pq \twoheadrightarrow \textcircled{\texttt{c}} \twoheadrightarrow p)$ and $(pq \twoheadrightarrow \textcircled{\texttt{l}} \twoheadrightarrow \lambda)$. The grammar $\mathscr{G}_{\text{bow}}$ can among others generate DAGs similar to garlands and rainbows. To generate an arbitrary long DAG looking like a garland, the set of meta-states $\mathscr{Q} = \{\varnothing, p, pq\}$ suffices. If, however, $\mathscr{G}_{\text{bow}}$ generates a DAG looking like a rainbow, it repeats the rule with the vertex label `o` an unbounded number of times.[9] Such a rainbow DAG in

[8] Note that, by definition, the empty DAG $\varnothing$ is not in the language although the automaton accepts it due to the start state as accepting state. Only DAGs $G \in D_\Sigma$ are considered and $\varnothing$ is not in $D_\Sigma$.

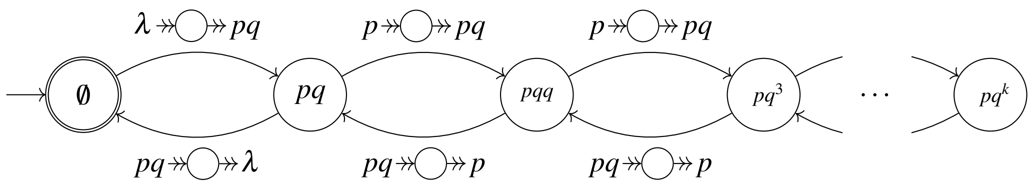
[9] Allowing pennants spanning more than one vertex is possible, too. With $\mathscr{Q} = \{pq^n \mid p, q \in N \text{ and } n \in \mathbb{N}\}$ the bow(s) for the pennants in a DAG $G_{garland} \in L^{\mathscr{Q}}(\mathscr{G})$ can span the maximum of $n$ vertices.



(a) The rule cycles in $\mathscr{G}_{\text{bow}}$



(b) Some DAGs in $L(\mathscr{G}_{\text{bow}})$



(c) An FSA for the language $L^{\{pq^k \mid k>0\}}(\mathscr{G}_{\text{bow}})$ with a fixed $k \in \mathbb{N}$.

Figure 6: The grammar $\mathscr{G}_{\text{bow}}$ generates, among others, DAGs looking like rainbows (*vertices implicit*) (a) via the rule cycles in (b). For a fixed $k \in \mathbb{N}$, restricting the grammar by $\mathscr{Q} = \{pq^k \mid k > 0\}$ allows the construction of an FSA and imposes a bound on the number of bows in a 'rainbow'.

$L^{\mathscr{Q}}(\mathscr{G})$, with the length of the bow(s) limited, can only be generated by a deterministic DAG grammar if the grammar's language is finite. But, a finite language cannot include DAGs with $n$-sized bows of arbitrary length. On the other hand, any grammar generating garland DAGs of unbounded length will also include the rainbow DAGs due to the possibility to swap the edges (Lemma 2.11). And, according to Lemma 4.6, we can swap the edges since the edge labels (the nonterminals) must be repeated for a DAG of unbounded length and distinct edges labeled with the same nonterminal in a derivation DAG can be swapped. Thus, restricting the language to $L^{\mathscr{Q}}(\mathscr{G}_{\text{bow}})$ by above given $\mathscr{Q}$ results in a language not in $RDL^{\text{det}}$, completing the proof. □

Again, swapping (Lemma 2.11) would allow us to generate unbounded rainbows, so we conclude: restricting a grammar by a set of meta-states can prevent the swapping operation.

**Corollary 7.2** (Swapping in `FD`)**.** *For a grammar $\mathscr{G}$ with its $L^{\mathscr{Q}}(\mathscr{G}) \in \mathtt{FD} \setminus \mathtt{FID}$ holds:*

- $\exists \lfloor D \rfloor \in L^{\mathscr{Q}}(\mathscr{G}) : \lfloor D(e_0 \bowtie e_1) \rfloor \notin L^{\mathscr{Q}}(\mathscr{G})$
- $|L^{\mathscr{Q}}(\mathscr{G})| < |L(\mathscr{G})|$
- $L(\mathscr{G}) = \{ \lfloor D(e_0 \bowtie e_1) \rfloor \mid D(e_0 \bowtie e_1) \in L^{\mathscr{Q}}(\mathscr{G}) \}$
- $L(\mathscr{G}) \in \mathtt{ID}$

Contrary to `FID`-grammars on whose derivation DAGs swapping is always allowed, the derivation DAGs of grammars generating languages in $L^{\mathscr{Q}}(\mathscr{G}) \in \mathtt{FD} \setminus \mathtt{FID}$ the swapping operation is restricted by the given set of meta-states $\mathscr{Q}$. Restricting a grammar by allowing only certain meta-states corresponds to restricting swapping on the derivation DAGs. Via those restrictions, DAGs are lost which cannot be accepted without the missing meta-states resulting in a language with less graphs. The transitive closure of the swapping operation on the language $L^{\mathscr{Q}}(\mathscr{G})$ returns the original language $L(\mathscr{G})$ which must be in the class `ID`.

For free – by `FD` definition – we can observe the closure properties valid for DFAs, since `FD` languages are recognized by DFAs.

**Observation 7.3** (`FD` – Closure under Union and Intersection)**.**
*The language class* `FD` *is closed under union and intersection.*

## 8 Conclusion

We have defined the DAG language classes `ID` and `FD` and characterized them. By imposing the set of meta-states as a given restriction, we additionally have defined `FD`, which intersects with *RDL* but is not a subset of it – adding to expressiveness. For languages in `FID` and `FD`, we proved that it is possible to construct a classical string automaton recognizing the language. Analysing `ID` further, we expect similarities with the Chomsky hierarchy.

**Pushdown Conjecture.** *Analogous to languages in* `FD` *being recognized by a finite state automaton, we conjecture all languages in* `ID` *to be recognized by a pushdown automaton.*

The FSA construction by meta-states, could be applied not only to the top-down deterministic version, but also to the plain regular DAG automaton. By dropping the determinism restriction, similar to dropping the planarity restriction [32] imposed in [21], possibly the NP-completeness for the membership problem could be tackled. Imposing useful restrictions to be provided by the set of meta-states $\mathscr{Q}$, as the language class `FD` requires it, is the task for more application centric research, like semantic NLP parsing [29, 3].

## References

[1] Lene Antonsen, Erik Axelson, Eckhard Bick, Børre Gaup, Sam Hardwick, Katri Hiovain-Asikainen, Arvi Hurskainen, Fred Karlsson, Kimmo Koskenniemi, Krister Lindén, Inari Listenmaa, Inga Mikkelsen, Sjur Nørstebø Moshagen, Flammie A Pirinen, Aarne Ranta, Jack Rueter, Daniel G. Swanson, Trond Trosterud & Linda Wiechetek (2023): *Rule-Based Language Technology*. *NEJLT Monographs* 2, Northern European Association for Language Technology (NEALT). Available at `http://hdl.handle.net/10062/89595`.

[2] Andrew Badr (2009): *HYPER-MINIMIZATION IN O($n^2$)*. *Int. J. Found. Comput. Sci.* 20(4), pp. 735–746, doi:10.1142/S012905410900684X.

[3] Laura Banarescu, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer & Nathan Schneider (2013): *Abstract Meaning Representation for Sembanking*. In: *Proc. 7th Linguistic Annotation Workshop, ACL 2013 Workshop*.

[4] Darcy Best & Max Ward (2022): *A faster algorithm for maximum independent set on interval filament graphs*. *J. Graph Algorithms Appl.* 26(1), pp. 199–205, doi:10.7155/JGAA.00588.

[5] Johannes Blum & Frank Drewes (2019): *Language theoretic properties of regular DAG languages*. *Inf. Comput.* 265, pp. 57–76, doi:10.1016/j.ic.2017.07.011. Available at `https://doi.org/10.1016/j.ic.2017.07.011`.

[6] Tommaso Boccato, Matteo Ferrante, Andrea Duggento & Nicola Toschi (2024): *4Ward: A relayering strategy for efficient training of arbitrarily complex directed acyclic graphs*. *Neurocomputing* 568, p. 127058, doi:10.1016/J.NEUCOM.2023.127058.

[7] Dietrich Braess (1968): *Über ein Paradoxon aus der Verkehrsplanung*. *Unternehmensforschung* 12(1), pp. 258–268, doi:10.1007/BF01918335.

[8] David Chiang, Frank Drewes, Daniel Gildea, Adam Lopez & Giorgio Satta (2018): *Weighted DAG Automata for Semantic Graphs*. *Computational Linguistics* 44(1), doi:10.1162/COLI_a_00309.

[9] Marco Damonte & Shay B. Cohen (2018): *Cross-lingual Abstract Meaning Representation Parsing*. In: *Proceedings of NAACL*, doi:10.18653/v1/N18-1104.

[10] Marco Damonte, Shay B. Cohen & Giorgio Satta (2017): *An Incremental Parser for Abstract Meaning Representation*. In: *Proceedings of EACL*, doi:10.18653/v1/E17-1051.

[11] Martin Dias, Mariano Martinez Peck, Stéphane Ducasse & Gabriela Arévalo (2014): *Fuel: a fast general purpose object graph serializer*. *Softw. Pract. Exp.* 44(4), pp. 433–453, doi:10.1002/SPE.2136.

[12] Frank Drewes (2017): *DAG Automata for Meaning Representation*. In Makoto Kanazawa, Philippe de Groote & Mehrnoosh Sadrzadeh, editors: *Proceedings of the 15th Meeting on the Mathematics of Language, MOL 2017, London, UK, July 13-14, 2017*, ACL, pp. 88–99, doi:10.18653/v1/w17-3409.

[13] Frank Drewes (2017): *On DAG Languages and DAG Transducers*. *Bulletin of the EATCS* 121.

[14] Senka Drobac, Krister Lindén, Tommi A. Pirinen & Miikka Silfverberg (2014): *Heuristic Hyperminimization of Finite State Lexicons*. In Nicoletta Calzolari, Khalid Choukri, Thierry Declerck, Hrafn Loftsson, Bente Maegaard, Joseph Mariani, Asunción Moreno, Jan Odijk & Stelios Piperidis, editors: *Proceedings of the Ninth International Conference on Language Resources and Evaluation, LREC 2014, Reykjavik, Iceland, May 26-31, 2014*, European Language Resources Association (ELRA), pp. 3319–3324. Available at `http://www.lrec-conf.org/proceedings/lrec2014/summaries/784.html`.

[15] Vida Dujmovic & Pat Morin (2019): *Dual Circumference and Collinear Sets*. In Gill Barequet & Yusu Wang, editors: *35th International Symposium on Computational Geometry, SoCG 2019, June 18-21, 2019, Portland, Oregon, USA*, *LIPIcs* 129, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 29:1–29:17, doi:10.4230/LIPIcs.SoCG.2019.29.

[16] Anjan Dutta, Josep Lladós & Umapada Pal (2013): *A symbol spotting approach in graphical documents by hashing serialized graphs*. *Pattern Recognit.* 46(3), pp. 752–768, doi:10.1016/J.PATCOG.2012.10.003.

[17] Krister Lindén Erik Axelson, Sam Hardwick (2023): *HFST Training Environment and Recent Additions (61-69) pdf*, pp. 60–69. 2 of Hurskainen et al. [1]. Available at `http://hdl.handle.net/10062/89595`.

[18] Akio Fujiyoshi (2010): *Recognition of directed acyclic graphs by spanning tree automata*. *Theor. Comput. Sci.* 411(38-39), pp. 3493–3506, doi:10.1016/j.tcs.2010.06.006.

[19] Abdullah Gharaibeh, Elizeu Santos-Neto, Lauro Beltrão Costa & Matei Ripeanu (2013): *Efficient Large-Scale Graph Processing on Hybrid CPU and GPU Systems*. *CoRR* abs/1312.3018. arXiv:1312.3018.

[20] Annegret Habel & Hans-Jörg Kreowski (1986): *May we introduce to you: hyperedge replacement*. In Hartmut Ehrig, Manfred Nagl, Grzegorz Rozenberg & Azriel Rosenfeld, editors: *Graph-Grammars and Their Application to Computer Science, 3rd International Workshop, Warrenton, Virginia, USA, December 2-6, 1986*, *Lecture Notes in Computer Science* 291, Springer, pp. 15–26, doi:10.1007/3-540-18771-5_41.

[21] Tsutomu Kamimura & Giora Slutzki (1981): *Parallel and Two-Way Automata on Directed Ordered Acyclic Graphs*. *Information and Control* 49, pp. 10–51, doi:10.1016/S0019-9958(81)90438-1.

[22] Tsutomu Kamimura & Giora Slutzki (1982): *Transductions of Dags and Trees*. *Mathematical Systems Theory* 15(3), pp. 225–249, doi:10.1007/BF01786981.

[23] Krister Lindén, Tommi Pirinen et al. (2009): *Weighting finite-state morphological analyzers using hfst tools*. In: *Finite-State Methods and Natural Language Processing-FSMNLP 2009 Eight International Workshop*.

[24] Andreas Maletti & Daniel Quernheim (2011): *Optimal Hyper-Minimization*. *Int. J. Found. Comput. Sci.* 22(8), pp. 1877–1891, doi:10.1142/S0129054111009094.

[25] Akira Matsubayashi & Yushi Saito (2023): *A Faster Algorithm for Recognizing Directed Graphs Invulnerable to Braess's Paradox*. In Daniele Frigioni & Philine Schiewe, editors: *23rd Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems, ATMOS 2023, September 7-8, 2023, Amsterdam, The Netherlands*, *OASIcs* 115, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 12:1–12:19, doi:10.4230/OASICS.ATMOS.2023.12.

[26] Sjur Nørstebø Moshagen, Flammie Pirinen, Lene Antonsen, Børre Gaup, Inga Mikkelsen, Trond Trosterud, Linda Wiechetek & Katri Hiovain-Asikainen (2023): *The GiellaLT infrastructure: A multilingual infrastructure for rule-based NLP*. 2 of Hurskainen et al. [1]. Available at `http://hdl.handle.net/10062/89595`.

[27] Flammie A Pirinen (2023): *Finite-State Technology in Rule-Based Natural Language Processing*, pp. 49–59. 2 of Hurskainen et al. [1]. Available at `http://hdl.handle.net/10062/89595`.

[28] Tommi Pirinen, Krister Lindén et al. (2010): *Finite-state spell-checking with weighted language and error models*. In: *Proceedings of LREC 2010 Workshop on Creation and use of basic lexical resources for less-resourced languages*.

[29] Daniel Quernheim & Kevin Knight (2012): *Towards Probabilistic Acceptors and Transducers for Feature Structures*. In: *Proc. 6th Workshop on Syntax, Semantics and Structure in Statistical Translation*, Association for Computational Linguistics, pp. 76–85.

[30] Eloize Rossi Marques Seno, Helena de Medeiros Caseli, Marcio Lima Inácio, Rafael T. Anchiêta & Renata Ramisch (2022): *XPTA: um parser AMR para o Português baseado em uma abordagem entre línguas*. *Linguamática* 14(1), pp. 49–68, doi:10.21814/lm.14.1.359.

[31] Daniel G. Swanson (2023): *Apertium*, pp. 95–111. 2 of Hurskainen et al. [1]. Available at `http://hdl.handle.net/10062/89595`.

[32] Ieva Vasiljeva, Sorcha Gilroy & Adam Lopez (2018): *The problem with probabilistic DAG automata for semantic graphs*. *CoRR* abs/1810.12266. arXiv:1810.12266.

[33] Magnus Wahlström (2017): *LP-branching algorithms based on biased graphs*. In Philip N. Klein, editor: *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, SIAM, pp. 1559–1570, doi:10.1137/1.9781611974782.102.

[34] Douglas B. West (2001): *Introduction to Graph Theory*, 2 edition. Prentice Hall.

## 9 Appendix

**Lemma 9.1** (Meta-state dependent)**.** *Let $G, G'$ be prefix DAGs with $\underline{G} = \underline{G'}$ derived by a DAG grammar $\mathcal{G} = (N, \Sigma, R)$. Then, $\mathcal{G}$ can apply the rule $r \in R$ as the next derivation step $G \Rightarrow_r H$ iff $G' \Rightarrow_r H'$. Similarly, $G$ is useful if and only if $G'$ is useful. However, if $G$ is language-useful this only implies that $G'$ is useful.*

*Proof of Theorem 3.3.* By the definition of a DAG grammar, a rule application depends only on the temporary vertices labeled by non-terminals – elements of the meta-states $\underline{G}$ and $\underline{G'}$. Consequently, $G \Rightarrow_r H$ is possible iff $G' \Rightarrow_r H'$ is. Valuability requires a rule sequence and is therefore also dependent on the meta-state only. It follows that $G$ is useful if and only if $G'$ is useful. On the other hand, a common meta-state does not preserve connectivity. With the given equality $\underline{G} = \underline{G'}$, $G$ could be a connected DAG while $G'$ is not. In that case $G$ is language-useful, but $G'$ not necessarily – only if the preceding derivation connects the unconnected components of $G'$. But since the language-useful $G$ is useful, $G'$ is, too. □

**Lemma 9.2** (Useful Prefix DAG)**.** *Let $G$ be a prefix DAG and $\mathcal{G} = (N, \Sigma, R)$ be a DAG grammar without useless rules. If $\varnothing \Rightarrow_R^* G$, then $G$ is* useful with respect to $\mathcal{G}$*, i.e. there exists a derivation $\varnothing \Rightarrow_R^* G \Rightarrow_R^* G'$ for a complete DAG $G'$. If $G'$ is connected it is* language-useful.

*Proof of Theorem 3.4.* We prove this by induction on the length $n$ of the derivation $\varnothing \Rightarrow_{r_1 \ldots r_n} G$. For the base case consider a derivation $\varnothing \Rightarrow_r G$ of length $n = 1$. As the rule set $R$ contains no useless rules, there is a derivation $\varnothing \Rightarrow_{r'_1 \ldots r'_k} G'$ where $G' \in D_\Sigma$ and $r = r'_i$ for some $i$. Moreover, as $\varnothing \Rightarrow_r G$, the head of the rule $r$ is the empty string and we may assume that $r = r'_1$. This means that $G$ is useful with respect to $\mathcal{G}$ (for the sake of brevity, we say useful subsequently).

Consider now a derivation $\varnothing \Rightarrow_{r_1 \ldots r_n} G$ of length $n > 1$. Let $\rho_1 = r_1 \ldots r_{n-1}$. By the induction hypothesis we know that there is a sequence $\rho_2$ of rules such that $\varnothing \Rightarrow_{\rho_1 \rho_2} H$ for some $H \in D_\Sigma$. Moreover, as $R$ contains no useless rules, there are rule sequences $\rho'_1, \rho'_2$, such that $\varnothing \Rightarrow_{\rho'_1 r_n \rho'_2} H'$ for some $H' \in D_\Sigma$. It is also possible to concatenate these two derivations and to interleave the individual derivation steps. This yields a derivation $\varnothing \Rightarrow_{\rho_1 \rho'_1 r_n} H^* \Rightarrow_{\rho_2 \rho'_2} (H \& H')$ where $H \& H'$ denotes the disjoint union of $H$ and $H'$.

Consider now the derivation $\varnothing \Rightarrow_{\rho_1 r_n} G$. We want to show that $G$ is useful. As $\rho'_1$ can be applied on the empty graph, there is some DAG $G^*$ such that $\varnothing \Rightarrow_{\rho_1 r_n} G \Rightarrow_{\rho'_1} G^*$. Moreover, $G^*$ and $H^*$ have the same meta-state, as they were generated through the same multiset of rules. As $H^*$ is useful it follows from Lemma 3.3, that $G^*$ is useful, and therefore, $G$ is useful.

Therefore, if $G$ is connected, so is the connected component $G''$ in the complete DAG $G'$ which finalized the prefix DAG $G$. This means that $\mathcal{G}$ can decide to choose only those derivation steps which yield a connected DAG $G' = G''$. Consequently, a connected prefix DAG $G$ is not only useful but also language-useful with respect to its grammar $\mathcal{G}$. □

**Lemma 9.3** (Unbounded without Rule Cycle)**.** *There exists a DAG grammar $\mathcal{G} = (N, \Sigma, R)$ and a vertex or edge label* u $\in \Sigma \cup N$ *such that the number of occurrences of* u *in a (derivation) DAG generated by $\mathcal{G}$ is unbounded, although* u *does not occur in any rule cycle.*

*Proof of Theorem 4.5.* Consider the DAG grammar $\mathcal{G} = (\{r, c, l\}, \{\texttt{r}, \texttt{c}, \texttt{l}\}, R)$ containing the following rules $R = \{\lambda \twoheadrightarrow \textcircled{\texttt{r}} \twoheadrightarrow rr,\ rr \twoheadrightarrow \textcircled{\texttt{c}} \twoheadrightarrow c, c \twoheadrightarrow \textcircled{\texttt{c}} \twoheadrightarrow l,\ l \twoheadrightarrow \textcircled{\texttt{l}} \twoheadrightarrow \lambda\}$ and let u $\in \{l, \texttt{l}\}$, thus let u either denote the edge label $l$ or the vertex label $\texttt{l}$. Then, the vertex label $\texttt{l}$ cannot occur in a *marked rule* $\bar{\alpha} \twoheadrightarrow \textcircled{\sigma} \twoheadrightarrow \bar{\beta}$, which, by definition, comprises two marked nonterminals. On the contrary, the only rule with the label

`1` is $l \twoheadrightarrow \textcircled{\texttt{1}} \twoheadrightarrow \lambda$ with $|\alpha\beta| = |l\lambda| = 1$ and thus comprises only one nonterminal. Consequently, `1` cannot occur in a rule cycle since a rule cycle contains marked rules only. And, neither can the edge label $l \in N$. In a rule cycle, $l$ would occur both in a head as well as in a tail which it does in $R$, but, again, the rule $l \twoheadrightarrow \textcircled{\texttt{1}} \twoheadrightarrow \lambda$ cannot take part in a rule cycle. Thus, by definition, u cannot participate in any of $\mathscr{G}$'s rule cycles.

The DAG $G \in L(\mathscr{G})$ in Figure 4a uses each rule $r \in R$ only once, just as both vertex label `1` and edge label $l$. We can take $k$ disjoint isomorphic copies of $G$ for any $k \in \mathbb{N}$ and connect them by swapping the copies of $e$, as shown in Figure 4c, by Theorem 2.11. The resulting DAG $G[e_{k-1} \bowtie e_k]^k$ is still accepted by $\mathscr{G}$ and connected. Moreover, it contains $k$ occurrences of label u, which proves the lemma. □

**Lemma 9.4** (Unbounded Label Occurrence)**.** *Let $\mathscr{G} = (N, \Sigma, R)$ be a minimal deterministic DAG grammar and* u $\in \Sigma \cup N$ *a label of a vertex or an edge. The number of occurrences of* u *in graphs $G \in L(\mathscr{G})$, or, for edge labels, in their corresponding derivation DAGs D, is unbounded, iff one of the two following conditions is fulfilled:*

*a) Label* u *occurs in some rule cycle of $\mathscr{G}$.*

*b) There exist both a rule cycle c in which an unmarked $q \in N$ occurs as well as a rule path $\Pi$ between this nonterminal q and the label* u*.*

*Proof of Theorem 4.6.* Suppose that a) is true. Then $L(\mathscr{G})$ is infinite, according to Theorem 2.9, which in turn means that there is no bound on the length of (possibly undirected) paths in graphs $G \in L(\mathscr{G})$. To obtain paths of unlimited length by the pigeonhole principle the repetition of rules is needed since $R$ is finite. We may do so by using the rule cycle $c_u$ comprising u. The derivation $\emptyset \Rightarrow_R G' \Rightarrow^*_{c_u} \Rightarrow_R G$ does not impose a bound on the number of label u occurrences on $G$s derived like that, which proves that a) implies unboundedness of u occurrences.

Suppose that b) is true. The cycle $c$ with the unmarked state $q$ generates not only the labels it comprises arbitrarily often, as showed in above paragraph, but also an unbounded number of the nonterminal $q$ may appear when taking the intermediate graphs into account that are generated in the various steps to yield $G$. At every $q$ the derivation steps $\Rightarrow_\Pi$ generate a useful DAG with respect to $L(\mathscr{G})$ with a path comprising u. But $\mathscr{G}$ does not bound the generations of $q$, thus neither on the derivation step $\Rightarrow_\Pi$ and consequently also the number of the labels us which shows that b) implies that the number of occurrences of u is not bounded.

Turning now to the second direction we assume that no bound on the number of us exists for the DAG language $L(\mathscr{G})$. This means that $\mathscr{G}$ can repeatedly generate u. With a finite number of rules $R$, this is only possible by applying rules with label u an unbounded number of times. Repetition of rules is obtained either by a rule cycle, a) or by rule paths to a cycle b). In every iteration of the cycle, also the rule path $\Pi$ is repeated and like that our label u.

Obviously, rules that do not participate in any rule cycle, not having a path to a rule cycle cannot be used in a derivation multiple times, which proves the second direction. This result carries over without difficulty when regarding $u$ as the edge label $u \in N$ in a derivation DAG $D$, instead of the vertex label $u \in \Sigma$ of a DAG $\lfloor D \rfloor = G$, which completes the proof. □

# A GLR-like Parsing Algorithm for Three-Valued Interpretations of Boolean Grammars with Strong Negation[*]

Patrik Adrián
Faculty of Informatics, University of Debrecen,
Kassai út 26, 4028 Debrecen, Hungary
adrianpatrik@mailbox.unideb.hu

György Vaszil
Faculty of Informatics, University of Debrecen,
Kassai út 26, 4028 Debrecen, Hungary
vaszil.gyorgy@inf.unideb.hu

Boolean grammars generalize context-free rewriting by extending the possibilities when dealing with different rules for the same nonterminal symbol. By allowing not only disjunction (as in the case of usual context-free grammars), but also conjunction and negation as possible connections between different rules with the same left-hand side, they are able to simplify the description of context-free languages and characterize languages that are not context-free. The use of negation, however, leads to the possibility of introducing rules that interplay in such a way which is problematic to handle in the classical, two-valued logical setting. Here we define a three valued interpretation to deal with such contradictory grammars using a method introduced originally in the context of logic programming, and present an algorithm to determine the membership status of strings with respect to the resulting three valued languages.

Ever since their publication in 1956, context-free grammars (CFG) of Chomsky [2] have served as the ubiquitous tool for formal grammar specification, thanks to their easy-to-understand semantics and admission of simple parsing algorithms. Other formalisms, such as tree adjunct grammars [6], parsing expression grammars [4] and others have since been developed, partially to address the inadequacy of CFGs to fully describe natural languages, but none have been as successful as CFGs themselves.

Even though the original semantics of a CFG are defined in terms of a rewrite system over an alphabet of terminal and nonterminal symbols, the parsing problem may just as well be seen as a problem of logic, where grammar rules serve as rules of inference and parsing is a search for the proof of the root proposition. The Boolean grammars of Okhotin [9] build upon this interpretation and extend traditional CFGs with conjunction (intersection) and negation (complementation) operations. When interpreted over a two-valued, classical logic, a Boolean grammar may be contradictory and not have a satisfying solution. In a three-valued logic where indeterminacy is a truth value that is stable under negation, such contradictions become tractable.

Our work involves the development of a parser based on a generalized LR (GLR) method for the entailment semantics of a Boolean grammar. Okhotin's GLR-like parser [10] for Boolean grammars works on a two-valued foundation and is not general in the sense of a generalized parser, as it cannot handle certain classes of grammars. His algorithm realizes negation by the deletion of edges from the graph structured stack (GSS) used by the algorithm; our solution is more in the spirit of the original GLR (see [14]), which uses a monotone approach to parsing, where edges are only created, never removed.

We were motivated by the work of Kountouriotis et al. [7] that described a tabular parser for the well-founded semantics of a Boolean grammar. The well-founded semantics, originally introduced in [5] is a three-valued semantic interpretation of a logic program that builds on a restricted version of the closed world semantics and a closely associated rule of inference often referred to as "negation as failure". The (also three-valued) semantic model of [3] builds on what is very close to the open world semantics and infers knowledge based only on entailment, rather than failure to be proven true.

[*]Supported by the University of Debrecen Scientific Research Bridging Fund (DETKA).

The reader would rightfully expect the toy grammar with the singular rule $S \to \neg S$ to be self-contradictory and not have any two-valued models. In a three-valued setting, the language defined by such a grammar has an indeterminate relation to all strings of the underlying alphabet, i.e. it neither contains, nor excludes them. A more interesting case is the similar grammar with the rule $S \to S$, which, unlike the previous example, does have a two-valued model; in fact, every conceivable language models this grammar. The well-founded model of this grammar is the language that excludes all strings. One might, however, argue that the choice made here is rather arbitrary and is only a leftover from the two-valued world; the "correct" three-valued solution here is that one also cannot determine the containment status of words within this language; this time not because of inconsistency, but inadequacy. The Fitting-semantics of logic programs (and by extension, Boolean grammars) is based on the latter philosophy, and considers the containment status to be determinate if and only if it cannot be otherwise (i.e. it is entailed by the axioms, here implied by the rules of the grammar).

## 1 Preliminaries

We base our discussion on a highly restricted fragment of first order logic that, for the lack of function symbols, variables and quantifiers, we consider to be effectively propositional.

An atom is of the form $P(c)$, where $c$ is a constant symbol and $P$ is a unary predicate. A formula is either an atom or is built using the usual connectives $\neg$, $\wedge$ and $\vee$, in decreasing order of precedence. The set of all constant symbols is the Herbrand-universe ($\mathscr{U}$) and the set of atoms are the Herbrand-base ($\mathscr{B}$) of the language.

A rule is of the form $A \leftarrow \phi$ where $A$ is an atom and $\phi$ is a formula. The symbol $A$ is the head, $\phi$ is the body of the rule. A set of rules is well-formed if and only if (iff) no two rules have the same head and, for every atom $A'$ that appears anywhere within the body of a rule, a rule with head $A'$ exists, i.e. we require that each atom is defined exactly once. The reader may assume that we are only dealing with well-formed rulesets.

We consider the set of possible truth values $\mathbb{B} = \{\top, \bot, \backsim\}$ representing truth, falsity and a third judgement understood as being indeterminate. The usual Boolean operations are as in Figure 1, also known as Kleene's strong three-valued logic. A valuation is a function $\mathscr{B} \to \mathbb{B}$ that assigns a truth value to every atom in the Herbrand-base. We define the strict partial order $\prec$ over $\mathbb{B}$ as $\backsim \prec \bot$ and $\backsim \prec \top$, with $\bot$ and $\top$ unrelated. The relation $\preceq$ is the reflexive closure of $\prec$. A valuation $I_1$ is no more certain than $I_2$, written as $I_1 \preceq I_2$ iff for all atoms $A$ it holds that $I_1(A) \preceq I_2(A)$, i.e. $I_2$ changes at most the truth values of atoms that are indeterminate in $I_1$. For an arbitrary formula $\phi$, it holds that $I_1(\phi) \preceq I_2(\phi)$.

| $\neg$ | |
|---|---|
| $\top$ | $\bot$ |
| $\backsim$ | $\backsim$ |
| $\bot$ | $\top$ |

(a) Negation.

| $\wedge$ | $\top$ | $\backsim$ | $\bot$ |
|---|---|---|---|
| $\top$ | $\top$ | $\backsim$ | $\bot$ |
| $\backsim$ | $\backsim$ | $\backsim$ | $\bot$ |
| $\bot$ | $\bot$ | $\bot$ | $\bot$ |

(b) Conjunction.

| $\vee$ | $\top$ | $\backsim$ | $\bot$ |
|---|---|---|---|
| $\top$ | $\top$ | $\top$ | $\top$ |
| $\backsim$ | $\top$ | $\backsim$ | $\backsim$ |
| $\bot$ | $\top$ | $\backsim$ | $\bot$ |

(c) Disjunction.

Figure 1: Kleene's strong three-valued connectives.

A rule $A \leftarrow \phi$ is satisfied by the valuation $I$ iff $I(A) = I(\phi)$, i.e. the truth value of its left-hand side is the same as the value of the formula on its right-hand side, when evaluated over $I$. A set of rules $\Pi$ is satisfied by $I$ iff all rules in $\Pi$ are satisfied by $I$. A set of ground rules $\Pi$ may be written as a (potentially countably infinite) vector equation $\mathbf{A} \equiv \phi$ where $\mathbf{A}$ contains atoms and $\phi$ contains formulas. A valuation $I$ is the solution of this vector equation iff $I(\mathbf{A}) = I(\phi)$, where the elements are evaluated memberwise.

We will now describe deduction based on the semantics defined by Fitting [3].

Given a set of rules $\Pi$, the operator $\phi$ maps an arbitrary valuation $I$ to its $\phi$-successor $\phi(I)$ such that

$$\phi(I)(\mathbf{A}) = I(\phi).$$

Let $I_\backsim$ be the null valuation such that $I_\backsim(A) = \backsim$ for all atoms $A$. Let $n$ be a finite ordinal and $\omega_0$ be the first infinite ordinal. We define

$$\begin{aligned} I_0 &= I_\backsim \\ I_n &= \phi(I_{n-1}) \\ I_{\omega_0} &= \sup_{n<\omega_0} I_n \end{aligned}$$

where the supremum is taken over $\preceq$ and is equal to $\bigcup_{n<\omega_0} I_n$ where union is understood as $(I_1 \cup I_2)(A) = \max_\preceq\{I_1(A), I_2(A)\}$. The sequence $I$ is monotone in $\preceq$ and has a supremum $\omega$ that we call the entailment model of $\Pi$.

An important property of $I$ is that it is monotone with regards to $\preceq$, i.e. it never "retracts" any conclusion already made. Since $\omega = \bigcup_{n<\omega_0} I_n$, any atom that has an assigned truth value in $\omega$ must have one in $I_n$ for some finite $n$.

In the original setting of logic programming, where arbitrary terms of first-order logic may be formed, determinacy is only semidecidable, though our formulas will be constructed such that it is fully decidable. This is due to the dependency set (the transitive closure of the set of atoms occurring in $\phi$), for any rule $A \leftarrow \phi$, being finite for every atom, therefore an evaluation procedure requiring only a finite number of evaluations to determine the status of any $A$.

## 2 Three-valued languages and semantics

Given an alphabet $\Sigma$, $\Sigma^\ell$ is the set of all strings (words) of length $\ell$ and $\Sigma^*$ is $\bigcup_{i\geq 0}\Sigma^i$. A (classical) language over $\Sigma$ is a (possibly improper) subset of $\Sigma^*$.

An $n$-partition of a word $w$ is the tuple of words $w_1, \ldots, w_n$ such that $w = w_1 \cdots w_n$, where $w_1 \cdots w_n$ is the concatenation of $w_1, \ldots, w_n$. Similarly, for natural numbers, an $n$-partition of a natural number $\ell$ is an element of $\mathbb{N}^n$ whose members add up to $\ell$. We will take advantage of the natural isomorphism between the partitions of a natural number $\ell$ and those of a word $w$ with $|w| = \ell$.

The concatenation of languages $L_1, \ldots, L_n$, denoted as $L_1 \cdots L_n$ is the language of words $w$ such that there exists a partition $w = w_1 \cdots w_n$ such that $w_i \in L_i$, for all $1 \leq i \leq n$.

A three-valued language is a pair of languages $L = \langle L^\top, L^\perp \rangle$ over an alphabet $\Sigma$ such that $L^\top \cap L^\perp$ is empty. Notice that it is *not* required that $L^\top \cup L^\perp = \Sigma^*$. We define the following operations on three-valued languages (we use $L|^{\ell}$ to denote the set of words in $L$ that are exactly of length $\ell$):

$$\begin{aligned}
\overline{L} &= \langle L^{\perp}, L^{\top} \rangle \\
L_1 \cup \cdots \cup L_n &= \langle L_1^{\top} \cup \cdots \cup L_n^{\top}, L_1^{\perp} \cap \cdots \cap L_n^{\perp} \rangle \\
L_1 \cap \cdots \cap L_n &= \langle L_1^{\top} \cap \cdots \cap L_n^{\top}, L_1^{\perp} \cup \cdots \cup L_n^{\perp} \rangle \\
L_1 \cdots L_n &= \Bigg\langle \bigcup_{(l \geq 0)} \bigcup_{(p_1 + \cdots + p_n = l)} \bigcap_{(i \leq n)} \Sigma^{p_1} \cdots \Sigma^{p_{i-1}} (L_i^{\top}|^{p_i}) \Sigma^{p_{i+1}} \cdots \Sigma^{p_n}, \\
&\qquad \bigcup_{(l \geq 0)} \bigcap_{(p_1 + \cdots + p_n = l)} \bigcup_{(i \leq n)} \Sigma^{p_1} \cdots \Sigma^{p_{i-1}} (L_i^{\perp}|^{p_i}) \Sigma^{p_{i+1}} \cdots \Sigma^{p_n} \Bigg\rangle
\end{aligned}$$

These definitions agree with those in [7], in particular

- a word is an element of $(L_1 \cdots L_n)^{\top}$ iff it has an $n$-partition such that for all $1 \leq i \leq n$ the $i$th part belongs to $L_i$ and
- a word is an element of $(L_1 \cdots L_n)^{\perp}$ iff in every $n$-partition there exists an $1 \leq i \leq n$ such that the $i$th part is excluded from $L_i$.

The characteristic function of a three-valued language $L = \langle L^{\top}, L^{\perp} \rangle$ is the function $L : \Sigma^* \to \mathbb{B}$ such that

$$L(w) = \begin{cases} \top & \text{if } w \in L^{\top}, \\ \perp & \text{if } w \in L^{\perp}, \\ \backsim & \text{otherwise.} \end{cases}$$

We will write $w \in L$ for $w \in L^{\top}$ and $w \notin L$ for $w \in L^{\perp}$; note that containment is not dichotomous. The characteristic functions of the above are

$$\begin{aligned}
\overline{L}(w) &= \neg L(w) \\
(L_1 \cup \cdots \cup L_n)(w) &= \bigvee_{i \leq n} L_i(w) \\
(L_1 \cap \cdots \cap L_n)(w) &= \bigwedge_{i \leq n} L_i(w) \\
(L_1 \cdots L_n)(w) &= \bigvee_{w = w_1 \cdots w_n} \bigwedge_{i \leq n} L_i(w_i)
\end{aligned}$$

A three-valued language may either include, exclude any given string or the containment may be indeterminate. Indeterminacy may, informally, be understood as a sort of "weak exclusion" that is unsuitable for further deduction. The set of all three-valued languages over the alphabet $\Sigma$ will be denoted by $\mathscr{L}$.

## 2.1 Boolean grammars

A Boolean grammar is a triple $G = \langle \mathbf{V}, \Sigma, \mathbf{P} \rangle$ where $\mathbf{V}$ is the a of grammar variables (nonterminals), $\Sigma$ is an alphabet (terminals) and $\mathbf{P}$ is a set of grammar rules (productions). We will use $\Gamma = \mathbf{V} \cup \Sigma \cup \{\varepsilon\}$ to denote the complete set of grammar symbols ($\varepsilon \notin \mathbf{V} \cup \Sigma$).

We define grammar expressions and grammar rules as follows.

- Members of $\Gamma$ are grammar expressions.

- If $\phi$ is an expression, then $\neg\phi$ is a negated expression.
- If $\phi_1,\dots,\phi_n$ are expressions, then $\phi_1 \vee \cdots \vee \phi_n$ is a disjunctive expression.
- If $\phi_1,\dots,\phi_n$ are expressions, then $\phi_1 \wedge \cdots \wedge \phi_n$ is a conjunctive expression.
- If $\phi_1,\dots,\phi_n$ are expressions, then $\phi_1 \cdots \phi_n$ is a concatenation expression.
- If $\phi$ is an expression and $X \in \mathbf{V}$, then $X \to \phi$ is a grammar rule and $X$ is its head.

Rules are the top-level constructs of a Boolean grammar and are not expressions themselves. A Boolean grammar is well-formed if, for all $X \in \mathbf{V}$, there is exactly one rule whose head is $X$. Furthermore, we assume that $n > 1$ and that no direct subexpression of a grammar expression is of the same kind as its parent.

Given a set of grammar variables $\mathbf{V}$, an interpretation is a function $I : \mathbf{V} \to \mathscr{L}$. We may naturally extend it to arbitrary expressions as follows:

- $I(\varepsilon) = \langle \emptyset, \overline{\emptyset} \rangle$;
- $I(t) = \langle \{t\}, \overline{\{t\}} \rangle$ where $t \in \Sigma$;
- $I(\neg\phi) = \overline{I(\phi)}$;
- $I(\phi_1 \vee \cdots \vee \phi_n) = I(\phi_1) \cup \cdots \cup I(\phi_n)$;
- $I(\phi_1 \wedge \cdots \wedge \phi_n) = I(\phi_1) \cap \cdots \cap I(\phi_n)$;
- $I(\phi_1 \cdots \phi_n) = I(\phi_1) \cdots I(\phi_n)$.

All complements are understood with respect to a universe of $\Sigma^*$.

A grammar rule $X \to \phi$ is to be understood as an equation $I(X) = I(\phi)$. An interpretation $I$ is a model (a solution) of a grammar if and only if all grammar rules hold in $I$.

Somewhat similar in spirit to the naturally reachable semantics of Okhotin [9], the entailment semantics of Boolean grammars may be defined using an iterative approach. Let $\langle X_1,\dots,X_{|\mathbf{V}|} \rangle$ be a particular ordering of the grammar variables. We may then write an interpretation $I$ as a vector of languages $\langle I(X_1),\dots,I(X_{|\mathbf{V}|}) \rangle$. Starting from $I_0 = \langle \langle \emptyset,\emptyset \rangle,\dots,\langle \emptyset,\emptyset \rangle \rangle$ as the null interpretation, we may assign the next interpretation $I_{n+1}$ as $\langle I_n(\phi_1),\dots,I_n(\phi_{|\mathbf{V}|}) \rangle$, where $\phi_i$ is the definition of variable $X_i$, i.e. there is a rule $X_i \to \phi_i$ in the grammar. The sequence always converges (in at most a countably infinite number of steps) and provides a natural foundation of what we consider to be a natural three-valued semantics of a Boolean grammar. The convergence also holds if only one element is updated at a time, i.e. if $i$ is arbitrarily chosen between 1 and $|\mathbf{V}|$ (assuming that each value is eventually picked a sufficient number of times), then $I_{n+1} = \langle I_n(X_1),\dots,I_n(\phi_i)\dots,I_n(X_{|\mathbf{V}|}) \rangle$.

This a construction, while it serves as a natural semantic model for a Boolean grammar, is not particularly useful for parsing. The following approach ultimately defines the same model but does so for a single word at a time, using a particular construction of logic rules based on the characteristic functions. This is the theoretical foundation of how our parser makes inferences.

Let the (countably infinitely many) constants of our language of logic be the words of $\Sigma^*$ and the (unary) predicate symbols be members of $\Gamma$. We will construct a countably infinite set of logic rules, one for each word and grammar rule, that expresses their semantics.

We define a function $\rho(\phi, w)$ that takes an arbitrary grammar expression $\phi$ and a variable $w$ in the language of logic and maps it to an open (parametric) logic formula as follows:

- if $\phi \in \Gamma$, then $\rho(\phi, w)$ is $\phi(w)$;
- if $\phi$ is $\neg\psi$, then $\rho(\phi, w)$ is $\neg\rho(w)$;
- if $\phi$ is $\psi_1 \vee \cdots \vee \psi_n$, then $\rho(\phi, w)$ is $(\rho(\psi_1, w) \vee \cdots \vee \rho(\psi_1, w))$;
- if $\phi$ is $\psi_1 \wedge \cdots \wedge \psi_n$, then $\rho(\phi, w)$ is $(\rho(\psi_1, w) \wedge \cdots \wedge \rho(\psi_1, w))$;
- if $\phi$ is $\psi_1 \cdots \psi_n$, then $\rho(\phi, w)$ is $\bigvee_{w=w_1\cdots w_n} \bigwedge_{i \le n} \rho(\psi_n, w_n)$.

where the variables $w_1,\dots,w_n$ are new variables. The parametric forms of the rules are:

- for each grammar rule $X \to \phi$ we generate $X(w) \leftarrow \rho(\phi, w)$;

- for each terminal $t \in \Sigma \cup \{\varepsilon\}$ we generate $t(w) \leftarrow t = w$.

As a final step, the variables are substituted by the constants, i.e. the words over $\Sigma$. This results in a total of $|\mathbf{P}| + |\Sigma| + 1$ rules for each word in $\Sigma^*$.

These rules together define the intended meaning of a Boolean grammar. Note that even though the set of rules is infinite (as there are infinitely many words in $\Sigma^*$), the value of every atom is defined, both directly and indirectly, through others with a word length that is not greater than itself. In the worst case, the number of atoms that need to be evaluated to determine the valuation of a string is the number of distinct substrings times the number of symbols, i.e. $(1 + \frac{1}{2} \cdot |w| \cdot (|w| + 1)) \cdot |\Gamma|$, which is quadratic in the length of the string.

We shall illustrate the above with the example grammar taken from [7] whose language $S$ includes precisely the strings that are of the form $ww$ over an alphabet $\{a, b\}$:

$$\begin{aligned}
S &\to \neg(A \vee B \vee AB \vee BA) \\
A &\to CAC \vee a \\
B &\to CBC \vee b \\
C &\to a \vee b
\end{aligned}$$

The open (parametric) rules generated for the above grammar are:

$$\begin{aligned}
&\varepsilon(w) \leftarrow w = \varepsilon \\
&a(w) \leftarrow w = a \\
&b(w) \leftarrow w = b \\
&S(w) \leftarrow \neg \left( A(w) \vee B(w) \vee \bigvee_{w = w_1 w_2} [A(w_1) \wedge B(w_2)] \vee \bigvee_{w = w_1 w_2} [B(w_1) \wedge A(w_2)] \right) \\
&A(w) \leftarrow \bigvee_{w = w_1 w_2 w_3} [C(w_1) \wedge A(w_2) \wedge C(w_3)] \vee a(w) \\
&B(w) \leftarrow \bigvee_{w = w_1 w_2 w_3} [C(w_1) \wedge B(w_2) \wedge C(w_3)] \vee b(w) \\
&C(w) \leftarrow a(w) \vee b(w)
\end{aligned}$$

Finally, we substitute words into the parameter $w$. As the instantiated set is infinite, we will only demonstrate some rules using *abab* and its substrings.

$$\begin{aligned}
S(abab) \leftarrow \neg\Big(&A(abab) \vee B(abab) \vee \\
&\big(A(\varepsilon) \wedge B(abab) \vee A(a) \wedge B(bab) \vee A(ab) \wedge B(ab) \vee \\
&A(aba) \wedge B(b) \vee A(abab) \wedge B(\varepsilon)\big) \vee \\
&\big(B(\varepsilon) \wedge A(abab) \vee B(a) \wedge A(bab) \vee B(ab) \wedge A(ab) \vee \\
&B(aba) \wedge A(b) \vee B(abab) \wedge A(\varepsilon)\big)\Big) \\
&\vdots \\
A(\varepsilon) \leftarrow &\big(C(\varepsilon) \wedge A(\varepsilon) \wedge C(\varepsilon)\big) \vee a(\varepsilon) \\
A(a) \leftarrow &\big(C(a) \wedge A(\varepsilon) \wedge C(\varepsilon) \vee C(\varepsilon) \wedge A(a) \wedge C(\varepsilon) \vee C(\varepsilon) \wedge A(\varepsilon) \wedge C(a)\big) \vee a(a) \\
&\vdots \\
A(aba) \leftarrow &\big(C(\varepsilon) \wedge A(\varepsilon) \wedge C(aba) \vee C(\varepsilon) \wedge A(a) \wedge C(ba) \vee C(\varepsilon) \wedge A(ab) \wedge C(a) \vee \\
&C(\varepsilon) \wedge A(aba) \wedge C(\varepsilon) \vee C(a) \wedge A(\varepsilon) \wedge C(ba) \vee C(a) \wedge A(b) \wedge C(a) \vee \\
&C(a) \wedge A(ba) \wedge C(\varepsilon) \vee C(ab) \wedge A(\varepsilon) \wedge C(a) \vee C(ab) \wedge A(a) \wedge C(\varepsilon) \vee \\
&C(aba) \wedge A(\varepsilon) \wedge C(\varepsilon)\big) \vee a(aba) \\
&\vdots \\
C(a) \leftarrow &\, a(a) \vee b(a) \\
C(ab) \leftarrow &\, a(ab) \vee b(ab) \\
&\vdots \\
a(a) \leftarrow &\, a = a \\
a(b) \leftarrow &\, b = a \\
&\vdots
\end{aligned}$$

The interested reader may want to determine the status of the words *a*, *b* and *ab* in *S* of the grammar

$$\begin{aligned}
A &\to \varepsilon \vee A \\
S &\to Ab
\end{aligned}$$

over the alphabet $\{a,b\}$[1].

## 3 The Boolean GLR parser

First we give a very short review of the GLR algorithm and some of its modifications we build our variant upon.

The LR automaton is essentially a Rabin-Scott construction of a trivial nondeterministic pushdown automaton for a context-free grammar. Whenever a (context-free) rule $A \to \alpha \bullet B\beta$ is being read, with the dot signaling the current position up to which it has already been recognized, it is allowed to transition

[1]Excluded, included and indeterminate.

to any rule $B \to \gamma$ without the consumption of any input. Transitions not consuming any input are called $\varepsilon$-transitions and their closure forms the states of the LR automaton. The LR parser is a deterministic simulation of this automaton using a single stack, and has two main operations, *shift* and *reduce*, roughly equivalent to the stack operations *push* and *pop*. Shifting happens when the automaton reads input and pushes the new state on the stack; reduction consists of the removal of as many states as there are on the right-hand side of a rule and a new state, corresponding to having read the left-hand side of the rule, is pushed in their place. The automaton has a special state that signals the recognition of the start symbol and serves as a terminator.

Given that some context-free grammars are not deterministically recognizable using the LR algorithm, as the parsing actions are ambiguous (shift/reduce or reduce/reduce conflict), the first attempts to broaden the algorithm's applicability involved the use of lookaheads to assist the decision process. Though an improvement over the naïve design, lookaheads only generalize LR parsing to deterministic context-free grammars, which is a proper subset of all CFGs.

Viewing the stack as a linear directed acyclic graph (DAG), it is possible to efficiently simulate nondeterminism by generalization of the "graph-stack" into a nonlinear DAG, exploring all paths the LR automaton might take . The resulting structure is often termed a graph structured stack (GSS) and can be seen as a generalization of a stack, where every path ending at the root is a record of a possible stack of the LR automaton. Note that it is possible, but not necessary, to use lookaheads to disambiguate actions of the GLR algorithm.

The original GLR has a weakness in design when it comes to nullable rules (in a CFG, a rule is nullable iff every symbol on its right-hand side is nullable, i.e. derives the empty string), namely that edges corresponding to nulled deductions are still created in the GSS. This not only negatively affects the algorithm's efficiency, but also raises problems of correctness on a general CFG when certain rules with nullable right-ends are concerned.

One solution to the problem is the $\varepsilon$-GLR construction of Nederhof and Sarbo [8] that modifies the Rabin-Scott closure so that the closure of an item $A \to \alpha \bullet B\beta$ not only includes items of $B \to \bullet\gamma$, but – iff $B$ is nullable – also $A \to \alpha B \bullet \beta$. This modification prevents the creation of nulled edges in the graph at the cost of more complicated reductions, as now edges corresponding to nullable symbols might or might not be absent from the GSS. Another approach is the RNGLR of Scott and Johnstone [11], which performs reductions early when all symbols to the right of the dot are nullable.

These algorithms may not be cubic in the worst case, as the path scanning (determining the GSS nodes at which a reduction may end) may be of complexity $O(|w|^{n-1})$ for a path with $n$ components, for a total runtime of $O(|w|^{n+1})$. One may rewrite the grammar in Chomsky Normal Form to guarantee cubic runtime, which may, depending on the implementation and applied postprocessing of the parsing results, completely destroy its semantic structure. The BRNGLR of Scott and Johnstone [13] treats items $A \to \alpha \bullet B\beta$ as intermediary nonterminals and performs path reductions in steps of 2, guaranteeing an at worst cubic runtime.

## 3.1 The Boolean LR automaton

We build our solution for Boolean grammars on the foundations laid by the $\varepsilon$-GLR and the BRNGLR, namely

1. never create an edge in the GSS for nulled inputs and
2. never perform reductions of length greater than 2.

Nullability of symbols is a property of the grammar and not the input, therefore it is possible to precompute this knowledge, for example by explicit evaluation of the $\Phi$ operator on logic rules given at the end of the previous part for the empty string, repeated until the interpretation has converged (i.e. does not change between successive evaluations; this must happen in at most $\mathbf{V}$ steps). Notice that $\varepsilon$ is always positively nullable, terminals are never so.

Given the generalized structure of a Boolean grammar in the sense that we allow arbitrary formulas on the right-hand side, the items that form a state of the automaton will be labeled with arbitrary expressions that appear on the right-hand sides of grammar rules. As in our three-valued setting the lack of a proof for truth is insufficient to derive falsity (which is different from not-truth), we also augment items with a *sign* that signals whether derivations of the item should result in a positive or negative proof of the formula. An item is, therefore a triple consisting of a sign (either $+$ or $-$) indicating whether a positive or negative proof is expected; a grammar formula $\phi$ and position of the "dot", an index that ranges from 0 to $n$ (inclusive) for $n$-ary concatenations and one of 0 or 1 for other items, signaling how much of a given expression has been recognized.

The successor of an item is the item with the same sign and formula, and a dot that is one position ahead. A completion item is one where the dot has the highest possible index; it does not have a successor.

Similarly to the context-free case, which only has positive concatenation and disjunction (in the form of nondeterminism induced by transitions on multiple possible rules), the states (i.e. sets of items) are the closure of some initial "seed" items over rules that will be given shortly, and may be computed using iterated saturation. The items originally present in the state are referred to as kernel items, while those added via the closure are the derived items.

For items with non-trivial grammar formulas further derived items must be present in the state, and we will say they are generated by the item(s) that caused forced their inclusion. The parents of an item are the non-concatenation items that generate it. Intuitively, these items represent the transitive closure of the grammar expressions that may be required for the proof the kernel items.

We now go over the various expression types to detail how their child items are generated:

- $\pm t$ where $t \in \Sigma \cup \{\varepsilon\}$: Terminal items serve as the trivial cases of the matching algorithm and generate no further items. An item $+\varepsilon$ is never matched against the input as it is required nullable (positively matches only the empty string and negatively matches everything else); $-\varepsilon$ matches any input segment that is not empty. The terminals match the respective single character in the input and negatively match everything else.
- $\pm X$ where $X \in \mathbf{V}$: In order to match a grammar variable, the expression on the right-hand side of its defining rule must be matched, therefore a grammar variable generates exactly one item, $\pm\phi$, where $\phi$ is the grammar expression on the right-hand side of the rule $X \to \phi$. Because of the well-formedness criterion on our definition of a Boolean grammar, there is exactly one such rule.
- $\pm\neg\phi$: A negated expression matches if and only if $\phi$ matches with the opposite sign, therefore negated items generate the expression without the negation but the opposite sign, i.e. $\mp\phi$.
- $\pm\bigvee_{i\leq n}\phi_i$ and $\pm\bigwedge_{i\leq n}\phi_i$: For these items to match, some or all of their formulas need to be satisfied over some string. While the reduction phase (how the results are aggregated) is different for these items, for the purpose of building the automaton, they are handled equivalently and generate all their subformulas without a change of sign.

Concatenations also generate items within the state as part of the closure, but the concatenation is not considered as a parent of the generated item. This is because the reducer handles concatenations differently from other kinds of formulae.

- $+\phi_1\cdots\phi_n$: This is the classical case of concatenation whose subformulas must be matched sequentially. In the spirit of the $\varepsilon$-GLR described in [8], whenever the language defined by the dotted subformula includes the empty string, the successor item, i.e. the item with the dot at the successive index, is also included in the closure.

- $-\phi_1 \cdots \phi_n$: A negative concatenation is proven over some string if and only if we are able to ascertain that in every possible partition of the string there is a substring that is excluded by the respective language. (We note, without proof, that negative concatenation is also associative.) Suppose that the state contains an item $-\phi_1 \cdots \phi_n$ with the dot before some subformula $\phi_r$. To obtain a negative proof, either a negative proof of $\phi_r$ must be obtained, or the proof of $\phi_r$ may be skipped entirely and only the remainder of the rule (that is $\phi_{r+1} \cdots \phi_n$) be matched. Given that we need to consider skipping input segments of zero length, we also unconditionally add the successor item to the current state under the assumption that it follows a zero-length skip.

The transitions from a state $s$ of the Boolean LR automaton are implied by the items of the state. For any item $\pm \bullet \phi$ that is not a concatenation, the automaton has a transition on $\pm\phi$ to a state with $\pm\phi \bullet$. For concatenation items, if the item is labeled $\pm\phi_1 \cdots \bullet \phi_r \cdots \phi_n$, then there is a transition $\pm\phi_r$ to a state with the item $\pm\phi_1 \cdots \phi_r \bullet \cdots \phi_n$. If the concatenation is negative, the transition is marked as optional: the transition may be taken over an arbitrary nonempty string in the input. Whenever this happens, the resulting edge in the GSS is marked with the special symbol $*$ and not the formula.

The rationale behind the optional ("don't care") transitions is that in a negative concatenation it is always enough to negatively prove one subformula for a partition; all others may be mapped to arbitrary input segments. The marker $*$ will be referred to as a wildcard and a match marked with $*$ a wildcard match.

The pseudocode for building the automaton is presented in **Algorithm 1**.

## 3.2 The Boolean GLR parser

We now turn our attention to the actual parser in **Algorithm 2**. A match in a given input stream is identified by its left and right extents, which are the positions where the match begins and ends. As for any given left extent $i$ a trivial negative match may have almost any right extent $j \geq i$, the scanner part of the algorithm, even though progressing through the input in a left-to-right manner, finds for the current position $j$ all possible left extents $i < j$ where a trivial match may have begun. A trivial match here is either a positive or a negative match on $\varepsilon$, a terminal symbol or a wildcard match.

Whenever an edge is created in the GSS, it signals the acquisition of new knowledge in the parsing process. It is necessary that further applications of this knowledge are investigated and the process is continued until no further derivations can be made. This is the job of the reducer.

Given that no reductions are performed over an interval of length zero, as these are precomputed, every reduction must involve at least one edge of the GSS. Whenever a new edge is created, the possible reductions starting with that edge are investigated. The set $\Delta_j$ at this point contains $\langle u, e \rangle$ pairs where $u$ is a vertex in the GSS and $e$ is an outgoing edge of $u$, pointing backwards, against the input direction. Such a pair is created exactly once for each edge in the GSS and serves as a work item for the reducer.

No constructs other than concatenation require more than one edge in the GSS to be traversed sequentially. As a first step, the reducer calls FINISH-REDUCTION with a formula $\pm\phi$ to sort out every reduction that is not a concatenation. The purpose of FINISH-REDUCTION is to take a recognized formula and apply it to its parents that are not concatenations. The parent formulas to be substituted into should be precomputed, but even searching for them is constant time in the length of the input.

A variable is considered matched whenever its definition is matched, therefore if one of the parents is a variable, an edge is immediately created that represents the match. Negation is similarly simple, upon matching $\pm\phi$ a transition on $\mp\neg\phi$ is recorded in the GSS. If the parent is a positive disjunction or a negative conjunction, matching the child immediately causes an edge to be created. These are collectively referred to as existential reductions.

**Algorithm 1** Construction of the Boolean LR automaton.

```
procedure BUILD-AUTOMATON($S$)
    let $s_0$ be a state with an empty kernel and items $+\bullet S$ and $-\bullet S$
    while there is an unprocessed state $s$ do
        CLOSURE($s$)
end procedure
procedure CLOSURE($s$)
    while there is an unprocessed item $\iota$ in $s$ that is not a completion item do
        if $\iota$ is labeled $-\varepsilon$ or $\pm c$ where $c \in \Sigma$ then
            add the completion of $\iota$ to the transition on the label of $\iota$
        else if $\iota$ is labeled $\pm A$ where $A \in \mathbf{V}$ then
            create a new item $\iota'$ in $s$ labeled $\pm\phi$ where there is a rule $A \to \phi$ in the grammar
            add the completion of $\iota$ to the transition on $\pm A$
        else if $\iota$ is labeled $\pm\neg\phi$ then
            add a new item $\iota'$ labeled $\mp\phi$ in $s$
            add the completion of $\iota$ to the transition on $\pm\neg\phi$
        else if $\iota$ is labeled $\pm\phi_1 \vee \cdots \vee \phi_n$ or $\pm\phi_1 \wedge \cdots \wedge \phi_n$ then
            for $\phi$ in $\phi_1, \ldots, \phi_n$ do
                add a new item $\iota'$ labeled $\pm\phi$ in $s$
            add the completion of $\iota$ to the transition on the label of $\iota$
        else if $\iota$ is labeled $+\phi_1 \cdots \bullet \phi_r \cdots \phi_n$ then
            add a new item labeled $+\phi_r$ to $s$
            add the successor of $\iota$ to the transition on $+\phi_r$
            if $\phi_r$ is nullable and ($r < n$ or $\exists$ a right-nullable kernel item $+\phi_1 \cdots \phi_n$ in $s$) then
                add the successor of $\iota$ to $s$
        else if $\iota$ is labeled $-\phi_1 \cdots \bullet \phi_r \cdots \phi_n$ then
            add a new item labeled $-\phi_r$ to $s$
            add successor of $\iota$ to the transition on $-\phi_r$
            mark the transition on $-\phi_r$ as optional
            if $r < n$ or there is a kernel item $-\phi_1 \cdots \phi_n$ in $s$ then
                add the successor of $\iota$ to $s$
end procedure
```

Suppose that the parent is either a negative disjunction or a positive conjunction. These items require that all children are matched before the parent edge is created. (Note that child matches may end at different nodes in the GSS; this is no problem as long as these nodes belong to the same generation, and therefore cover the same part of the input.) Therefore, when processing these reductions, we only record in $u$ that one of the subformulas was matched, and only create the parent edge when records for all subformulas are present. These records are invalidated whenever the parser position advances, as matches on a segment $(i, j)$ are not meaningful for any other $(i, j')$.

So far we have discussed how reductions for the non-concatenation items are performed. We will now switch our attention to concatenations, as path tracing is not handled by the FINISH-REDUCTION function. For any formula other than negation, an edge will only satisfy an item of the same sign as the edge's label. Whenever an edge $e$ from $u$ to $v$ is created, where $u$ is a node in the current generation, only a reduction via the respective sign needs to be considered.

As the edge $e$ represents a transition of the underlying automaton, if $e$ is labeled $+\phi_r$, there must be some item $+\cdots\phi_r\bullet\cdots$ in the kernel of $u$. The edge $e$ may be the last edge of such a reduction only if the rule is right-nullable, i.e. all of $\phi_{r+1}\cdots\phi_n$ are positively nullable. In this case we shall traverse the edge and call the function EXTEND-POSITIVE-REDUCTION, which, if the concatenation is fully reduced (i.e. $r = 1$) allows parent items to progress by invoking FINISH-REDUCTION, otherwise it merely queues the rest of the rule for further progressing. Notice that even though we use the same set $\Delta_j$ for the queue as CREATE-EDGE, this causes no confusion, as these items are of the shape $\langle v, \pm\phi_1\cdots\bullet\phi_r\cdots\phi_n\rangle$, i.e. they do not name the specific edge the reduction should be continued on.

The function CONTINUE-POSITIVE-REDUCTION takes a partially completed reduction that already has had at least one edge matched and traces the path further either by matching edges in the GSS or by eliminating them if they're positively nullable.

The last piece of the puzzle is the negative deduction of a concatenation. In order to prove for some segment $(i, j)$ that a concatenation does not hold, one must prove that in every partition of that segment there is at least one part that negatively matches. The problem with the naïve approach of simply enumerating all partitions is that there is $O(|w|^{n-1})$ many of them, where $n$ is the number of concatenated entities. That's way too many. Luckily, the binarization technique is also applicable for negative concatenation, as the operation, like the positive case, remains associative.

Suppose that $-\phi_1\cdots\phi_n$ is to be proven over some segment $(i, j)$. It is clear that whatever partition one chooses, it has a position, call it $k$ that splits the interval into subintervals $(i,k)$ and $(k, j)$ such that either $-\phi_1$ matches over $(i,k)$ or $-\phi_2\cdots\phi_n$ does over $(k, j)$. If one is able to prove $-\phi_2\cdots\phi_n$ over $(k, j)$, then for any choice of $i \le k$ $-\phi_1\cdots\phi_n$ holds over $(i, j)$ for that specific $k$. If the proof of $-\phi_2\cdots\phi_n$ over $(k, j)$ was unsuccessful, then we are limited to choices of $i$ where $(i, j)$ matches $-\phi_1$. We call these suffix and prefix proofs, respectively, of the partitioning point $k$.

One has to do this for all $i \le k \le j$ to consider a concatenation negatively proven. Note that for a positive concatenation/negative disjunction the number of subproofs required to be reducibe depends on the number of subformulas (i.e. a proof of $+\phi_1 \wedge\cdots\wedge\phi_n$ requires proofs of $+\phi_1, \dots, +\phi_n$ each, for a negative concatenation the number of subproofs is dependent on the length of the string it is being proven over ($j - i + 1$ for a segment with extents $i$ and $j$).

We note that as the parser progresses in the input, suffix proofs get invalidated as the $j$ in $(k, j)$ changes, but prefix proofs may be considered permanent.

**Algorithm 2** The Boolean GLR parser

```
procedure PARSE($S$)
    BUILD-AUTOMATON($S$)
    create a node labeled $s_0$ in $U_0$
    if $\pm\varepsilon$ matches $S$ then
        yield the sign of $\varepsilon$ at position 0
    for $j$ in $[1..|w|]$ do
        SHIFTER
        REDUCER
end procedure
procedure SHIFTER
    for each node $u$ in generations $U_i$ where $i < j$ do
        for each terminal transition $t$ from $u$ do
            if $w[i, j]$ matches $t$ then
                CREATE-EDGE($u,t$)
        for each optional transition from $u$ do
            CREATE-EDGE($u,*$)
end procedure
procedure CREATE-EDGE($u,l$)
    let $v$ be the node in $U_j$ reached by the transition on $l$ from $u$
    add an edge $e$ labeled $l$ from $v$ to $u$
    add $\langle v,e\rangle$ to $\Delta_j$
end procedure
procedure REDUCER
    while there is a pending reduction $\langle u,e\rangle$ in $\Delta_j$ do
        FINISH-REDUCTION($e.target, e.label$)
        if $e$ is positive then
            for $\iota$ in $u.kernel$ labeled $+\phi_1 \cdots \phi_r \bullet \cdots \phi_n$ and $+\varepsilon$ matches $\phi_{r+1} \cdots \phi_n$ do
                EXTEND-POSITIVE-REDUCTION($e.target, +\phi_1 \cdots \bullet \phi_r \cdots \phi_n$)
            while there is an unprocessed continuation $\langle v, +\phi_1 \cdots \phi_r \bullet \cdots \phi_n\rangle$ in $\Delta_j$ do
                CONTINUE-POSITIVE-REDUCTION($v, +\phi_1 \cdots \phi_r \bullet \cdots \phi_n$)
        else
            for $\iota$ in $u.kernel$ labeled $-\phi_1 \cdots \phi_r \bullet \cdots \phi_n$ do
                if $e$ is labeled $-\phi_r$ or $e$ is labeled $*$ and $-\varepsilon$ matches $\phi_{r+1} \cdots \phi_n$ then
                    if $e$ is labeled $-\phi_r$ then
                        permanently mark position $j$ as complete for $-\phi_1 \cdots \bullet \phi_r \cdots \phi_n$ at $v$
                    EXTEND-NEGATIVE-REDUCTION($e.target, -\phi_1 \cdots \bullet \phi_r \cdots \phi_n, j$)
            while there is an unprocessed continuation $\langle v, -\phi_1 \cdots \phi_r \bullet \cdots \phi_n\rangle$ in $\Delta_j$ do
                CONTINUE-NEGATIVE-REDUCTION($v, -\phi_1 \cdots \phi_r \bullet \cdots \phi_n$)
end procedure
```

**Algorithm 2** The Boolean GLR parser (cont.)

```
procedure FINISH-REDUCTION($u, \phi$)
    let $\iota$ be the item in $u$ that is labeled $\bullet\phi$
    for $\iota'$ in $\iota.parents$ do
        if $\iota'$ is a variable, negation, positive disjunction or negative conjunction then
            CREATE-EDGE($u, \phi$)
            if $\phi = \pm S$ then
                yield the sign of $\phi$ at position $j$
        else
            mark the subformla $\phi$ as complete for $\iota'$ in $u$
            if all subformulas of $\iota'$ are complete in $u$ then
                CREATE-EDGE($u, \phi$)
end procedure
procedure CONTINUE-POSITIVE-REDUCTION($v, +\phi_1 \cdots \phi_r \bullet \cdots \phi_n$)
    if there is an item $+\phi_1 \cdots \bullet \phi_r \cdots \phi_n$ in $v$ and $+\varepsilon$ matches $\phi_r$ then
        EXTEND-POSITIVE-REDUCTION($v, +\phi_1 \cdots \bullet \phi_r \cdots \phi_n$)
    if $+\phi_1 \cdots \phi_r \bullet \cdots \phi_n$ is a kernel item in $v$ then
        for $e$ in $v.edges$ do
            EXTEND-POSITIVE-REDUCTION($e.target, +\phi_1 \cdots \bullet \phi_r \cdots \phi_n$)
end procedure
procedure CONTINUE-NEGATIVE-REDUCTION($v, -\phi_1 \cdots \phi_r \bullet \cdots \phi_n$)
    let $i \leftarrow$ the position of $v$
    if $v$ contains the item $-\phi_1 \cdots \bullet \phi_r \cdots \phi_n$ then
        EXTEND-NEGATIVE-REDUCTION($v, -\phi_1 \cdots \bullet \phi_r \cdots \phi_n, i$)
    if $v$ contains the kernel item $-\phi_1 \cdots \phi_r \bullet \cdots \phi_n$ then
        for $e$ in $v.edges$ where $e$ is labeled $-\phi_r$ or $e$ is labeled $*$ do
            EXTEND-NEGATIVE-REDUCTION($e.target, -\phi_1 \cdots \bullet \phi_r \cdots \phi_n, i$)
end procedure
procedure EXTEND-POSITIVE-REDUCTION($v, +\phi_1 \cdots \bullet \phi_r \cdots \phi_n$)
    if $r = 1$ then
        FINISH-REDUCTION($v, +\phi_1 \cdots \phi_n$)
    else
        add $\langle v, \pm\phi_1 \cdots \bullet \phi_r \cdots \phi_n \rangle$ to $\Delta_j$
end procedure
procedure EXTEND-NEGATIVE-REDUCTION($v, -\phi_1 \cdots \bullet \phi_r \cdots \phi_n, p$)
    mark position $p$ as complete for $-\phi_1 \cdots \bullet \phi_r \cdots \phi_n$ at $v$
    if $r = n$ or $-\phi_1 \cdots \bullet \phi_r \cdots \phi_n$ just got completed at $v$ then
        if $r = 1$ then
            FINISH-REDUCTION($v, -\phi_1 \cdots \phi_n$)
        else
            add $\langle v, -\phi_1 \cdots \bullet \phi_r \cdots \phi_n \rangle$ to $\Delta_j$
end procedure
```

## 3.3 Notes

### 3.3.1 Generating parse trees

Our algorithm does not deal with the construction of parse trees. As the aptly titled paper [12] states, "[r]ecognition is not parsing", and we do indeed refer to our algorithm as a parser, rather than a recognizer, whereas it is, in a strict sense, the latter. Our excuse for doing so is that through the application of the usual techniques, it should not pose a significant technical challenge to turn the algorithm into an actual parser; the algorithm is structured and the GSS is constructed in a way that contains all the information that would be included in a parse tree. We therefore consider the implementation of parse trees a technicality that was omitted for brevity, but should not be hard to implement, should the reader desire to.

### 3.3.2 Optimization opportunities

We mention two possible opportunities for the optimization of the algorithm.

The first one involves the building of the Boolean LR automaton. Suppose that, for example, a state $s$ contains an item $+\phi_1 \cdots \bullet (\psi_1 \vee \psi_2) \cdots \phi_n$. Then, by closure, it also contains $+\bullet\, \psi_1 \vee \psi_2$ and then $+\bullet\, \psi_1$ and $+\bullet\, \psi_2$. Suppose $\psi_1$ is matched. The automaton currently has a transition on $+\psi_1$ to a state $p$ with the item $+\psi_1\bullet$, where it is reduced, trivially traced back to $s$, it is found that it has an existentially reducible parent $+\bullet\, \psi_1 \vee \psi_2$, which causes another transition on the formula $+\psi_1 \vee \psi_2$ to a state $q$ with items $+\psi_1 \vee \psi_2\bullet$ and $+\phi_1 \cdots (\psi_1 \vee \psi_2) \bullet \cdots \phi_n$.

Notice, however, that when $+\psi_1$ is matched, it is always the case that $+\psi_1 \vee \psi_2$ is matched, too, therefore it is possible to transition on both into at the same time. A drawback of this optimization is that the GSS would lose some of its structure, making the potential recovery of a parse tree harder.

The second optimization involves the tracing of paths in the GSS during the reduction phase. As the GSS is an append-only structure whose new edges are drawn only to the current generation, which is monotonously moving to the right, given any pending concatenation $\langle v, \pm\phi_1 \cdots \bullet\, \phi_r \cdots \phi_n \rangle$ the set of GSS nodes where the paths reading $\phi_{r-1}, \dots, \phi_1$ (in order of backwards traversal) may end does not change as the algorithm progresses. It would therefore be possible to build these sets progressively as part of CREATE-EDGE, in a manner similar to [1]. This space-time tradeoff drastically reduces the time spent searching for paths in the GSS at the cost of an extra $O(|w|^2)$ storage.

### 3.3.3 Complexity bounds

Any generation of the GSS may contain at most $Q$ nodes, where $Q$ is the number of states of the Boolean LR automaton. For a string of length $|w|$, the largest possible number of different edges is of order $O(|w|^2 \cdot Q \cdot F)$ where $F$ is the cardinality of the set of all possible labels of edges. Note that both $Q$ and $F$ are constant for any given grammar, therefore the size of the GSS is of order $O(|w|^2)$.

As for the time complexity, assume a given $j$. The SHIFTER runs in time $O(j \cdot Q \cdot F)$ in worst case, with CREATE-EDGE being $O(1)$. In the reducer, FINISH-REDUCTION is once again of $O(1)$ complexity (the iteration on the parent items is invariant with respect to the input length) that is called for at most $O(j \cdot Q \cdot F)$ times for each edge pointing away from generation $j$. The amount of work done on the concatenations are bounded by the size of $\Delta_j$, which are at most of size $O(j)$. For each pending reduction $\langle v, \pm\phi_1 \cdots \bullet\, \phi_r \cdots \phi_n \rangle$ the only operation that is not of constant time is the loop on the outgoing edges of $v$ in the CONTINUE methods, meaning there is $O(j^2)$ steps performed overall for each concatenation. Summing up all the above results in an $O(|w|^3)$ runtime, as $j$ runs over each position.

## 4 Summary

Boolean grammars are a straightforward generalization of context-free grammars that both allow the description of some languages that are not context-free, and simplify the description of others that are. The introduction of negation, however opens up the possibility of contradictory grammars that have no classical solution. An approach based on three-valued logic, where grammar rules are taken as a system of logic equations, always produces a model through the iteration of a simple substitutive process. The words found as being included or excluded from the language are exactly those entailed by the logic equations. Containment in the three-valued sense is always decidable within tame polynomial bounds for any given word of the alphabet and therefore serves as a suitable basis for a parser.

We provided a short overview of the logical foundations of the three-valued interpretation of Boolean grammars and gave an efficient algorithm from the GLR family of constructions that is able to determine the containment status of a string within cubic polynomial bounds.

## References

[1] John Aycock & R. Nigel Horspool (2002): *Practical Earley parsing*. The Computer Journal 45(6), pp. 620–630, doi:10.1093/comjnl/45.6.620. Publisher: OUP.

[2] Noam Chomsky (1956): *Three models for the description of language*. IEEE Transactions on Information Theory 2(3), pp. 113–124, doi:10.1109/tit.1956.1056813.

[3] Melvin Fitting (1985): *A Kripke-Kleene semantics for logic programs*. The Journal of Logic Programming 2(4), pp. 295–312, doi:10.1016/s0743-1066(85)80005-4.

[4] Bryan Ford (2004): *Parsing expression grammars: a recognition-based syntactic foundation*. In: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 111–122, doi:10.1145/982962.964011.

[5] Allen Van Gelder, Kenneth A. Ross & John S. Schlipf (1991): *The well-founded semantics for general logic programs*. Journal of the ACM 38(3), pp. 619–649, doi:10.1145/116825.116838.

[6] Aravind K. Joshi, Leon S. Levy & Masako Takahashi (1975): *Tree adjunct grammars*. Journal of Computer and System Sciences 10(1), pp. 136–163, doi:10.1016/S0022-0000(75)80019-5.

[7] Vassilis Kountouriotis, Christos Nomikos & Panos Rondogiannis (2009): *Well-founded semantics for Boolean grammars*. Information and Computation 207(9), pp. 945–967, doi:10.1016/j.ic.2009.05.002.

[8] Mark-Jan Nederhof & Janos J. Sarbo (1996): *Increasing the Applicability of LR Parsing*. In Harry Bunt & Masaru Tomita, editors: Recent Advances in Parsing Technology, Kluwer Academic Publishers, pp. 35–57. Available at `https://doi.org/10.1007/978-94-010-9733-8_3`.

[9] Alexander Okhotin (2004): *Boolean grammars*. Information and Computation 194(1), pp. 19–48, doi:10.1016/j.ic.2004.03.006.

[10] Alexander Okhotin (2006): *Generalized LR parsing algorithm for Boolean grammars*. International Journal of Foundations of Computer Science 17(03), pp. 629–664, doi:10.1142/s0129054106004029.

[11] Elizabeth Scott & Adrian Johnstone (2006): *Right Nulled GLR Parsers*. ACM Transactions on Programming Languages and Systems 28(4), pp. 577–618, doi:10.1145/1146809.1146810. Place: New York, NY, USA.

[12] Elizabeth Scott & Adrian Johnstone (2010): *Recognition is not parsing — SPPF-style parsing from cubic recognisers*. Science of Computer Programming 75(1), pp. 55–70, doi:10.1016/j.scico.2009.07.001. Special Issue on ETAPS 2006 and 2007 Workshops on Language Descriptions, Tools, and Applications (LDTA '06 and '07).

[13] Elizabeth Scott, Adrian Johnstone & Rob Economopoulos (2007): *BRNGLR: a cubic Tomita-style GLR parsing algorithm*. Acta Informatica 44(6), pp. 427–461, doi:10.1007/s00236-007-0054-z.

[14] Masaru Tomita (1985): *An Efficient Context-Free Parsing Algorithm for Natural Languages*. In: *Proceedings of the 9th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'85, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, p. 756–764.

# Determinism in Multi-Soliton Automata

Henning Bordihn
Institut für Informatik und Computational Science
Universität Potsdam
Potsdam, Germany
henning@cs.uni-potsdam.de

Helena Schulz
Fakultät für Elektrotechnik und Informatik
TU Berlin
Berlin, Germany
schulz-helena@gmx.de

Soliton automata are mathematical models of soliton switching in chemical molecules. Several concepts of determinism for soliton automata have been defined. The concept of strong determinism has been investigated for the case in which only a single soliton can be present in a molecule. In the present paper, several different concepts of determinism are explored for the multi-soliton case. It is shown that the degree of non-determinism is a connected measure of descriptional complexity for multi-soliton automata. A characterization of the class of strongly deterministic multi-soliton automata is presented. Finally, the concept of perfect determinism, forming a natural extension of strong determinism, is introduced and considered for multi-soliton automata.

## 1 Introduction

Soliton automata represent a model based on the switching behaviour of certain chemical molecules in which the bonds between (mainly carbon) atoms posses alternating weights. When some kind of disturbance is injected, it travels through the molecule like a wave (or likewise a particle). The disturbance is called *soliton* as it travels through the molecule "unhindered", without loss of energy and without interference. The bonds between the molecule's atoms are changed along the path the soliton takes. This results in a different molecule. Taking the so obtained molecules as states, one is led to a system which behaves like an automaton.

For a brief account of the history of solitons we refer to [7] and [8, pp.18–19]. An extensive list of references regarding soliton computations and soliton automata can be found in [1]. The notion of soliton automata is encountered in [3]. In that paper also the concepts of determinism and strong determinism of soliton automata are considered and have been further investigated in [4, 5]. Strong determinism requires that, for every possible start and target atom, a soliton can take at most one path leading through the molecule. The main simplification of soliton automata as considered in [3, 4, 5] is the assumption that only one single soliton can be present in a molecule at the same time. This restriction has been overcome in [1] (and the subsequent paper [6]), where multi-soliton automata have been taken into consideration in which more than one soliton can travel through a molecule simultaneously. Several different concepts of determinism for multi-soliton automata are defined in [9] and [2].

The present paper aims to continue this line of research. In the next section, the necessary notions related to soliton automata and the various concepts of determinism are given. We restrict ourselves to the case of multi-soliton automata since single-soliton automata as considered in [3] are special cases of multi-soliton automata. In addition to deterministic and strongly deterministic soliton automata, also the concepts of perfect determinism and the degree of non-determinism are defined. Perfect determinism describes a natural concept that is somewhat "in between" determinism and strong determinism for soliton automata. The degree of non-determinism is a measure of descriptional complexity quantifying the amount of non-determinism of soliton automata. Section 2 concludes with the proof showing that the degree of non-determinism is connected with respect to soliton automata, that is, for every positive integer $g$, there is a soliton automaton with degree of non-determinism $g$.

Section 3 extends the notions of determinism to graphs underlying soliton automata (namely the graphs representing the bonding structure of the molecules under consideration, called soliton graphs). Similarly to the results known for the single-soliton case [3], we give a characterization of soliton graphs always inducing strongly deterministic soliton automata. In [3] it is shown that a single-soliton automaton is strongly deterministic if and only if its underlying graph is a tree or a so-called chestnut (see Definition 21). For the multi-soliton case, we show that a soliton graph is strongly deterministic if and only if it is a tree. Moreover, we prove that there is a chestnut which is not even perfectly deterministic. The paper concludes with a few remarks on open research questions related to the results presented here.

## 2 Soliton Automata

First, we introduce some notation and review some basic notions. The sets of positive integers, of non-negative integers and of integers are denoted by $\mathbb{N}$, $\mathbb{N}_0$ and $\mathbb{Z}$, respectively. We use standard notation for sets. We write $|S|$ for the cardinality of a set $S$. When no confusion is likely, we omit set brackets for singleton sets.

An *alphabet* is a finite non-empty set the elements of which are called *symbols.* Let $\Sigma$ be an alphabet. The set of all (finite) words over $\Sigma$, including the empty word $\lambda$, is denoted by $\Sigma^*$; let $\Sigma^+ = \Sigma^* \setminus \{\lambda\}$. The length $\lg(w)$ of a word $w \in \Sigma^*$ is defined by

$$\lg(w) = \begin{cases} 0, & \text{if } w = \lambda, \\ 1 + \lg(v), & \text{if } w = av \text{ with } a \in \Sigma \text{ and } v \in \Sigma^*. \end{cases}$$

A *semi-automaton* is a construct $\mathscr{A} = (Q, \Sigma, \tau)$ where $Q$ is a non-empty set, $\Sigma$ is an alphabet and $\tau : Q \times \Sigma \to 2^Q$ is a mapping. The elements of $Q$ are called states; $\Sigma$ is the input alphabet of $\mathscr{A}$; $\tau$ is the transition function of $\mathscr{A}$. In this paper, we assume that $Q$ is finite and that, for all $q \in Q$ and all $a \in \Sigma$, $\tau(q,a) \neq \emptyset$. Moreover, we drop the prefix "semi-" as we do not consider any other kind of automata.

Let $\mathscr{A} = (Q, \Sigma, \tau)$ be an automaton. The transition function $\tau$ is extended to $2^Q \times \Sigma^*$ as follows: for $R \subseteq Q$ and $w \in \Sigma^*$, let

$$\tau(R,w) = \begin{cases} R, & \text{if } w = \lambda, \\ \tau\left(\bigcup_{q \in R} \tau(q,a), v\right), & \text{if } w = av \text{ with } a \in \Sigma \text{ and } v \in \Sigma^*. \end{cases}$$

For $w \in \Sigma^*$, let $\tau_w$ be the mapping defined by $\tau_w(R) = \tau(R,w)$ for all $R \subseteq Q$. Instead of $\tau_w(R)$ we often write $R\tau_w$.

The automaton $\mathscr{A}$ is said to be deterministic if $|\tau_a(q)| = 1$ for all $a \in \Sigma$ and all $q \in Q$. In that case $\tau_a$ is considered as a mapping of $Q$ into $Q$, that is as a transformation of $Q$ rather than of $2^Q$. Inputs $u$ and $v$ of $\mathscr{A}$ are said to be equivalent if and only if $\tau_u = \tau_v$.

A *graph* is a pair $G = (N, E)$ with $N$ the set of *nodes* and $E \subseteq N \times N$ the set of edges. We consider only finite undirected graphs. An edge connecting nodes $n$ and $n'$ is given both as $(n,n')$ and $(n',n)$. Therefore, we require that, for $n, n' \in N$, $(n,n') \in E$ if and only if $(n',n) \in E$ and that these represent the same edge. Thus, any two nodes can be connected by at most one edge. A *path* is a sequence of nodes $n_0, n_1, \ldots, n_k$ such that for $0 \leq i < k$ the pair $(n_i, n_{i+1}) \in E$.

A *weight function* for $G$ is a mapping $w : N \times N \to \mathbb{N}_0$ satisfying

$$w(n,n') = w(n',n) \begin{cases} = 0, & \text{if } (n,n') \notin E \\ > 0, & \text{if } (n,n') \in E. \end{cases}$$

A *weighted graph* is a triple $(N,E,w)$ such that $(N,E)$ is a graph and $w$ is a weight function.

For a node $n$, the set $V(n) = \{n' \mid (n,n') \in E\}$ is the *vicinity* of $n$. The *degree* of $n$ is $d(n) = |V(n)|$, and the *weight* of $n$ is $w(n) = \sum_{n' \in V(n)} w(n,n')$. A node $n$ is said to be *isolated* if $d(n) = 0$, *exterior* if $d(n) = 1$, and *interior* if $d(n) > 1$.

We now provide several definitions regarding soliton automata.

**Definition 1 ([3])** *A* soliton graph *is a weighted graph $G = (N,E,w)$ satisfying the following conditions:*

1. *$N$ is the finite, non-empty set of nodes.*
2. *$E \subseteq N \times N$ is the set of undirected edges, such that $(n,n') \in E$ if and only if $(n',n) \in E$.*
3. *Every node $n \in N$ has the following properties:*
   (a) *$(n,n) \notin E$.*
   (b) *$1 \leq d(n) \leq 3$.*
   (c) *$w(n) \in \{1,2\}$ if $n$ is exterior, and $w(n) = d(n) + 1$ if $n$ is interior.*
4. *Every component (maximal connected subgraph) of $G$ has at least one exterior node.*

A soliton graph is an abstraction of a polyacetylene molecule. Carbon atoms are represented as interior nodes, and connections to surrounding structures are represented as exterior nodes. When drawing soliton graphs, we use letters for interior nodes and numbers for exterior nodes. Just like in polyacetylene, only single and double bonds are allowed. In the graph, single bonds are represented as edges with a weight of 1, and double bonds are represented as edges with a weight of 2. We draw an edge of weight 1 as a simple line and an edge of weight 2 as two parallel lines. The conditions regarding weight and degree imply that the two edges at a node of degree 2 must have different weights, and that, of the three edges meeting at a node of degree 3, two must have weight 1 and one must have weight 2. An example of a soliton graph is depicted in Figure 1.

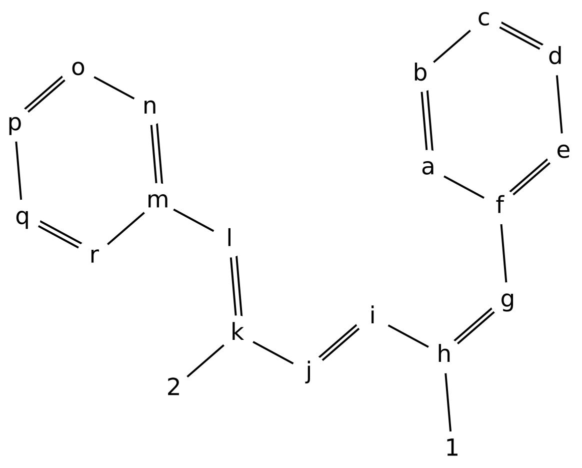


Figure 1: A soliton graph with two external nodes.

In [1] it has been reasoned about properties of the abstract model. The derived properties can be summarized as follows:

(I) One can insert and extract solitons at exterior nodes.

(II) Solitons move at a constant speed and have to move in every step. The speed is measured discretely as moving from one node to another one.

(III) Solitons move at the same speed, which is why solitons cannot overtake each other on the same path.

(IV) Solitons move over edges of alternating weights.

(V) When a soliton travels along an edge of weight $w$, the weight of the edge changes to $3-w$.

(VI) A soliton does not travel along the same edge twice in immediately consecutive steps.

(VII) Multiple solitons cannot travel along the same edge in the same step.

**Definition 2 (Bursts of Inputs [1])** *Let $S$ be a finite non-empty set not containing the symbols $\|$ and $\perp$. Moreover, let $S \cap \mathbb{N}_0 = \emptyset$.*

*A burst over $S$ is a word of the form*

$$s_1 \|_{k_1} s_2 \|_{k_2} \cdots s_{m-1} \|_{k_{m-1}} s_m \perp$$

*with the following properties:*

1. *$m \in \mathbb{N}$;*
2. *$s_1, s_2, \ldots, s_m \in S$;*
3. *$k_1, k_2, \ldots, k_{m-1} \in \mathbb{N}_0$;*

*The* length *of such a burst is $m$.*

*For $m \in \mathbb{N}$, let $\mathscr{B}_m(S)$ be the set of all bursts of length $m$ over $S$. Let*

$$\mathscr{B}_{\leq m}(S) = \bigcup_{i=1}^{m} \mathscr{B}_i(S) \text{ and } \mathscr{B}(S) = \bigcup_{i \geq 1} \mathscr{B}_i(S).$$

Let $G$ be a soliton graph, let $X$ be the set of its exterior nodes and $S = X \times X$. Then any set $B \subseteq \mathscr{B}(S)$ is called a set of bursts for $G$. The pair $s_i \in S$ contains the two nodes the $i$th soliton enters and leaves the graph through, respectively. A burst of the form $s_1 \|_{k_1} s_2 \|_{k_2} \cdots s_{m-1} \|_{k_{m-1}} s_m \perp$ is to be interpreted as follows. If the burst is initiated at time $t$, the symbol $s_1$ is input at time $t$; $s_2$ is input at time $t + k_1$; and, in general, $s_j$ is input at time $t + \sum_{i=1}^{j-1} k_i$. Here the empty sum is defined to be 0. The symbol $\perp$ indicates that the input process pauses until the system has stabilized.

**Definition 3 (Position Map [1])** *For $m \in \mathbb{N}$, let $\mathfrak{m} = \{1, 2, \ldots, m\}$. Further, let $G = (N, E, w)$ be a soliton graph such that $N \cap \mathbb{N}_0 = \emptyset$. A* position map *for $m$ is a mapping of $\mathfrak{m}$ into $N \cup \mathbb{N}_0$.*

If $\pi$ is a position map for $m$, then $\pi(i)$ indicates at which node the $i$th soliton is or how many steps are still required until it will enter the graph. Thus $\pi(i) = 1$ means that the ith soliton will enter the graph in the next step. $\pi(i) = n$ with $n \in N$ means that the soliton is at node $n$. $\pi(i) = 0$ means, by definition, that the $i$th soliton has left the graph.

**Definition 4 (Initial Position Map for a Burst [1])** *Let*

$$b = (n_1, n'_1)\|_{k_1}(n_2, n'_2)\|_{k_2} \cdots (n_m, n'_m)\perp$$

*be a burst of length m. The* initial position map $\pi_b$ for *b is defined as follows: Let r be minimal such that* $k_1 = k_2 = \cdots k_r = 0$ *and* $k_{r+1} > 0$ *or* $r = m-1$. *Then*

$$\pi_b(i) = \begin{cases} n_i, & \text{if } 1 \le i \le r+1, \\ k_{r+1}, & \text{if } i = r+2, \\ \pi_b(i-1) + k_{i-1}, & \text{if } i > r+2. \end{cases}$$

For example, let

$$b = (n_1, n'_1)\|_0(n_2, n'_2)\|_3(n_3, n'_3)\|_1(n_4, n'_4)\|_0(n_5, n'_5)\perp$$

be a burst. Then $\pi_b$ is given by the following table:

| Soliton $i$ | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Position $\pi_b(i)$ | $n_1$ | $n_2$ | 3 | 4 | 4 |

This means that the first two solitons start at node $n_1$ and $n_2$, respectively. The other solitons have to wait for 3 or 4 time steps.

**Definition 5 (Final Position Map [1])** *A position map $\pi$ for m is said to be* final *if* $\pi(i) = 0$ *for all* $i \in \mathfrak{m}$.

The processing of a burst starts with its initial position map and ends with a final position map corresponding in terms of the number of solitons. Small intermediate steps occur leading from the initial position map to the final position map. A burst is successful if and only if all its solitons have left the soliton graph after a finite amount of time.

**Definition 6 (Potential Successor Map [1])** *Let G be a soliton graph. Let* $m \in \mathbb{N}$, *and let* $\pi$ *and* $\pi'$ *be position maps for m. Let*

$$b = (n_1, n'_1)\|_{k_1}(n_2, n'_2)\|_{k_2} \cdots (n_m, n'_m)\perp$$

*be a burst of length m.*

*The map $\pi'$ is a* potential (direct) successor *of $\pi$ (with respect to b), if and only if*

$$\pi'(i) = \begin{cases} \pi(i) - 1, & \text{if } \pi(i) \in \mathbb{N}_0 \text{ and } \pi(i) > 1, \\ n_i, & \text{if } \pi(i) \in \mathbb{N}_0 \text{ and } \pi(i) = 1, \\ n, & \text{if } \pi(i) \in N,\ \pi(i) \neq n'_i,\ n \in N, \text{ and } \big(\pi(i), n\big) \in E, \\ 0, & \text{if } \pi(i) = n'_i \text{ or if } \pi(i) = 0. \end{cases}$$

*for* $i = 1, 2, \ldots, m$.

This ensures that the waiting times of the solitons are reduced in every step, solitons enter the graph at the right node, they have to use an edge in order to reach the next node, and that 0 is a value in the position map if the corresponding soliton reached the exterior node it is supposed to leave the graph through.

**Definition 7 (Configuration and Configuration Trail [1])** *Let* $G = (N, E, w)$ *be a soliton graph. Let* $m \in \mathbb{N}$, *and let*

$$b = (n_1, n'_1)\|_{k_1}(n_2, n'_2)\|_{k_2} \cdots (n_m, n'_m)\perp$$

*be a burst of length m.*

1. *A* configuration *(for $b$) is a pair $(G', \pi)$ such that $G' = (N, E, w')$ is a weighted graph with weights in $\{1, 2\}$ and $\pi$ is a position map for $m$.*
2. *A* configuration trail *for $G$ and $b$ is a finite sequence*

$$(G_0, \pi_0), (G_1, \pi_1), \ldots$$

*of configurations for $b$ with the following properties.*

   (a) *$G_0 = G$, and $\pi_0$ is the initial position map for $b$.*

   (b) *$\pi_1$ is a potential successor of $\pi_0$ such that $\pi_0(i) \in N$ implies $\pi_1(i) \in N$ for all $i \in \mathfrak{m}$.* *$G_1 = (N, E, w_1)$ is obtained from $G_0 = (N, E, w_0)$ by changing the weights of some edges as follows: If $\pi_0(i) \in N$, then*

$$w_1\big(\pi_0(i), \pi_1(i)\big) = w_1\big(\pi_1(i), \pi_0(i)\big) = 3 - w_0\big(\pi_0(i), \pi_1(i)\big).$$

   *For all other edges the weights remain unchanged.*

   (c) *Let $j > 1$. The sequence*

$$(G_0, \pi_0), (G_1, \pi_1), \ldots, (G_j, \pi_j)$$

   *is a configuration trail, if and only if*

$$(G_0, \pi_0), (G_1, \pi_1), \ldots, (G_{j-1}, \pi_{j-1})$$

   *is a configuration trail such that $\pi_{j-1}$ is not final, $G_j = (N, E, w_j)$, and the following conditions are satisfied (for all $i \in \mathfrak{m}$):*

      i. *$\pi_j$ is a potential successor of $\pi_{j-1}$.*
      ii. *If $\pi_{j-1}(i) \in N$ is exterior and $\pi_{j-2}(i) = 1$, then $\pi_j(i) \in N$.*
      iii. *If $\pi_{j-1}(i) \in N$ is exterior and equal to $n'_i$, and if $\pi_{j-2}(i) \in N$, then $\pi_j(i) = 0$.*
      iv. *If $\pi_{j-1}(i) \in N$ is interior and $\pi_{j-2}(i) \in N$, then*

$$w_{j-2}\big(\pi_{j-2}(i), \pi_{j-1}(i)\big) \neq w_{j-1}\big(\pi_{j-1}(i), \pi_j(i)\big).$$

      v. *If $\pi_j(i) \neq 0$, then $\pi_j(i) \neq \pi_{j-1}(i)$ and $\pi_j(i) \neq \pi_{j-2}(i)$.*
      vi. *$G_j$ is obtained from $G_{j-1}$ by changing the weights of some edges as follows: If $\big(\pi_{j-1}(i), \pi_j(i)\big) \in E$, then*

$$w_j\big(\pi_{j-1}(i), \pi_j(i)\big) = w_j\big(\pi_j(i), \pi_{j-1}(i)\big) = 3 - w_{j-1}\big(\pi_{j-1}(i), \pi_j(i)\big).$$

      *All other weights remain unchanged.*

3. *A configuration trail is* legal, *if it satisfies the following conditions for all $j \geq 1$:*

   (a) *If $\pi_{j-1}(i)$ and $\pi_{j-1}(i')$ are nodes and $\pi_{j-1}(i) = \pi_{j-1}(i')$ for some distinct $i$ and $i'$, then $\pi_j(i) \neq \pi_j(i')$.*

   (b) *If $\pi_{j-1}(i)$ and $\pi_{j-1}(i')$ are nodes with $\big(\pi_{j-1}(i), \pi_{j-1}(i')\big) \in E$, then $\pi_j(i) \neq \pi_{j-1}(i')$ or $\pi_j(i') \neq \pi_{j-1}(i)$.*

4. *A configuration trail*

$$(G_0, \pi_0), (G_1, \pi_1), \ldots, (G_j, \pi_j)$$

*is* partial *if $\pi_j$ is not final. Otherwise, it is* total.

A configuration defines the weights of the current graph and the positions of the solitons for a certain time step. Note that the graph in a configuration need not be a soliton graph. It represents the situation when all solitons have reached the "next" nodes on their ways. Consider, for example, the soliton graph in Figure 1. If a soliton entered the graph at node 1 and has reached node $h$, the weight of edge $(1,h)$ has changed to 2; thus $w(h) = 5$.

The conditions above ensure that all solitons behave exactly as defined in the rules concerning soliton movements. A consequence of the condition that no two solitons can traverse the same edge at the same time is that they also cannot enter the same exterior node at the same time. This holds true both for exterior nodes used as entry points and those used as exit points. Two solitons can be at an interior node simultaneously, but must leave it on different edges. Moreover, they cannot simply swap places.

**Definition 8 (Soliton Path)** *Let $G = (N,E,w)$ be a soliton graph. Let $m \in \mathbb{N}$, let*

$$b = (n_1,n'_1)\|_{k_1}(n_2,n'_2)\|_{k_2}\cdots(n_m,n'_m)\bot$$

*be a burst of length m, and let $C = (G_0,\pi_0),(G_1,\pi_1),\ldots,(G_j,\pi_j)$ be a configuration trail for G and b, $j \geq 0$. For every $i \in \mathfrak{m}$, let $\ell$ be the smallest and r be the largest number, $0 \leq \ell \leq r \leq j$ such that $\pi_\ell(i) \in N$ and $\pi_r(i) \in N$. The path*

$$\pi_\ell(i),\pi_{\ell+1}(i),\ldots,\pi_r(i)$$

*is the soliton path of soliton i in C. For $\ell \leq h < r$, the edge $(\pi_h(i),\pi_{h+1}(i))$ is said to be* used *by soliton i in C.*

**Definition 9 (Result of a Burst [1])** *Let G be a soliton graph and let b be a burst. The* result of burst *b* on *G is the set Result$(G,b)$ of weighted graphs $G'$ such that there is a total legal configuration trail for G and b transforming G into $G'$.*

Every element of Result$(G,b)$ is again a soliton graph.

Let $B \subseteq \mathscr{B}(X \times X)$ be a set of bursts. Let

$$\text{Result}(G,B) = \bigcup_{b \in B} \text{Result}(G,b).$$

For $i \in \mathbb{N}_0$, let

$$\text{Result}^i(G,B) = \begin{cases} G, & \text{if } i = 0, \text{ and} \\ \text{Result}(\text{Result}^{i-1}(G,B),B), & \text{if } i > 0 \end{cases}$$

and

$$\text{Result}^*(G,B) = \bigcup_{i \geq 0} \text{Result}^i(G,B).$$

We can use the resulting soliton graphs we obtain by traversing total legal configuration trails as states of an automaton. Such an automaton is induced by an underlying soliton graph and a set of bursts.

**Definition 10 (Multi-Soliton Automaton [1])** *Let G be a soliton graph with set X of exterior nodes. Let $B \subseteq \mathscr{B}(X \times X)$ be a set of bursts. Let*

$$States(G,B) = Result^*(G,B).$$

*The B-*soliton automaton *of G is the finite automaton $\mathscr{A}_B(G)$ with inputs $b \in B$, state set States$(G,B)$ and non-deterministic transition function*

$$\tau(G',b) = \begin{cases} Result(G',b), & \text{if } Result(G',b) \neq \emptyset, \\ \{G'\}, & \text{otherwise,} \end{cases}$$

*for $G' \in States(G,B)$ and $b \in B$.*

Note that States$(G,B)$ is bounded, as the set of vertices and the set of edges do not change, only the weights do. Therefore, there is a finite set $B$ of bursts such that States$(G,B) = $ States$(G,B')$ for all sets $B'$ of bursts with $B \subseteq B'$. If there is no risk of confusion, a $B$-soliton automaton will be called multi-soliton automaton or simply soliton automaton.

Now we define different kinds of determinism for soliton automata.

**Definition 11 (Determinism [2])** *Let $G = (N,E,w)$ be a soliton graph and let $B$ be a set of bursts for $G$. $\mathscr{A}_B(G)$ is called*

*(I)* deterministic*, if $|Result(G',b)| = 1$ for all $G' \in States(G,B)$ and all $b \in B$.*

*(II)* strongly deterministic*, if for all $G' \in States(G,B)$ and $b \in B$, there is at most one total legal configuration trail for $G'$ and $b$.*

A soliton automaton is deterministic if there is exactly one successor state for each state in the set States$(G,B)$ and each burst in $B$. Strong determinism is an even stronger constraint, as it also restraints in how many ways it is possible to transition from one state into another. An automaton has this kind of determinism if there is at most one total legal configuration trail for each state in States$(G,B)$ and each burst in $B$.

By definition, all total legal configuration trails are considered as transitions between the automaton's states. There are, however, cases in which an infinite number of configuration trails are possible for a state and a burst. For example, a soliton can get into a situation where it has the possibility to traverse a cycle infinitely many times. Since configuration trails for those kinds of situations contain "unnecessary" repetitions, we aim to classify configuration trails into two categories: perfect and imperfect. In order to determine when a configuration trail becomes imperfect, we search for equivalent configurations in it. Therefore, we first need to describe when two configurations are called equivalent.

**Definition 12 ([2])** *Let $G$ be a soliton graph. Let $C = (G_0,\pi_0),(G_1,\pi_1),...,(G_i,\pi_i)$ be a partial configuration trail that is not a total configuration trail. The set of possible successor position maps, denoted as $SC(C)$, is the set containing all $\pi_{i+1}$, such that $(G_0,\pi_0),(G_1,\pi_1),...,(G_i,\pi_i),(G_{i+1},\pi_{i+1})$ is a configuration trail.*

**Definition 13 (Successor-Equivalence [2])** *Let $(G_0,\pi_0),(G_1,\pi_1),...,(G_k,\pi_k)$ be a configuration trail. For integers $i$ and $j$ with $0 \le i \le k$ and $0 \le j \le k$, let $C = (G_0,\pi_0),(G_1,\pi_1),...,(G_i,\pi_i)$ and let $C' = (G_0,\pi_0),(G_1,\pi_1),...,(G_j,\pi_j)$. The configurations $(G_i,\pi_i)$ and $(G_j,\pi_j)$ are called successor-equivalent, if $(G_i,\pi_i) = (G_j,\pi_j)$ and $SC(C) = SC(C')$. This property is written as $(G_i,\pi_i) \equiv_{SC} (G_j,\pi_j)$.*

**Definition 14 ((Im)Perfect Configuration Trail [2])** *Let $G$ be a soliton graph and let $b$ be a burst. Let $C = (G_0,\pi_0),(G_1,\pi_1),...,(G_k,\pi_k)$ be a configuration trail with $G_0 = G$ and $\pi_0 = \pi_b$ (the initial position map for $b$). $C$ is called imperfect, if at least two configurations $(G_i,\pi_i)$ and $(G_j,\pi_j)$ exist, $0 \le i < j \le k$, where $(G_i,\pi_i) \equiv_{SC} (G_j,\pi_j)$. Otherwise, $C$ is called perfect.*

A configuration trail is perfect, if there are no two occurrences of successor-equivalent configurations in it. In the other case, we define it as imperfect, because then it would contain unnecessary steps. Consider the case of a configuration trail with two successor-equivalent configurations $(G_i,\pi_i)$ and $(G_j,\pi_j)$. We could make the exact same next moves after time step $i$ and $j$, so we might as well cut out all the configurations from $(G_{i+1},\pi_{i+1})$ until $(G_j,\pi_j)$. In [2] it is shown that considering only perfect configuration trails in the construction of a soliton automaton does not change its set of states. Also, if an imperfect configuration trail exists in a soliton automaton it can not be strongly deterministic.

Let $G$ be the soliton graph from configuration 1 in Figure 2 and let $B = \{(1,1)\perp\}$. In [3] it is shown that the $B$-soliton automaton $\mathscr{A}_B(G)$ is strongly deterministic. On the other hand, by using the set of bursts $B' = \{(1,1)\|_1(1,1)\perp\}$ the automaton $\mathscr{A}_{B'}(G)$ is not strongly deterministic. This is due to the white soliton having two possible successor positions in configuration 11. It could move to node $b$, like in configuration 12, or it could move to node $d$, resulting in both solitons staying inside the cycle. Eventually, this configuration trail would lead to a configuration successor-equivalent to configuration 11 and can therefore be classified as imperfect. However, for this graph and this burst we cannot find any perfect total legal configuration trails, except from the trail continued from configuration 12. In order to further discriminate such situations we introduce the following property.

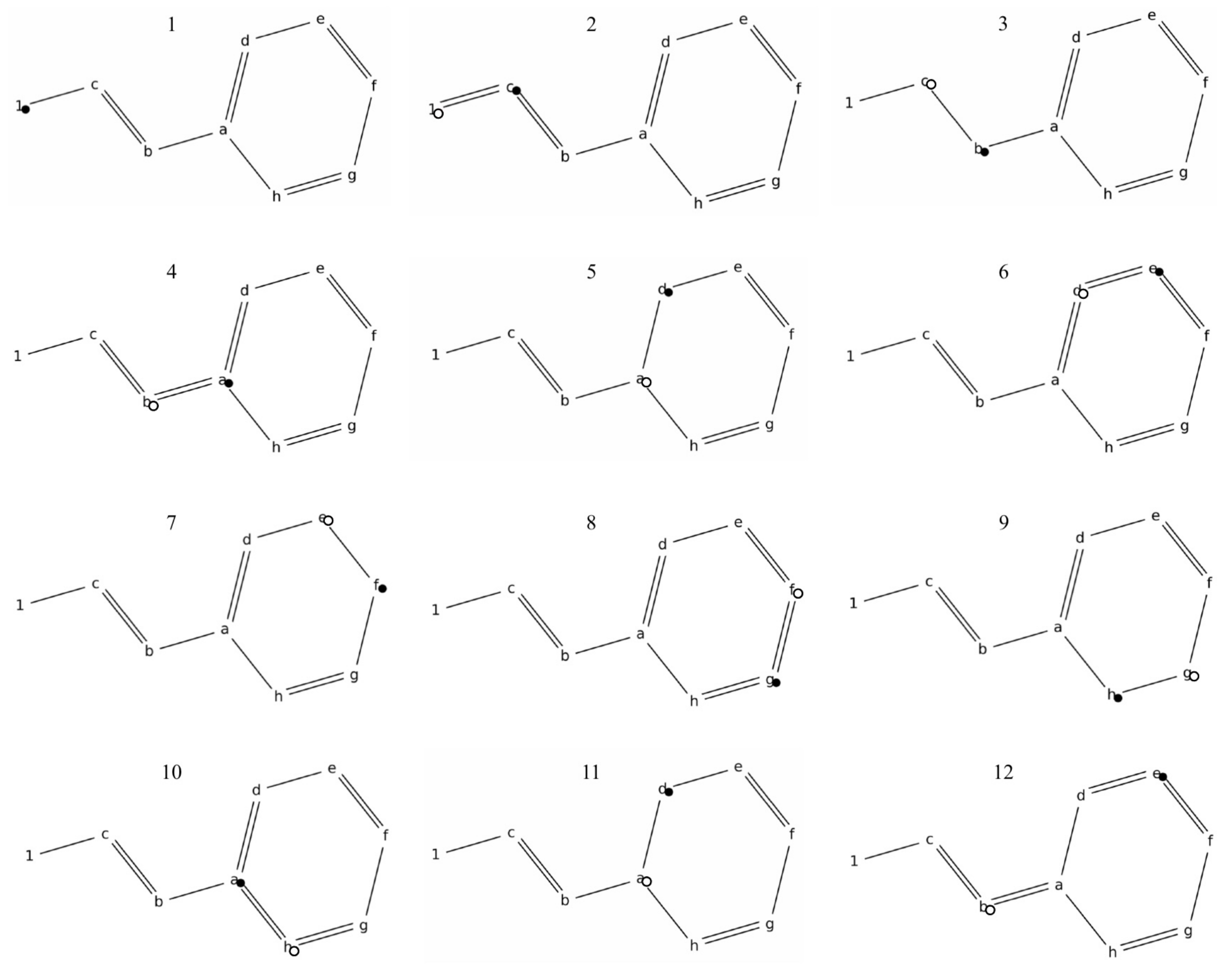


Figure 2: Part of a configuration trail for the burst $(1,1)\|_1(1,1)\perp$. The first soliton is depicted as a black pebble, while the second one is depicted as a white pebble.

**Definition 15 (Perfect Determinism)** *Let $G = (N,E,w)$ be a soliton graph and let $B$ be a set of bursts for $G$. $\mathscr{A}_B(G)$ is called* perfectly deterministic, *if for all $G' \in States(G,B)$ and $b \in B$, there is at most one perfect total legal configuration trail for $G'$ and $b$.*

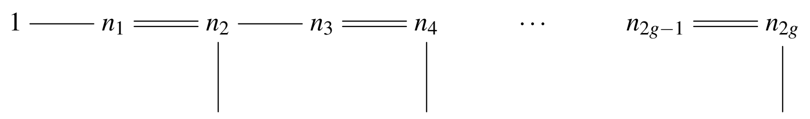
The automaton $\mathscr{A}_{B'}(G)$ from our example is perfectly deterministic, but not strongly deterministic. Hence, perfect determinism lies "in between" determinism and strong determinism.

Distinct soliton automata that are *not* deterministic may be different "distances" away from fulfilling the determinism property. We now formulate a measure of descriptional complexity that quantifies this distance.

**Definition 16 (Degree of Non-Determinism [2])** *Let $G = (N,E,w)$ be a soliton graph and let $B$ be a set of bursts for $G$. The* degree of non-determinism *of $\mathscr{A}_B(G)$ is the smallest integer $g \geq 1$, such that $|Result(G',b)| \leq g$ for all $G' \in States(G,B)$ and all $b \in B$.*

**Theorem 1** *The degree of non-determinism is a* connected *measure of descriptional complexity, that is, for every positive integer g, there is a soliton automaton $A_g$ such that its degree of non-determinism is g.*

*Proof.* For $g \geq 1$, let $G_g = (N_g, E_g, w_g)$ be the soliton graph with exactly two exterior nodes 1 and 2 and a path $1, n_1, n_2, \ldots, n_{2g-1}, n_{2g}$ with $w_g(1,n_1) = 1$ which we will call *basic chain* in the sequel. Moreover, additional edges leave the basic chain at every other node of the basic chain:



The edges leaving the basic chain all lead to the exterior node 2 and belong to a sub-graph forming a binary tree with $n_2, n_4, \ldots, n_{2g}$ as leaves and node 2 as its root, in which the root has weight 2 and branching edges always have weight 1. The inner nodes of that tree are denoted by $v_1, \ldots, v_r$ with $r = 2g - 3$ (if $g > 1$). There is an edge $(n_2, v_1)$ with weight 1 and, for $1 < k \leq g$, there is an edge $(n_{2k}, v_{2k-3})$ with weight 1. The first three soliton graphs $G_1, G_2, G_3$ are depicted in Figure 3.

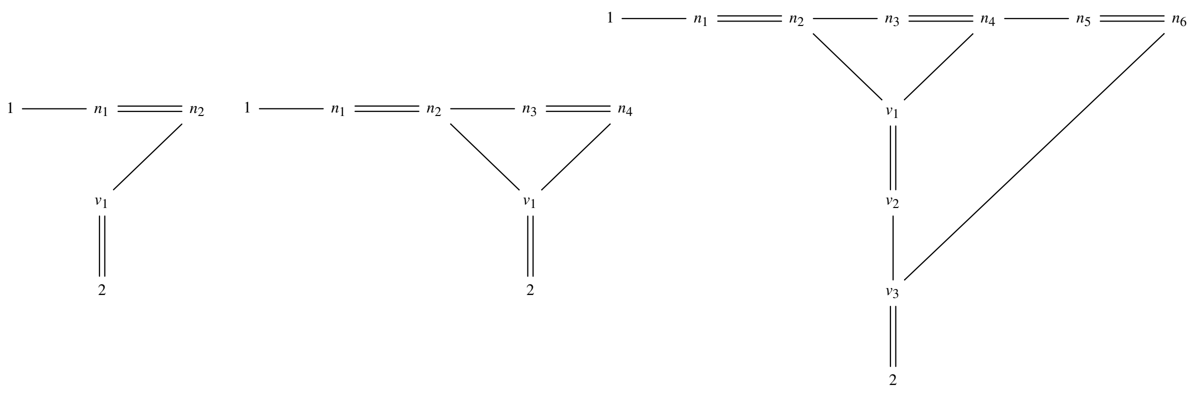
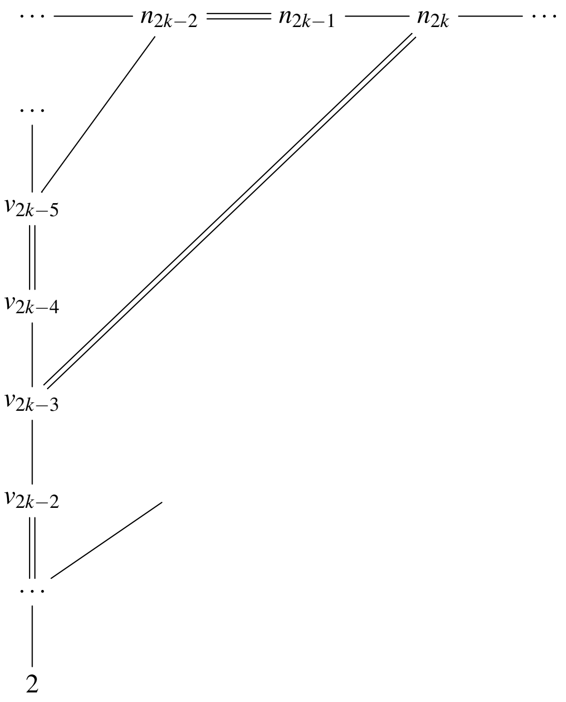


Figure 3: Soliton graphs $G_1$, $G_2$ and $G_3$

Further, let $B = \{(1,2)\perp\}$. Since the soliton has a non-deterministic choice only on every node $n_{2k}$ of the basic chain, $1 \leq k < g$, there are exactly $g$ soliton paths for the soliton in $B$. Notice that once the soliton has left the basic chain it has to follow the path leading directly to node 2 since it enters a node with degree 3 only when it has just traversed an edge with weight 1. Thus, $|\text{Result}(G_g, B)| \leq g$. The soliton uses exactly one of the edges $(n_2, v_1)$ or $(n_{2k}, v_{2k-3})$, $1 < k \leq g$, and the weight of the used edge has turned to 2 whereas the other edges leaving the basic chain keep their weight 1. Consequently, $|\text{Result}(G_g, B)| = g$.

Next, we prove $\text{Result}(G',B) = \{G_g\}$ for all $G' \in \text{Result}(G_g,B)$ and every $g \geq 1$. Assume the soliton has used edge $(n_{2k}, v_{2k-3})$ in $G_g$, for some $k$, $1 < k \leq g$ (the case $k = 1$, when it has used $(n_2, v_1)$, is similar). Then, the resulting soliton graph $G'$ is of the form



In $G'$, every soliton path for burst $(1,2)\perp$ has the prefix $1, n_1, n_2, \ldots, n_{2k}, v_{2k-3}$. Now, this path can be continued with $v_{2k-2}, v_{2k-1}, \ldots, 2$ leading directly to node 2, and the resulting soliton graph is $G_g$. Alternatively, the soliton can use edge $(v_{2k-3}, v_{2k-4})$ (or, if $k = 1$, edge $(v_1, n_4)$), or it can use $(v_{2k-3}, v_{2k-2})$ and later an edge with weight 1 leading back towards the basic chain (which has the same weights as in $G_g$ now). In any such case, some node $n_j$ of the basic chain will be reached via an edge with weight 1. Therefore, the soliton has to use the edge $(n_j, n_{j-1})$ next, leading "to the left" in the basic chain. Now, it cannot reach node 2, because every node $n_\ell$ in the basic chain having degree 3 is reached via an edge with weight 1 and has to be left to $n_{\ell-1}$ since $w(n_\ell, n_{\ell-1}) = 2$. Therefore, no further soliton graphs are added to $\text{Result}(G',B)$.

In conclusion, $|\text{Result}(G',B)| = 1$ for all $G' \in \text{Result}(G_g,B)$ and $g$ is the smallest integer with $|\text{Result}(G,B)| \leq g$ for all $G \in \text{States}(G_g,B)$. Hence, the degree of nondeterminism of $\mathscr{A}_B(G_g)$ is $g$, for every $g \geq 1$. $\square$

## 3 Graph Properties and Determinism

So far, we defined determinism properties only on soliton automata. We now extend our definitions to soliton graphs.

**Definition 17 (Graph Determinism)** *Let G be a soliton graph. G is called*

*(I)* deterministic*, if for all sets B of bursts for G $\mathscr{A}_B(G)$ is deterministic.*

*(II)* strongly deterministic*, if for all sets B of bursts for G $\mathscr{A}_B(G)$ is strongly deterministic.*

*(III)* perfectly deterministic*, if for all sets B of bursts for G $\mathscr{A}_B(G)$ is perfectly deterministic.*

For our statements about graph determinism it is important to consider soliton graphs that can not be decomposed into independent sub-graphs. In the case of a single wave, meaning the case of a single soliton traversing a soliton graph, impervious paths may appear. A path is impervious if none of its edges is used by the soliton in any configuration trail [3]. An example of an impervious path is the path h-i-j-k in Figure 1. The soliton has to enter the soliton graph either via node 1 or node 2, hence by traversing an edge with weight 1. Since it has to use an edge with weight 2 next, it can only move towards the cycle on the respective side. On its way back, on node h or k, respectively, it has to traverse an edge with weight 2 in the next step, still not allowing it to enter the path h-i-j-k and forcing it to leave the soliton graph via the node it entered the graph through. In order to formalize this idea for the case of multiple solitons being present we give the following definitions.

**Definition 18 (Used Edge)** *Let $G_0 = (N,E,w)$ be a soliton graph, let $n,n' \in N$ and let b be a burst. The edge $(n,n')$ is said to be* used *in a configuration trail $(G_0,\pi_0),(G_1,\pi_1),...,(G_k,\pi_k)$ with $\pi_0 = \pi_b$ if there exists a soliton i and a timestep j with $0 < j \leq k$, $\pi_{j-1}(i) = n$ and $\pi_j(i) = n'$.*

**Definition 19 (Impervious Path)** *Let $G = (N,E,w)$ be a soliton graph and let $n,n' \in N$. A path from n to $n'$ is said to be* impervious *if none of its edges are used in a configuration trail in any $G' \in States(G,B)$ with any set of bursts B for G.*

For the case of a single wave, if a soliton graph contains impervious paths then it can be decomposed into multiple connected components. Soliton graphs which, after the removal of impervious paths, are connected, are called indecomposable. For more details see [3].

**Definition 20 (Indecomposable Soliton Graph)** *Let G be a soliton graph. G is said to be* indecomposable *if it does not contain an impervious path.*

**Definition 21 (Chestnut)** *An indecomposable soliton graph is called a* chestnut *if it consists of a single cycle of even length and some paths leading into it with the following conditions:*

*(I) Entry points of different paths entering the cycle have even distances;*

*(II) Paths leading to the cycle may meet only at even distances from entry into the cycle.*

For more details see [3].

**Proposition 2** *Let $G = (N,E,w)$ be an indecomposable soliton graph. If G is a chestnut, then it is not strongly deterministic.*

*Proof.* As $G$ is a chestnut, the graph $(N,E)$ contains a cycle of even length (at least 4), that is there is an integer $k \geq 2$ and a path

$$n_0, n_1, \ldots, n_{2k}$$

with $n_0 = n_{2k}$ and $n_i \neq n_j$ for $0 \leq i < j < 2k$. For every exterior node $e$, there is a path leading to the cycle. Without loss of generality, let $m_0, m_1, \ldots m_\ell$ be such path with $\ell \geq 1$, $m_0 = e$ and $m_\ell = n_s$ for some $s$, $0 \leq s < 2k$. If $w(m_{\ell-1}, m_\ell) = 2$, then $w(n_s, n_{s'}) = w(n_s, n_{s''}) = 1$, where $s' = (s+1) \bmod 2k$ and $s'' = (s-1) \bmod 2k$. As the length of the cycle is even and two edges with weight 2 cannot meet in a soliton graph, there is a node $n_r \neq n_s$ in the cycle such that $w(n_r, n_{r'}) = w(n_r, n_{r''}) = 1$, $r' = (r+1) \bmod 2k$ and $r'' = (r-1) \bmod 2k$. An example graph with these properties is visualized in Figure 4. Because $G$ is a soliton graph, $d(n_r) = 3$ and $|r-s|$ is odd. Thus, $G$ is not a chestnut. Hence, $w(m_{\ell-1}, m_\ell) = 1$. Consequently, without loss of generality, $w(n_s, n_{s'}) = 2$ and $w(n_s, n_{s''}) = 1$ (Figure 5) and there is a total legal configuration trail for the burst $(e,e)\perp$. This is seen as follows: the soliton enters the cycle via edge

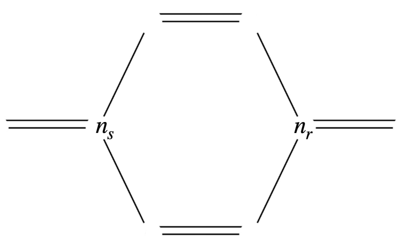


Figure 4: A cycle with even length and two edges with weight 2 leading into it.

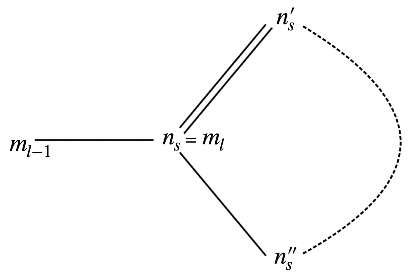


Figure 5: A node of degree 3 as entry point of a cycle.

$(m_{l-1}, n_s)$ and changing its weight to 2. It has to continue to $n'_s$. After completing the cycle it has moved from $n''_s$ to $n_s$ via an edge with weight 1 and must leave the cycle to $m_{l-1}$.

Now, consider the burst $b = (e,e)\|_1(e,e)\perp$. After the first soliton from $b$ has reached $n_{s'}$, the second one is at node $n_s$ and must follow the first soliton to $n_{s'}$, since otherwise the two solitons would collide inside the cycle eventually. When the first soliton has reached $n_s$ again, it must continue to $n_{s'}$ because it has traversed $(n_{s''}, n_s)$ with weight 1 and $w(n_s, m_{\ell-1}) = 1$. In the next step (when the second soliton is at $n_s$), the second soliton has the option to leave the cycle to $m_{\ell-1}$ or to further follow the first soliton in the cycle. After completing another round through the cycle, exactly the same situation will be encountered again. This situation is depicted in Figure 2, configuration 11.

Whenever the second soliton behaved to leave the cycle, the first soliton will be able to complete its path to $n_s$ and then also leave the cycle from $n_s$ to $m_{\ell-1}$ and on to $e$. In conclusion, there is more than one total legal configuration trail for $b$. Hence, $G$ is not strongly deterministic. □

Looking at the details in this proof, several (an infinite number of) imperfect configuration trails, but only one perfect configuration trail, exist. That is why one might wonder, whether all chestnuts are perfectly deterministic. The following statement disproves this assumption.

**Proposition 3** *There is a chestnut G which is not perfectly deterministic.*

*Proof.* Let $G$ be the chestnut in configuration $a$ in Figure 6 and let $b_G = (1,1)\|_3(3,1)\|_1(3,1)\perp$ be a burst. There are two total legal configuration trails for $G$ and $b_G$. We show selected configurations of both trails in Figure 6, which are $a$, $b$, $c$, $d_1$ for the first and $a$, $b$, $c$, $d_2$ for the second trail. They differ in the third soliton, depicted as a black diamond, moving downwards to node $g$ in $d_1$ and upwards to node $e$ in $d_2$. Therefore, both configurations trails are perfect. As both can be continued to total legal configuration trails, $G$ is not perfectly deterministic. □

**Proposition 4** *Let $G = (N,E,w)$ be an indecomposable soliton graph. If $(N,E)$ is a tree, then G is strongly deterministic.*

*Proof.* Let $G = (N,E,w)$ be an indecomposable soliton graph, $X$ be the set of its exterior nodes and $B$ a set of bursts over $X \times X$. Let $(n,n')$ be any pair of exterior nodes. If $(N,E)$ is a tree, then there is exactly one path $n_0, n_1, \ldots n_k$ from $n_0 = n$ to $n_k = n'$ such that $i \neq j$ implies $n_i \neq n_j$, that is, no node occurs more than once. In every total legal configuration trail $C$ for $G$ and a burst $b \in B$, condition 2.*(c)v.* of Definition 7 guarantees that only paths with that property can be a soliton path of a soliton in $b$ (solitons are not allowed to turn around spontaneously). Consequently, for every soliton $i$ in $b \in B$ and every total legal configuration trail $C$ for $G$ and $b$, there is at most one soliton path of soliton $i$ in $C$. Therefore, the automaton $\mathscr{A}_B(G)$ is strongly deterministic. As $B$ was arbitrary, $G$ is strongly deterministic. □

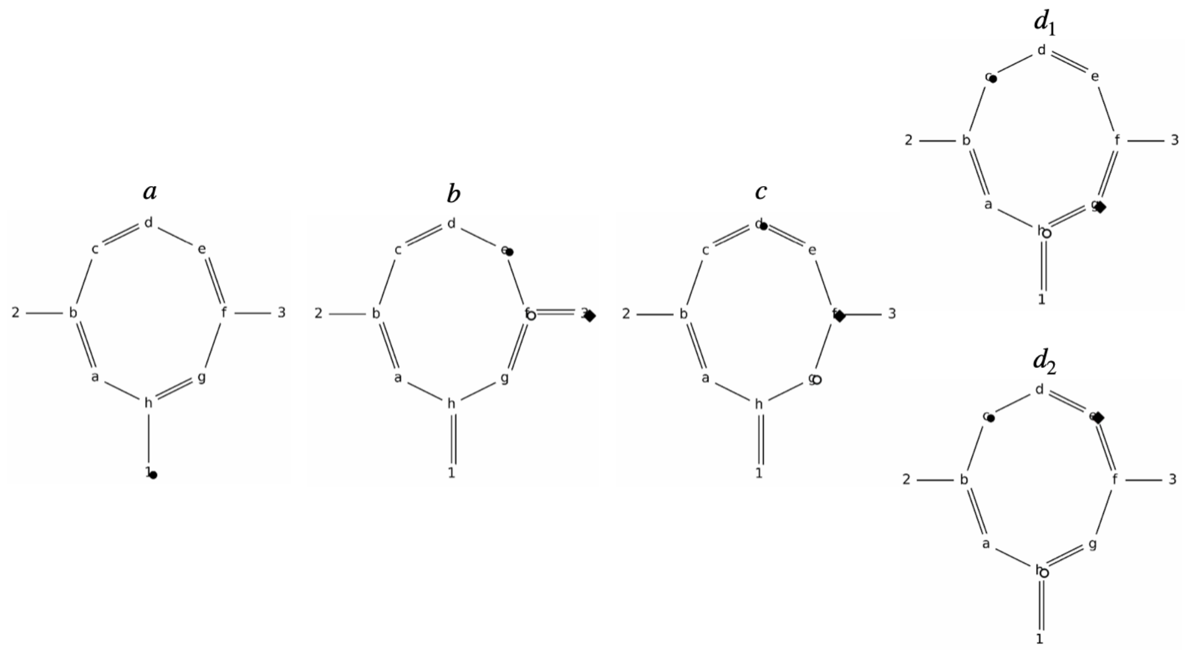


Figure 6: Selected configurations of two configuration trails for the burst $(1,1)\|_3(3,1)\|_1(3,1)\bot$. The first soliton is depicted as a black pebble, the second soliton as a white pebble and the third soliton as a black diamond. Configurations $a$, $b$ and $c$ appear in both configuration trails, while $d_1$ is part of the first trail and $d_2$ is part of the second trail.

**Theorem 5** *Let $G = (N,E,w)$ be an indecomposable soliton graph. $G$ is strongly deterministic if and only if $(N,E)$ is a tree.*

*Proof.* For single-soliton automata ([3]) it is known that an indecomposable solition graph is strongly deterministic if and only if $G$ is a chestnut or $(N,E)$ is a tree, see Proposition 5.4 in [3]. Proposition 31 of [1] implies for every soliton graph $G$ with set $X$ of exterior nodes that the single-soliton automaton based on $G$ is the soliton automaton $\mathscr{A}_B(G)$ where $B = \{(n,n')\bot \mid n,n' \in X\}$.[1] In conclusion, if $G$ is neither a chestnut nor is $(N,E)$ a tree, then there is a set of bursts $B$ such that $\mathscr{A}_B(G)$ is not strongly deterministic, thus $G$ is not strongly deterministic. By Proposition 2, $G$ is not strongly deterministic, if it is a chestnut. Therefore, if $G$ is strongly deterministic, then $(N,E)$ is a tree. By Proposition 4, the statement follows. □

## 4 Concluding Remarks

So far, the restriction for soliton automata to be (strongly) deterministic has only been investigated for the single-soliton case in the literature, see [3]. In [2, 9] several concepts of determinism have been defined for multi-soliton automata, but they have not been further investigated. In the present paper, the new notion of perfect determinism is defined, forming a weaker requirement than strong determinism but a stricter requirement than determinism. A characterization of strongly deterministic soliton graphs is given that is deviating from the known result for single-soliton automata. An example of a soliton

[1] See also Definition 10 in [2].

graph is presented that is strongly deterministic in the single-soliton case but is not even perfectly deterministic in the multi-soliton case. The degree of non-determinism is shown to be a connected measure of descriptional complexity for soliton automata.

The results use the condition that the soliton graphs are indecomposable, that is, there are no impervious paths in the soliton graphs. An interesting research question is whether impervious paths can appear at all in soliton graphs in the multi-soliton case. A soliton passing a node of a path that is impervious to that soliton opens the path for a second soliton following. The question is whether or not this principle can be generalized to open an unbounded number of impervious paths which may be "hidden" behind each other without eventually causing collisions so that each soliton can leave the graph again, constituting a total legal configuration trail for the respective burst.

In addition to the characterization of strongly deterministic soliton graphs one could also seek to characterize perfectly deterministic and deterministic soliton graphs. Another field of future research is the investigation of the transition monoids of multi-soliton automata.

## References

[1] Henning Bordihn & Helmut Jürgensen (2022): *Multi-Wave Soliton Automata*. Journal of Automata, Languages and Combinatorics 27, pp. 1–3, 91–130, doi:10.25596/jalc-2022-091.

[2] Henning Bordihn & Helena Schulz (2025): *Determinism and Simulation of Soliton Automata*. Journal of Automata, Languages and Combinatorics. Accepted at JALC.

[3] Jürgen Dassow & Helmut Jürgensen (1990): *Soliton automata*. Journal of Computer and System Sciences 40(2), pp. 154–181, doi:10.1016/0022-0000(90)90010-I.

[4] Jürgen Dassow & Helmut Jürgensen (1991): *Deterministic Soliton Automata with a Single Exterior Node*. Theor. Comput. Sci. 84(2), pp. 281–292, doi:10.1016/0304-3975(91)90164-W.

[5] Jürgen Dassow & Helmut Jürgensen (1993): *Deterministic Soliton Automata with at Most One Cycle*. J. Comput. Syst. Sci. 46(2), pp. 155–197, doi:10.1016/0022-0000(93)90002-E.

[6] Tore Koss (2022): *Reverting and combining soliton bursts*. J. Autom. Lang. Comb. 27(1–3), pp. 179–186, doi:10.25596/jalc-2022-179.

[7] P. S. Lomdahl (1984): *What is a soliton?* Los Alamos Science 10, pp. 27–31.

[8] Y. Lu (1988): *Solitons & Polarons in Conducting Polymers*. World Scientific, Singapore, doi:10.1142/0242.

[9] Helena Schulz (2023): *Untersuchungen zum Determinismuskonzept bei Mehrwellen-Soliton-Automaten anhand einer zu implementierenden Simulation*. Bsc thesis, Universität Potsdam.

# Operational State Complexity of Block Languages

Guilherme Duarte     Nelma Moreira     Rogério Reis[*]

CMUP & DCC, Faculdade de Ciências da Universidade do Porto
Rua do Campo Alegre, 4169-007 Porto, Portugal
{guilherme.duarte,nelma.moreira,rogerio.reis}@fc.up.pt

Luca Prigioniero

Department of Computer Science, Loughborough University
Epinal Way, Loughborough LE11 3TU, United Kingdom
l.prigioniero@lboro.ac.uk

In this paper we consider block languages, namely sets of words having the same length, and study the deterministic and nondeterministic state complexity of several operations on these languages. Being a subclass of finite languages, the upper bounds of operational state complexity known for finite languages apply for block languages as well. However, in several cases, smaller values were found. Block languages can be represented as bitmaps, which are a good tool to study their minimal finite automata and their operations, as we illustrate here.

## 1 Introduction

In this paper we consider finite languages where all words have the same length, which are called *homogeneous* or *block* languages. Their investigation is mainly motivated by their applications to several contexts such as code theory [10] and image processing [8, 9]. We will focus on the state complexity of operations [5, 14]. The *deterministic* (*nondeterministic*) *state complexity* of a regular language $L$ is the number of states of its minimal complete deterministic (nondeterministic, resp.) finite automaton.

Here, we are interested in operational complexity, that is the size of the model accepting a language resulting from an operation performed on one or more languages. In particular, the *state complexity of an operation* (or *operational state complexity*) on regular languages is the worst-case state complexity of a language resulting from the operation, considered as a function of the state complexities of the operands. As a subclass of finite languages, block languages inherit some properties known for that class, which differ from the existing ones for the class of regular languages [2]. Due to the fact that, in our case, all words have the same length, there are some gains in terms of state complexity. For example, it is known that the elimination of nondeterminism from an $n$-state nondeterministic finite automaton for a block language costs $2^{\Theta(\sqrt{n})}$ in size [9], which is smaller than the general case for finite languages.

A block language can be well characterized by its characteristic function which we denote by *bitmap*. In particular, given an alphabet of size $k$ and a length $\ell$, a block language can be represented by a binary string of length $k^\ell$, also called *bitmap*, in which each symbol (or *bit*) indicates whether the correspondent word, according to the lexicographical order, belongs to the language (bit equal to 1) or not (bit equal to 0). Duarte et al. [4] used this representation as a tool to investigate several properties of block languages, namely how to convert bitmaps into minimal deterministic and nondeterministic finite automata and what are the maximal numbers of states that the resulting automata can have. In this paper, we also

[*]This work was partially supported by CMUP, member of LASI, which is financed by national funds through FCT – Fundação para a Ciência e a Tecnologia, I.P., under the projects with reference UIDB/00144/2020 and UIDP/00144/2020.

use bitmaps for studying the complexity of operations on block languages. Due to the distinguishing property of the length of the words, we study Boolean binary operations over block languages with the same length as well as block complement (i.e., $\Sigma^\ell \setminus L$). Nonetheless, we also consider operations such as concatenation, Kleene star, and Kleene plus, which are not closed for the class of block languages of a given length.

The paper is organized as follows. In the next section we fix notation and review the bitmap representation for block languages. In Section 3, we revise the operational state complexities of basic operations on finite languages. Then, we study the state complexity on block languages for the following operations: reversal, word addition and removal, intersection, union, block complement, concatenation, Kleene star, and plus. In Table 2, we summarize our results and we conclude the paper in Section 4 by describing further lines of investigation.

## 2 Preliminaries

In this section we review some basic definitions about finite automata and languages and fix notation. Given an *alphabet* $\Sigma$, a *word* $w$ is a sequence of symbols, and a *language* $L \subseteq \Sigma^\star$ is a set of words on $\Sigma$. The empty word is denoted by $\varepsilon$. The *(left) quotient* of a language $L$ by a word $w \in \Sigma^\star$ refers to the set $w^{-1}L = \{w' \in \Sigma^\star \mid ww' \in L\}$. The *reversal* of a word $w = \sigma_0\sigma_1 \cdots \sigma_{n-1}$ is denoted by $w^{\mathrm{R}}$ and is obtained by reversing the order of the symbols of $w$, that is $w^{\mathrm{R}} = \sigma_{n-1}\sigma_{n-2} \cdots \sigma_0$. The reversal of a language $L$ is $L^{\mathrm{R}} = \{\, w^{\mathrm{R}} \mid w \in L \,\}$. Given two integers $i, j$ with $i < j$, let $[i, j]$ denote the set of integers from $i$ to $j$, including both $i$ and $j$, namely $\{i, \dots, j\}$. Moreover, we shall omit the left bound if it is equal to 0, thus $[j] = \{0, \dots, j\}$.

A *nondeterministic finite automaton* (NFA) is a five-tuple $A = \langle Q, \Sigma, \delta, I, F\rangle$ where $Q$ is a finite set of states, $\Sigma$ is a finite alphabet, $I \subseteq Q$ is the set of initial states, $F \subseteq Q$ is the set of final states, and $\delta : Q \times \Sigma \to 2^Q$ is the transition function. We consider the *size* of an NFA as its number of states. The transition function can be extended to words and sets of states in the natural way. When $I = \{q_0\}$, we use $I = q_0$. An NFA accepting a non-empty language is *trim* if every state is accessible from an initial state and every state leads to a final state. Given a state $q \in Q$, the *right language* of $q$ is $\mathscr{L}_q(A) = \{\, w \in \Sigma^\star \mid \delta(q, w) \cap F \neq \emptyset \,\}$, and the *left language* is $\overleftarrow{\mathscr{L}}_q(A) = \{\, w \in \Sigma^\star \mid q \in \delta(I, w) \,\}$. The *language accepted* by $A$ is $\mathscr{L}(A) = \bigcup_{q \in I} \mathscr{L}_q(A)$. An NFA $A$ is *minimal* if it has the smallest number of states among all NFAs that accept $\mathscr{L}(A)$.

An NFA is *deterministic* (DFA) if $|I| = 1$ and $|\delta(q, \sigma)| \leq 1$, for all $(q, \sigma) \in Q \times \Sigma$. We can convert an NFA $A$ into an equivalent DFA $\mathsf{D}(A)$ using the well-known subset construction. Two states $q_1$, $q_2$ are *equivalent* (or *indistinguishable*) if $\mathscr{L}_{q_1}(A) = \mathscr{L}_{q_2}(A)$. A *minimal* DFA has no different equivalent states, every state is reachable and it is unique up to isomorphism.

The *state complexity* of a language $L$, $\mathsf{sc}(L)$, is the size of its minimal DFA. The *nondeterministic state complexity* of a language $L$, $\mathsf{nsc}(L)$, is defined analogously.

A trim NFA $A = \langle Q, \Sigma, \delta, I, F\rangle$ for a finite language of words of size at most $\ell$ is acyclic and ranked, i.e., the set of states $Q$ can be partitioned into $\ell + 1$ disjoint sets $Q_0 \cup Q_1 \cup \cdots \cup Q_\ell$, such that for every state $q \in Q_i$, $A$ reaches a final state by words of length at most $i$ ($Q_i = \{q \in Q \mid \forall w \in \Sigma^\star, \delta(q, w) \in F \implies |w| \leq i\}$) and all transitions from states of rank $i$ lead only to states in $i'$, with $i, i' \in [\ell]$ and $i' < i$. We define the *width* of a rank $i$, namely $\mathsf{w}(i)$, as the cardinality of the set $Q_i$, and the *width* of $A$ to be the maximal width of a rank, i.e., $\mathsf{w}(A) = \max_{i \in [\ell]} |Q_i|$. A DFA for a finite language is also ranked but it may have a *dead state* $\Omega$ which is the only state with a self-loop and without a rank. In a trim acyclic automaton, two states $q$ and $q'$ are equivalent if they are both in the same rank, either final or not final, and their transition functions lead to equivalent states, i.e., $\delta(q,w) \in F \iff \delta(q',w) \in F$, for each word $w \in \Sigma^*$. An acyclic DFA can be minimized by merging equivalent states and the resulting algorithm runs in linear time in the size of the automaton (Revuz algorithm, [1, 12]).

## 2.1 Block Languages and Bitmap Representation

Given an alphabet $\Sigma = \{\sigma_0, \dots, \sigma_{k-1}\}$ of size $k > 0$ and an integer $\ell > 0$, a *block language* $L \subseteq \Sigma^\ell$ is a set of words of length $\ell$ over $\Sigma$. Let $A = \langle Q, \Sigma, \delta, q_0, F\rangle$ be a *NFA* that accepts a block language with a single initial state. Because all accepted words have the same length, we can assume that the finite automata for block languages have only one final state, i.e., $F = \{q_f\}$, for some $q_f \in Q$. Moreover, as before, the set of states $Q$ can be partitioned into $Q_0 \cup Q_1 \cup \cdots \cup Q_\ell$ where $Q_i$ is the set of states with rank $i$ and $\delta(Q_i, \sigma) \subseteq Q_{i-1}$, where $i = 1, \dots, \ell$ and $\sigma \in \Sigma$. We also have a unique final state in rank 0, that is $Q_0 = F = \{q_f\}$. If $A$ is a DFA for a block language, then there exists an extra dead state $\Omega$. For each $q \in Q_i$ and $\sigma \in \Sigma$, either $\delta(q,\sigma) \in Q_{i-1}$ or $\delta(q,\sigma) = \Omega$ (but $q$ must have at least a transition to $Q_{i-1}$), for all $i \in [1,\ell]$.

Câmpeanu and Ho [3] estimated the maximal number of states of a minimal DFA accepting a block language. In the next lemma, we recall that result and related properties. In Fig. 1 the constraints on the widths of the ranks of a minimal DFA are depicted.

**Lemma 1.** *Let $L \subseteq \Sigma^\ell$ be block language over an alphabet of size $k$ and $\ell > 0$. Then, we have*

1. $\mathsf{sc}(L) \leq \frac{k^{\ell-r+1}-1}{k-1} + \sum_{i=0}^{r-1}(2^{k^i}-1)+1$, *where* $r = \min\{n \in [\ell] \mid k^{\ell-n} \leq 2^{k^n}-1\}$*;*
2. $r = \lfloor \log_k \ell \rfloor + 1 + x$, *for some* $x \in \{-1,0,1\}$*;*
3. *Let $A$ be a minimal DFA of maximal size for a block language. Then,* $\mathsf{w}(A) = \max\{k^{\ell-r}, 2^{k^{r-1}}-1\}$*, where* $\mathsf{w}(r-1) = 2^{k^{r-1}}-1$ *and* $\mathsf{w}(r) = k^{\ell-r}$*. Moreover, let $r_{k,\ell}$ be the rank that the width of $A$ is reached, either $r$ or $r-1$.*

*Proof (sketch).* The first statement was proven in [3, Corollary 10] and follows from the fact that for each rank $i \in [\ell]$, we have that $\mathsf{w}(i) \leq 2^{k^i}-1$ and $\mathsf{w}(\ell-i) \leq k^i$. Then, for a DFA to have maximal size we have $\mathsf{w}(i) = 2^{k^i}-1$, for $i \in [r-1]$, and $\mathsf{w}(i) = k^{\ell-i}$, for $i \in [r,\ell]$. Finally, we need to add one for the dead state. The second statement was proven in [4]. The third statement follows from the first, noticing that $\mathsf{w}(r-1) = 2^{k^{r-1}}-1$ and depending on whether $k^{\ell-r} > 2^{k^{r-1}}-1$ or not. We set $r_{k,\ell}$ to be the rank such that $\mathsf{w}(A) = \mathsf{w}(r_{k,\ell})$. ☐

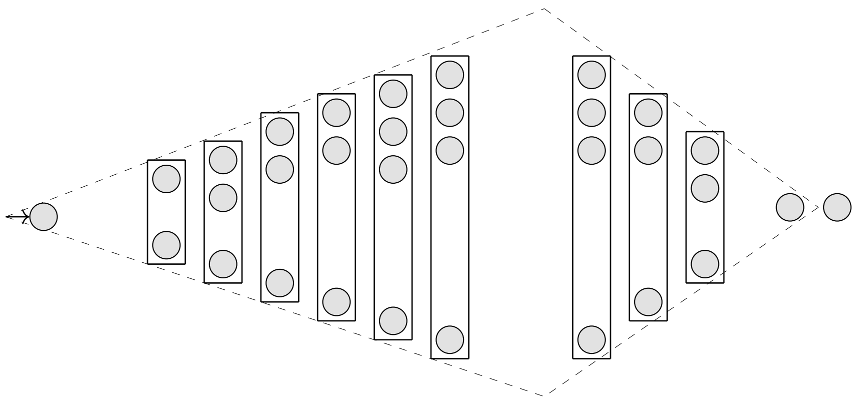


Figure 1: Constraints in the widths of the ranks of a minimal DFA for a block language. Each rank (except the last and the first ones) is represented by a rectangle. The rightmost state is the dead-state ($\Omega$).

A block language $L$ can be characterized by a word in $\{0,1\}^{k^\ell}$ called *bitmap* and denoted as

$$\mathsf{B}(L) = b_0 \cdots b_{k^\ell - 1},$$

where $b_i = 1$ if and only if $i \in [k^\ell - 1]$ is the index of $w$ in the lexicographical ordered list of all the words of $\Sigma^\ell$ and the word $w$ is in $L$. In this case, we denote $i$ by $\mathsf{ind}(w)$. The bitmap of a language can be denoted by $\mathsf{B}$ when it is unambiguous to which language the bitmap refers to. Reciprocally, given a bitmap $\mathsf{B} \in \{0,1\}^{k^\ell}$ and an alphabet $\Sigma$ of size $k$, $\mathscr{L}(\mathsf{B}) \subseteq \Sigma^\ell$ denotes the language represented by $\mathsf{B}$.

Boolean bitwise operations on bitmaps trivially correspond to Boolean set operations on block languages of the same length. Formally, given two bitmaps $\mathsf{B}_1, \mathsf{B}_2 \in \{0,1\}^{k^\ell}$, the bitmap $\mathsf{B}_1 \circ \mathsf{B}_2$ is obtained by carrying out the bitwise operation $\circ \in \{\vee, \wedge\}$ between $\mathsf{B}_1$ and $\mathsf{B}_2$, while $\overline{\mathsf{B}_1}$ is the bitwise complement of $\mathsf{B}_1$.

Duarte et al. [4] studied block languages using bitmaps. In particular, it was shown how to convert bitmaps into minimal deterministic and nondeterministic finite automata.

A bitmap $\mathsf{B} \in \{0,1\}^{k^\ell}$ of a language $L \subseteq \Sigma^\ell$, for some $\ell > 0$, can be split into factors of length $k^i$, for $i \in [\ell]$. Let $s^i_j = b_{jk^i} \cdots b_{(j+1)k^i - 1}$ denote the $j$-th factor of length $k^i$, for $j \in [k^{\ell-i} - 1]$. Since each factor of length $k^i$ can also be split into $k$ factors, $s^i_j$ is inductively defined as:

$$s^i_j = \begin{cases} b_j, & \text{if } i = 0, \\ s^{i-1}_{jk} \cdots s^{i-1}_{(j+1)k-1}, & \text{otherwise.} \end{cases}$$

Furthermore, let $i \in [\ell]$, $j \in [k^{\ell-i} - 1]$, and $w \in \Sigma^{\ell-i}$ be the word of index $j$ of length $\ell - i$, in lexicographic order. Then, $s^i_j$ corresponds to the bitmap of $w^{-1}L$.

Given a bitmap $\mathsf{B} \in \{0,1\}^{k^\ell}$, let $\mathscr{B}_i$ be the set of factors of $\mathsf{B}$ of length $k^i$, for $i \in [\ell]$, in which there is at least one bit different from zero, that is,

$$\mathscr{B}_i = \{s \in \{0,1\}^{k^i} \mid \exists j \in [k^{\ell-i} - 1] : s = s^i_j \neq 0^{k^i}\}.$$

**Example 1.** *Let $\Sigma = \{a,b\}$, $k = 2$, and $\ell = 4$. Consider*

$$L = \{aaaa, aaba, aabb, abab, abba, abbb, babb, bbaa, bbab, bbba\}.$$

*The bitmap of $L$ is* $\mathsf{B}(L) = 1011011100011110$. *Moreover, we have that $s^2_0 = 1011$ is the bitmap of $(aa)^{-1}L = \{aa, ba, bb\}$, $s^3_1 = 00011110$ the bitmap of $b^{-1}L = \{abb, baa, bab, bba\}$, and $s^4_0 = \mathsf{B}$ the bitmap of $L$. We also have $\mathscr{B}_0 = \{1\}$, $\mathscr{B}_1 = \{01, 10, 11\}$, $\mathscr{B}_2 = \{0001, 0111, 1011,\ 1110\}$, $\mathscr{B}_3 = \{00011110, 10110111\}$, and $\mathscr{B}_4 = \{\mathsf{B}\}$.*

The sets $\mathscr{B}_i$ are related to the states of the minimal finite automata representing the block language with bitmap $\mathsf{B}$, as shown in [4]. We now briefly recall such a result.

Given a bitmap $\mathsf{B}$ associated with a block language $L \subseteq \Sigma^\ell$, with $|\Sigma| = k$ and $\ell > 0$, one can directly build the minimal DFA $A$ for $L$. Formally, $A = \langle Q \cup \{\Omega\}, \Sigma, \delta, \mathsf{B}, \{1\} \rangle$, where the set of states $Q$ correspond to bitmap factors, that is, $Q = \bigcup_{i \in [\ell]} \mathscr{B}_i$; the initial state is the bitmap $\mathsf{B}$; and the final state is the bitmap factor 1. The transition function $\delta$ is given by the decomposition of each bitmap factor. Let $s \in \mathscr{B}_i$, where $s = s_0 \cdots s_{k-1}$ and $|s_j| = k^{i-1}$, for $i \in [1, \ell]$ and $j \in [k-1]$. Then, the transition function contains $\delta(s, \sigma_j) = s_j$. Moreover, the states in $\mathscr{B}_i$ have rank $i$. The *DFA* can be completed considering transitions to $\Omega$ (dead-state) in the usual way.

A similar construction can be used to obtain a minimal NFA for $L$, where each rank will contain the minimal cover of the sets $\{\mathscr{B}_i\}_{i \in [\ell]}$. The main difference with the deterministic case is that the quotients of the language, corresponding to factors from the bitmap, are represented by a set of states, instead of a single one. For $i \in [\ell]$ and $s \in \mathscr{B}_i$, a *cover* of $s$ is a set of $n > 0$ binary words $\{c_0, \ldots, c_{n-1}\}$, where $|c_j| = s$, for all $j \in [n-1]$, such that the disjunction of the set equals $s$, that is, $\bigvee_{j\in[n-1]} c_j = s$. Since bitmap factors correspond to block languages, we have $\mathscr{L}(s) = \bigcup_{j\in[n-1]} \mathscr{L}(c_j)$. A set $\mathscr{C}_i$ of binary words of length $k^i$ is a cover for the set $\mathscr{B}_i$ if all the words in $\mathscr{B}_i$ are covered by $\mathscr{C}_i$. For instance, it can be easily noticed that $\mathscr{B}_i$ covers itself. Moreover, we say that $\mathscr{C}_i$ is a *minimal cover* for $\mathscr{B}_i$ if there is no other set $\mathscr{C}_i'$ smaller than $\mathscr{C}_i$ that covers $\mathscr{B}_i$. Then, a minimal NFA $A = \langle Q, \Sigma, \delta, \{\mathsf{B}\}, \{1\}\rangle$ for $L$ can be constructed as follows. As indicated, the single final state is the factor 1. Additionally, we define the function $\rho : \{0,1\}^\star \to 2^{\{0,1\}^\star}$ that maps factors into covers, where initially we set $\rho(1) = \{1\}$. Next, for every rank $i = 1, \ldots, \ell$, we consider a minimal cover $\mathscr{C}_i$ for $\mathscr{B}_i$, and we set, for every $s \in \mathscr{B}_i$, $\rho(s) = \{c_0, \ldots, c_{n-1}\} \subseteq \mathscr{C}_i$, such that $\rho(s)$ covers $s$. Furthermore, we set $\mathscr{C}_i$ as the set of states at rank $i$ of $A$, and so $Q = \bigcup_{i\in[\ell]} \mathscr{C}_i$. The transitions from rank $i$ to rank $i-1$ will then be determined in a similar way to the DFA construction. For each state $c \in \mathscr{C}_i$ in rank $i$, we decompose $c$ into $c_0 \cdots c_{k-1}$, where $|c_j| = k^{i-1}$, for every $j \in [k-1]$, and set $\delta(c, \sigma_j) = \rho(c_j)$, if only $c_j \neq 0^{k^{i-1}}$, where $\sigma_j \in \Sigma$. We must also guarantee that $\rho$ is defined for each $c_j$ or, alternatively, that $c_j \in \mathscr{B}_{i-1}$. For that, we need to limit the search space of the cover $\mathscr{C}_i$, so that each word in the set is a composition of $k$ words from $\mathscr{B}_{i-1}$ or $0^{k^{i-1}}$. Formally, $\mathscr{C}_i \subseteq (\mathscr{B}_{i-1} \cup 0^{k^{i-1}})^k \setminus 0^{k^i}$. Also, $\mathscr{B}_\ell = \{\mathsf{B}\}$, so the minimal cover for $\mathscr{B}_\ell$ is itself. This result implies that $\mathsf{B}$ will be the single initial state at rank 0.

In this paper, bitmaps will be a useful tool for the study of operational state complexities. Not only languages are easily represented by their bitmaps but also bitwise operations on bitmaps mimic the operations on languages.

## 3 Operational Complexity

In this section we consider operations on block languages using their bitmap representations and study both the deterministic and nondeterministic state complexity of those operations. More precisely, the *operational state complexity* is the worst-case state complexity of a language resulting from the operation, considered as a function of the state complexities of the operands. For instance, the state complexity of the union of two block languages can be stated as follows: given an $m$-state DFA $A_1$ and an $n$-state DFA $A_2$, how many states are sufficient and necessary, in the worst case, to accept the language $L(A_1) \cup L(A_2)$ by a DFA?

An upper bound can be obtained by providing an algorithm that, given DFAs for the operands, constructs a DFA that accepts the resulting language, and the number of states, in the worst case, of the resulting DFA is an upper bound for the state complexity of the referred operation. To show that an upper bound is tight, a family of languages (one language, for each possible value of the state complexity) for each operation must be given such that the resulting automata achieve that bound. We can call those families *witnesses* or *streams*.

We will mainly consider operations under which the family of block languages is closed, i.e., the resulting language is also a block language. In particular, we will consider the union and intersection of two block languages whose words are of the same length, the concatenation of two arbitrary block languages, the reversal, the complement of block languages closed to the block (i.e., $\Sigma^\ell \setminus L$), and word addition and removal from a block language. We will also analyze the Kleene star and plus operations of block languages, which in general do not yield a block language.

Of course, the upper bounds of operational state complexity known for finite languages apply for block languages. In Table 1, we review some complexity results for finite languages. The first two

Table 1: Some complexity bounds for finite languages

| | Upper bound | $\|\Sigma\|$ | Ref. | | | |
|---|---|---|---|---|---|---|
| NFA $\to$ DFA | $\Theta(k^{\frac{m}{1+\log k}})$ | 2 | [13] | | | |
| $\mathsf{sc}(L)$ | $\frac{k^{\ell+2}}{\ell(k-1)^2\log_2 k}(1+o(1))$ | 2 | [3] | | | |
| | $\mathsf{sc}$ | $\|\Sigma\|$ | Ref. | $\mathsf{nsc}$ | $\|\Sigma\|$ | Ref. |
| $L_1 \cup L_2$ | $mn-(m+n)$ | $f(m,n)$ | [6] | $m+n-2$ | 2 | [7] |
| $L_1 \cap L_2$ | $mn-3(m+n)+12$ | $f(m,n)$ | [6] | $O(mn)$ | 2 | [7] |
| $\overline{L}$ | $m$ | 1 | | $\Theta(k^{\frac{m}{1+\log k}})$ | 2 | [7] |
| $L_1L_2$ | $(m-n+3)2^{n-2}-1, m+1 \geq n$ | 2 | [2] | $m+n-1$ | 2 | [7] |
| | $m+n-2$, if $p_1 = 1$ | 1 | | | | |
| $L^\star$ | $2^{m-3}+2^{m-p-2}, p \geq 2, m \geq 4$ | 3 | [2] | $m-1, m>1$ | 1 | [7] |
| | $m-1$, if $f=1$ | 1 | | | | |
| $L^R$ | $O(k^{\frac{m}{1+\log k}})$ | 2 | [2] | $m$ | 2 | [7] |

lines give the bounds for the determinization of an $m$-state NFA and the asymptotic upper bound of the maximal size of a minimal DFA, respectively. For the operational state complexities, we consider $|\Sigma| = k$ or $|\Sigma| = f(\overline{m})$ if a growing alphabet is taken into account, $|F_i| = f_i$, and $p_i = |F_i - \{q_i\}|$, for the $i$-th operand and its set of final states, $F_i$.

Additionally, we show how to build the bitmap of the language resulting by applying each operation and also present a family of witness languages parameterized by the state complexity of the operands to show that the bounds provided are tight. In general, other additional parameters are the length $\ell$ of the words and the widths of each rank.

## 3.1 Reversal

In the following, given a bitmap $\mathsf{B}$ of a block language $L \subseteq \Sigma^\ell$, $|\Sigma| = k$, and $\ell > 0$, we compute the bitmap for the reversal language $L^{\mathrm{R}}$. Recall that the perfect shuffle of length 1, denoted $\sqcup\!\sqcup_1$, of two words $u = u_0 \cdots u_{n-1}$ and $v = v_0 \cdots v_{n-1}$ of the same length is obtained by interleaving the letters of $u$ and $v$, namely $u \sqcup\!\sqcup_1 v = u_0v_0 \cdots u_{n-1}v_{n-1}$. If $j$ is a divisor of $n$, the perfect shuffle of length $j$, denoted $\sqcup\!\sqcup_j$, of $u$ and $v$ is the perfect shuffle of blocks of length $j$, that is,

$$u \sqcup\!\sqcup_j v = u_0 \cdots u_{j-1}v_0 \cdots v_{j-1} \cdots u_{n-j} \cdots u_{n-1}v_{n-j} \cdots v_{n-1}.$$

Finally, if $|u| = |v|$, we denote $u \sqcup\!\sqcup_j v$ by $\sqcup\!\sqcup_j^2(uv)$. This can be generalized for any number $m \geq 2$ of words $w_0, \ldots, w_{m-1}$ of the same length by considering the perfect shuffle of blocks of length $j$ taken from each of the $w_i$ words, that is, $\sqcup\!\sqcup_j^m(w_0 \cdots w_{m-1})$. For $j = 1$ and $w_i = w_{i,0} \cdots w_{i,n-1}$, $i \in [m-1]$, one has

$$\sqcup\!\sqcup_1^m(w_0 \cdots w_{m-1}) = w_{0,0} \cdots w_{n-1,0}w_{0,1} \cdots w_{m-1,1} \cdots w_{0,n-1} \cdots w_{m-1,n-1}.$$

Let $\mathsf{R}_0 = \mathsf{B}$ and $\mathsf{R}_i = \sqcup\!\sqcup_{k^{i-1}}^k(\mathsf{R}_{i-1})$, for $i \in [1, \ell-1]$ and $k = |\Sigma|$.

**Lemma 2.** *Let $L \subseteq \Sigma^\ell$ be a block language, for some $\ell > 0$. The bitmap for the reversal of $L$, namely $\mathsf{B}(L^{\mathrm{R}})$, is $\mathsf{R}_{\ell-1}$.*

*Proof.* Let us prove that $\mathscr{L}(\mathsf{R}_{\ell-1}) = L^{\mathrm{R}}$. For $i = 0$, of course $\mathscr{L}(\mathsf{R}_0) = \mathscr{L}(\mathsf{B}) = L$. Next, for $i = 1$ we have $\mathsf{R}_1 = ⧢_1^k(\mathsf{B})$. This operation performs the cyclic permutation $S_1 = (0\ 1\ \cdots\ \ell-1)$ in each word of $\mathscr{L}(\mathsf{B})$, that is, each symbol of every word in $\mathscr{L}(\mathsf{B})$ is shifted one position to their right and the last symbol becomes the first. The following operation, $\mathsf{R}_2 = ⧢_k^k(\mathsf{R}_1)$, performs the permutation $S_2 = (1\ 2\ \cdots\ \ell-1)$ in every word of $\mathscr{L}(\mathsf{R}_1)$. Analogously, in this transformation each symbol apart from the first of every word in $\mathscr{L}(\mathsf{R}_1)$ is shifted one position to their right but the last symbol becomes the second. In general, the $j$-th shuffle performs the permutation $S_j = (j-1\ j\ \cdots\ \ell-1)$, for $j \in [1, \ell-1]$. The composition of the transformations $S_1, S_2, \ldots, S_{\ell-1}$ ensure that $\mathscr{L}(\mathsf{R}_{\ell-1}) = L^{\mathrm{R}}$ [11]. $\square$

**Example 2.** *Let $\Sigma = \{a,b\}$ and $\ell = 3$. Let $\mathsf{B} = b_0b_1b_2b_3b_4b_5b_6b_7$ be a bitmap for a block language $L$ such that $b_0 = b_3 = b_4 = 1$, and the remaining bits are $0$. We have*

$$\begin{aligned}
\mathsf{R}_0 &= b_0b_1b_2b_3b_4b_5b_6b_7 \quad \text{and} \quad \mathscr{L}(\mathsf{R}_0) = \{aaa, abb, baa\},\\
\mathsf{R}_1 &= b_0b_4b_1b_5b_2b_6b_3b_7 \quad \text{and} \quad \mathscr{L}(\mathsf{R}_1) = \{aaa, bab, aba\},\\
\mathsf{R}_2 &= b_0b_4b_2b_6b_1b_5b_3b_7 \quad \text{and} \quad \mathscr{L}(\mathsf{R}_2) = \{aaa, bba, aab\},
\end{aligned}$$

*and $\mathscr{L}(\mathsf{R}_2) = L^{\mathrm{R}}$, as desired.*

Now we turn to the analyze of the state complexity of this operation. The DFA for the reversal of a block language $L \subseteq \Sigma^\ell$, with $\ell > 0$, is given by reversing each transition on a DFA for $L$ and then determinising the resulting NFA. The cost of the determinisation of an $m$-state NFA for a block language is $2^{\Theta(\sqrt{m})}$ in size [9], so the state complexity of the reversal must also be limited by this bound.

**Corollary 1.** *Given an m-state DFA for a block language L, $2^{O(\sqrt{m})}$ states are sufficient for a DFA accepting $L^{\mathrm{R}}$.*

In the following, we show that this bound is tight. Let $\ell > 0$, $\Sigma = \{a,b\}$, $k = 2$, and consider Lemma 1. We define a family of languages, parametrized by $\ell$, that attain the maximal state complexity. For convenience, let $r_\ell = r_{2,\ell}$. Then, consider

$$\mathsf{MAX}_\ell = \{w_1w_2 \mid w_1 \in \Sigma^{\ell-r_\ell}, w_2 \in \Sigma^{r_\ell}, i = \mathsf{ind}(w_1),\ j = \mathsf{ind}(w_2),\ \text{and}\ (i+1) \wedge 2^j \neq 0\},$$

where we use the notation $i \wedge 2^j \neq 0$ to indicate that the $j$-th least significant bit of the binary representation of $i$, namely $i_{[2]}$, is 1. Informally, these languages contain words of size $\ell$ that can be split in $w_1$ of size $\ell - r_\ell$ and $w_2$ of size $r_\ell$, with corresponding indices $i = \mathsf{ind}(w_1)$ and $j = \mathsf{ind}(w_2)$, such that the $j$-th least significant bit of $(i+1)_{[2]}$ is 1.

**Proposition 1** ( [4]). *The minimal DFA A accepting the language $\mathsf{MAX}_\ell$ has maximal size and $t_\ell = \mathsf{w}(A) = \mathsf{w}(r_\ell) = \max(2^{\ell-r}, 2^{2^{r-1}} - 1)$. Moreover, let*

$$P_{t_\ell,r_\ell} = \prod_{i=1}^{t_\ell} \mathsf{pad}(i_{[2]}, 2^{r_\ell})^{\mathrm{R}},$$

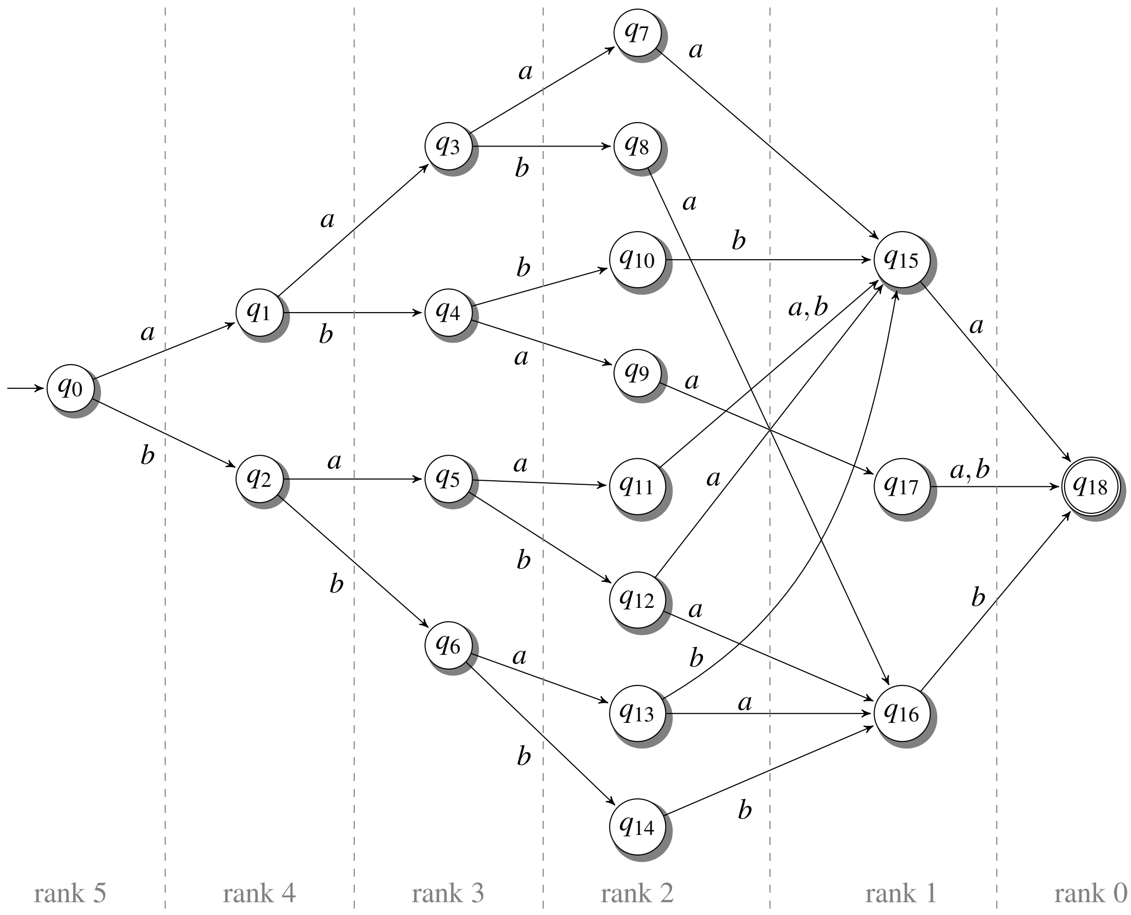
*where $\mathsf{pad}(s,t)$ is a function that adds leading zeros to a binary string s until its length equals t. Then the bitmap of the language $\mathsf{MAX}_\ell$ is given by*

$$\mathsf{B}(\mathsf{MAX}_\ell) = \begin{cases} P_{t_\ell,r_\ell}, & \text{if } t_\ell = 2^{\ell-r},\\ P_{t_\ell,r_\ell} \cdot 0^{2^{r_\ell}}, & \text{if } t_\ell = 2^{2^{r-1}} - 1. \end{cases}$$

**Example 3.** *For $\ell = 5$, we have*

$$\begin{aligned}\mathsf{MAX}_5 = \{&aaaaa, aabab, abaaa, abaab, abbba, baaaa,\\ &baaba, babab, babba, bbaaa, bbaab, bbaba, bbbbb\},\end{aligned}$$

*and* $\mathsf{B}(\mathsf{MAX}_5) = \prod_{i=1}^{8} \mathsf{pad}(i_{[2]}, 4)^{\mathsf{R}} = 10000100110000101010011011100001$. *The minimal DFA is represented below, where the dead state is omitted.*



$$\begin{tikzcd}
 & & & q_7 \arrow[rr, "a"] & & q_{15} \arrow[r, "a"] & \\
 & & q_3 \arrow[ur, "a"] \arrow[r, "b"] & q_8 \arrow[drr, "a"] & & & \\
 & q_1 \arrow[ur, "a"] \arrow[r, "b"] & q_4 \arrow[r, "b"] \arrow[dr, "a"] & q_{10} \arrow[rr, "b"] & & q_{15} & \\
\arrow[r] & q_0 \arrow[u, "a"] \arrow[d, "b"] & & q_9 \arrow[rr, "a"] & & q_{17} \arrow[r, "{a,b}"] & q_{18} \\
 & q_2 \arrow[r, "a"] \arrow[dr, "b"] & q_5 \arrow[r, "a"] \arrow[dr, "b"] & q_{11} \arrow[uurr, "{a,b}"] & & & \\
 & & q_6 \arrow[r, "a"] \arrow[dr, "b"] & q_{12} \arrow[uuurr, "a"] \arrow[drr, "a"] & & & \\
 & & & q_{13} \arrow[uuuurr, "b"] \arrow[rr, "a"] & & q_{16} \arrow[uuur, "b"] & \\
 & & & q_{14} \arrow[urr, "b"] & & &
\end{tikzcd}$$

Figure 2: The minimal DFA accepting the language $\mathsf{MAX}_\ell$ for $\ell = 5$. The sink-state is omitted, as well as all transitions from and to it.

The reversal of $\mathsf{MAX}_\ell$, namely $\mathsf{MAX}_\ell^{\mathsf{R}}$, has a minimal DFA whose width is at most $2^{r_\ell+1}$.

**Lemma 3.** *Let $r$ and $r_\ell$ be defined as before for $\ell > 0$ and alphabet size $k = 2$. A minimal DFA $B$ such that $\mathsf{w}(B) \leq 2^{r_\ell+1}$ is sufficient to accept the reversal of the language $\mathsf{MAX}_\ell$.*

*Proof.* Let $B$ be a DFA such that the last $r_\ell + 1$ ranks have maximal width, that is, $2^{\ell-r'}$ states in each rank $r' \in [\ell - r_\ell, \ell]$. In particular, the width of the rank $\ell - r_\ell$ is $2^{r_\ell}$. We can order the states in this rank in such way that $q_j$ is the state whose left language is the reverse of the $j$-th word of $\Sigma^{r_\ell}$, i.e., $\overleftarrow{\mathscr{L}}(q_j) = \{w^{\mathsf{R}}\}$ where $j = \mathsf{ind}(w)$, for each $j \in [2^{r_\ell} - 1]$.

Moreover, let us define the right language of $q_j$ as

$$\mathscr{L}_{q_j} = \{ w \in \Sigma^{\ell - r_\ell} \mid i = \mathsf{ind}(w^{\mathrm{R}}),\ (i+1) \wedge 2^j \neq 0 \}.$$

Clearly, $B$ accepts $\mathsf{MAX}_\ell^{\mathrm{R}}$. Consider $B_{q_j}$, the DFA $B$ with initial state $q_j$, and let us show that $\mathsf{w}(B_{q_j}) \leq 2$.

Let $j \in [2^{r_\ell} - 1]$, $r' \in [\ell - r_\ell - 1]$, and $\mathscr{B}_{r'}$ be the set of factors of size $2^{r'}$ of the bitmap of $\mathsf{MAX}_\ell^{\mathrm{R}}$. Let $s \in \mathscr{B}_{r'}$ and $w_1 \in \Sigma^{\ell - r_\ell - r'}$ such that $s$ represents the language $w_1^{-1}\mathscr{L}_{q_j}$. From the definition of $\mathscr{L}_{q_j}$, $w_1 w_2 \in \mathscr{L}_{q_j}$ if $(i+1) \wedge 2^j \neq 0$, where $w_2 \in \Sigma^{r'}$ and $i = \mathsf{ind}(w_2^{\mathrm{R}} w_1^{\mathrm{R}})$. We will now show that the number of states on the rank $r'$ of $B_{q_j}$ is bounded by 2, by arguing that $|\mathscr{B}_{r'}| \leq 2$. Consider the following two cases:

1. $j < \ell - r_\ell - r'$: in this case, it is sufficient to check if $(i'+1) \wedge 2^j \neq 0$, where $i' = \mathsf{ind}(w_1^{\mathrm{R}})$, since $|w_1^{\mathrm{R}}| = \ell - r_\ell - r'$. Then, either $i'+1$ has its $j$-th bit equal to 1, which implies that $s = 1 \cdots 1$, or has not, implying that $s = 0 \cdots 0$, so $s \notin \mathscr{B}_{r'}$. Therefore, $|\mathscr{B}_{r'}| = 1$.
2. $j \geq \ell - r_\ell - r'$: if $w_1 \in \Sigma^{\ell - r_\ell - r'} \setminus \{b^{\ell - r_\ell - r'}\}$, the binary representation of $i'+1$, with $i' = \mathsf{ind}(w_1^{\mathrm{R}})$, requires at most $\ell - r_\ell - r'$ bits. Then, the $j$-th bit of $i$, corresponds to the $(j - \ell + r_\ell + r')$-bit of $w_2$. Hence, $u^{-1}\mathscr{L}_{q_j} = v^{-1}\mathscr{L}_{q_j}$, for all $u, v \in \Sigma^{\ell - r_\ell - r'} \setminus \{b^{\ell - r_\ell - r'}\}$. On the other hand, if $w_1 = b^{\ell - r_\ell - r'}$, it results on a different quotient, since $\ell - r_\ell - r' + 1$ bits are needed for the binary representation of $i'+1$. Therefore, $|\mathscr{B}_{r'}| = 2$.

This result implies that $\mathsf{w}(B_{q_j}) = 2$. As a consequence, the width of the ranks $r' \in [\ell - r_\ell - 1]$ of $A$ are bounded by $2^{r_\ell + 1}$, as desired. $\square$

Then, we have the following bound on the state complexity of the reversal of a language.

**Theorem 4.** *Let $L \subseteq \Sigma^\ell$, $\ell > 0$, such that $\mathsf{sc}(L) = m$. Then, $\mathsf{sc}(L^{\mathrm{R}}) \in 2^{\Theta(\sqrt{m})}$.*

*Proof.* By Corollary 1 we have that $2^{O(\sqrt{m})}$ states are sufficient for a DFA accepting $L^{\mathrm{R}}$. Now, we prove that this cost is necessary in the worst case.

Let $A$ and $B$ be the minimal DFAs for $\mathsf{MAX}_\ell$ and $\mathsf{MAX}_\ell^{\mathrm{R}}$ with set of states $Q$ and $P$, respectively. For this proof, assume that $2^{\ell - r} > 2^{2^{r-1}} - 1$, which implies that $r_\ell = r$. A similar proof follows, otherwise.

By Proposition 1, $A$ is of maximal size and its width is exactly $2^{\ell - r}$. Therefore, the number of states of $A$ is

$$\begin{aligned}
|Q| &= \sum_{i=0}^{r_\ell - 1} (2^{2^i} - 1) + \sum_{i=r_\ell}^{\ell} 2^{\ell - i} = \sum_{i=0}^{r_\ell - 1} 2^{2^i} - r_\ell + 2^{\ell - r_\ell + 1} - 1 \\
&\geq 2^{2^{r_\ell} - 1} - (r_\ell + 1) && 2^{\ell - r_\ell + 1} > 0 \\
&\geq 2^{2^{\log_2 \ell + 2} - 1} - (\log_2 \ell + 3) && 2^{2^{r_\ell} - 1} \gg r_\ell,\ r = r_\ell, \text{ and } r < \log_2 \ell + 2 \\
&\geq 2^{4\ell - 1} - (\log_2 \ell + 3) \in 2^{\Omega(\ell)}.
\end{aligned}$$

By Lemma 3, the width of $B$ is bounded by $2^{r_\ell + 1}$. Moreover, the width of each rank $i \in [r-1]$ of $B$ is also bounded by $2^{2^i} - 1$, as we have seen in Lemma 1. Then, let $r' = \min\{n \in [\ell] \mid 2^{r_\ell + 1} \leq 2^{2^n} - 1\}$. In particular, we have

$$\begin{aligned}
2^{r_\ell + 1} > 2^{2^{r' - 1}} - 1 &\implies 2^{r_\ell + 1} \geq 2^{2^{r' - 1}} \implies r_\ell + 1 \geq 2^{r' - 1} \\
&\implies \log_2 \ell + 3 \geq 2^{r' - 1} && r = r_\ell \text{ and } r < \log_2 \ell + 2 \\
&\implies r' \leq \log_2(\log_2 \ell + 3) + 1.
\end{aligned}$$

The value of $r'$ tells us how many ranks in $B$ can achieve the maximal width of $2^{r_\ell+1}$. Then, the number of states of $B$ is bounded by

$$\begin{aligned}
|P| &\leq \sum_{i=0}^{r'-1}(2^{2^i}-1)+2^{r_\ell+1}(\ell-r_\ell-r')+\sum_{i=\ell-r_\ell}^{\ell}2^{\ell-i} \\
&\leq 2^{2^{r'}}-r'+2^{r_\ell+1}(\ell-r_\ell-r'+1)-1 \\
&\leq 2^{2^{\log_2(\log_2\ell+3)+1}}+2\cdot 2^{r_\ell}(\ell+1) && r'\leq \log_2(\log_2\ell+3)+1 \\
&\leq 2^6\cdot 2^{\log_2\ell^2}+2^3\cdot 2^{\log_2\ell}(\ell+1) && r=r_\ell \text{ and } r<\log_2\ell+2 \\
&\leq 2^6\ell^2+2^3(\ell^2+\ell)=O(\ell^2).
\end{aligned}$$

Thus, given $L=\mathsf{MAX}_\ell^{\mathrm{R}}$ with $\mathsf{sc}(L)=m$, we have that $\mathsf{sc}(L^{\mathrm{R}})=2^{\Omega(\sqrt{m})}$, as desired. □

The NFA for the reversal of a language $L\subseteq\Sigma^\ell$ is given by reversing the transitions on the NFA for $L$. In fact, the nondeterministic state complexity of the reversal of a finite language coincides with the nondeterministic state complexity of the language, so no better result can be obtained for the block languages.

**Theorem 5.** *Let $L\subseteq\Sigma^\ell$, for some $\ell>0$. Then,* $\mathsf{nsc}(L^{\mathrm{R}})=\mathsf{nsc}(L)$.

*Proof.* The construction above shows that $\mathsf{nsc}(L^{\mathrm{R}})\leq\mathsf{nsc}(L)$. The following family of languages shows that it is tight. Let $L_\ell=\{a^\ell\}$, with $\ell>0$. We have both that $\mathsf{nsc}(L_\ell)=\ell+1$ and $L_\ell=L_\ell^{\mathrm{R}}$. □

## 3.2 Word Addition and Word Removal

Consider a language $L\subseteq\Sigma^\ell$, for some $\ell>0$, over an alphabet of size $k$. The operations of adding or removing a word $w\in\Sigma^\ell$ from the language, $L\setminus\{w\}$ and $L\cup\{w\}$, respectively, correspond to the not operation on the $\mathsf{ind}(w)$-th bit of $\mathsf{B}(L)$. From that observation, we can estimate the state complexity of these operations.

**Theorem 6.** *Let $L\subseteq\Sigma^\ell$ be a block language with $|\Sigma|=k$ and $\ell>0$, such that $\mathsf{sc}(L)=m$. Let $\oplus\in\{\setminus,\cup\}$, $w\in\Sigma^\ell$, and $L'=L\oplus\{w\}$. Then, $n-(\ell-1)\leq\mathsf{sc}(L')\leq m+(\ell-1)$.*

*Proof.* Let $\mathsf{B}(L)$ and $\mathsf{B}(L')$ be the bitmaps of $L$ and $L'$, respectively. Let us assume that the operation $\oplus$ results in a different language. Then, the bitmaps $\mathsf{B}(L')$ and $\mathsf{B}(L)$ differ exactly for one bit. Let $i\in[\ell]$. Then, there is exactly one $j\in[k^{\ell-i}-1]$ such that $s_j^i\neq t_j^i$, where $s_j^i$ and $t_j^i$ denote the $j$-th bitmap factor of size $k^i$ of $\mathsf{B}(L)$ and $\mathsf{B}(L')$, respectively. Also, recall $\mathscr{B}(L)_i$ (resp. $\mathscr{B}(L')_i$), the set of factors of size $k^i$ of $\mathsf{B}(L)$ (resp. $\mathsf{B}(L')$). Then, there are four possible cases:

1. $s_j^i\in\mathscr{B}(L')_i$ and $t_j^i\in\mathscr{B}(L)_i$: the two sets have the same size;
2. $s_j^i\in\mathscr{B}(L')_i$ and $t_j^i\notin\mathscr{B}(L)_i$: $\mathscr{B}(L')_i$ has one more element than $\mathscr{B}(L)_i$;
3. $s_j^i\notin\mathscr{B}(L')_i$ and $t_j^i\in\mathscr{B}(L)_i$: $\mathscr{B}(L)_i$ has one more element than $\mathscr{B}(L')_i$;
4. $s_j^i\notin\mathscr{B}(L')_i$ and $t_j^i\notin\mathscr{B}(L)_i$: the two sets have the same size.

Then, the difference on the number of states from a DFA which accepts the language $L'$ and the DFA which accepts $L$ is bounded by $\ell-1$, which is the number of ranks neither initial nor final. □

These bounds also extend to the nondeterministic state complexity, as proved in the following result.

**Theorem 7.** *Let $L \subseteq \Sigma^\ell$ be a block language with $|\Sigma| = k$ and $\ell > 0$, such that* $\mathsf{nsc}(L) = m$. *Let $\oplus \in \{\setminus, \cup\}$, $w \in \Sigma^\ell$, and $L' = L \oplus \{w\}$. Then, $m - (\ell - 1) \leq \mathsf{nsc}(L') \leq m + (\ell - 1)$.*

*Proof.* Consider the proof of Theorem 6 and its notation. If the case 2 verifies, that is, $|\mathscr{B}(L')_i| = |\mathscr{B}(L)_i| + 1$, then the cover will require at most one more segment to cover the new set. Analogously, the size of the cover for $\mathscr{B}(L')_i$ can be smaller by one than the cover for $\mathscr{B}(L)_i$, for the case 3. □

The upper bounds from Theorems 6 and 7 are reached as stated in the following theorem.

**Theorem 8.** *The bounds given in Theorems 6 and 7 are tight.*

*Proof.* Let $\Sigma = \{a, b\}$ and $\ell > 0$. Consider $L_\ell = \{a, b\}^\ell$, whose bitmap is $\mathsf{B}(L_\ell) = 1^{2^\ell}$. Let $w = a^\ell$. We have that $\mathsf{nsc}(L_\ell) = \ell + 1$ and $\mathsf{sc}(L_\ell) = 1 + \mathsf{nsc}(L) = 2 + \ell$, while $\mathsf{sc}(L_\ell \setminus \{w\}) = 2\ell + 1 = 1 + \mathsf{nsc}(L_\ell \setminus \{w\})$. In the same way it is possible to prove for word addition. □

The family of languages in the previous proof is also a witness for the upper bound of the operation $\Sigma^\ell \setminus \{w\}$.

## 3.3 Intersection

Let $L_1, L_2 \subseteq \Sigma^\ell$ be two block languages, for some $\ell > 0$ and $|\Sigma| = k$, and their respective bitmaps $\mathsf{B}(L_1)$, $\mathsf{B}(L_2)$. The bitmap of the intersection of $L_1$ and $L_2$ is given by $\mathsf{B}(L_1) \wedge \mathsf{B}(L_2)$.

Now, let $A = \langle Q \cup \{\Omega_1\}, \Sigma, \delta_1, q_0, \{q_f\}\rangle$ and $B = \langle P \cup \{\Omega_2\}, \Sigma, \delta_2, p_0, \{p_f\}\rangle$ be the minimal DFAs for $L_1$ and $L_2$, respectively. For obtaining a DFA for $L_1 \cap L_2$, one can use the standard product construction, and obtain a product automaton $C$. As shown in [6] (see Table 1) the size of $C$ is at most $mn - 3(m + n) + 12$, if $|Q| = m - 1$ and $|P| = n - 1$. This bound is the result of:

- There are no transitions to the initial state neither in $A$, $B$, nor $C$ (this saves $m - 1 + n - 1$ states);
- In $C$, all pairs of states $(q, \Omega_2)$ and $(\Omega_1, p)$, for $q \in Q$ and $p \in P$, can be merged with $(\Omega_1, \Omega_2)$ (this saves $m - 2 + n - 2$ states);
- In $C$, all pairs of states $(q, p_f)$ or $(q_f, p)$, for $q \in Q$ and $p \in P$, can be merged with $(q_f, p_f)$ or $(\Omega_1, \Omega_2)$ (if in general $q_f$ and $p_f$ are the pre-dead states, this saves $m - 3 + n - 3$ states).

However, for block languages ,pre state can be saved since a state $(q, p)$ of $C$ is both accessible from the initial state and leads to the final state if and only if $\mathsf{rank}(q) = \mathsf{rank}(p)$, for every $q \in Q$, $p \in P$.

Let $Q_i$ be the set of states in rank $i$ in $A$ and $m_i = \mathsf{w}(i) = |Q_i|$, for $i \in [\ell]$. Let $P_i$ be the set of states in rank $i$ in $B$ and $n_i = \mathsf{w}(i) = |P_i|$, for $i \in [\ell]$. Additionally, $m = 1 + \sum_{i \in [\ell]} m_i$ and $n = 1 + \sum_{i \in [\ell]} n_i$ since the dead states $\Omega_j$ do not belong to any rank. We have that:

**Lemma 9.** *Given two DFAs $A$ and $B$ for block languages $L_1$ and $L_2$, respectively, a DFA with $\sum_{i=0}^{\ell} m_i n_i + 1$ states is sufficient to recognize the intersection of $L_1$ and $L_2$, where $m_i$ and $n_i$ are the widths of rank $i$ in $A$ and $B$, respectively, for $i \in [\ell]$.*

*Proof.* Given the above considerations, the states of the DFA resulting from trimming $C$ are, in the worst-case, $\bigcup_{i \in [\ell]} Q_i \times P_i$ and a single dead state is needed. □

Let us show that this bound is tight for a fixed size of the alphabet, as opposed to the general case of finite languages where a growing alphabet is required [6]. Consider the following family of languages, defined over an alphabet $\Sigma$ of size $k$, and let $d > 0$ and $x \in \{0,1\}$:

$$L_{k,d,x} = \{\, a_0 \cdots a_{2d-1} \in \Sigma^{2d} \mid \forall i \in [d-1] : i \equiv x \pmod 2 \implies a_i = a_{2d-i} \,\}.$$

Informally, it contains the words that can be split into two halves of size $d$, where, if $x = 0$ ($x = 1$, resp.), then the symbols in even (odd, resp.) positions of the first half are equal to their symmetric position in the second half.

**Lemma 10.** *Let $k \geq 2$, $d \geq 0$, and $x \in \{0,1\}$. Also, let $A$ be the minimal DFA for $L_{k,d,x}$ over a $k$-letter alphabet $\Sigma$ and let $m_i$ be the width of $A$, for $i \in [2d]$. Then, for $i \in [d,2d]$ we have:*

$$m_i = \begin{cases} k^{\lceil \frac{2d-i}{2} \rceil}, & \textit{if } x = 0; \\ k^{\lfloor \frac{2d-i}{2} \rfloor}, & \textit{if } x = 1, \end{cases}$$

*and for $i \in [d]$ we have $m_i = m_{2d-1}$.*

*Proof.* Let us prove for $x = 0$.

1. $(\forall i \in [d,2d]) : m_i = k^{\lceil \frac{2d-i}{2} \rceil}$:
   Let $w_1, w_2 \in \Sigma^{2d-i}$ such that they differ at least in one even position. Now, let $w_3 = \sigma^{2(i-d)} w_1^{\mathrm{R}}$, for some $\sigma \in \Sigma$. It is easy to see that $w_1 w_3 \in L_{k,d,0}$ but $w_2 w_3 \notin L_{k,d,0}$, so $w_1$ and $w_2$ have different quotients, and so they have to reach different states. Therefore, the number of states on rank $i$ of $A$ is given by $k^{\lceil \frac{2d-i}{2} \rceil}$, where the exponent is the number of odd integers between $i$ and $2d-1$.
2. $(\forall i \in [d]) : m_i = m_{2d-i}$:
   Let us look at $A^{\mathrm{R}}$, the NFA for $L_{k,d}^{\mathrm{R}}$ given by reversing every transition in $A$ and swapping the initial with the final states. In fact, it is easy to see that $L_{k,d,x} = L_{k,d,x}^{\mathrm{R}}$, hence $\mathscr{L}(A) = \mathscr{L}(A^{\mathrm{R}})$. In 1, we proved that $m_j = k^{\lceil \frac{2d-j}{2} \rceil}$, for every rank $j \in [d,2d]$. The $i$-th rank in $A$ corresponds to the $(2d-i)$-th rank in $A^{\mathrm{R}}$, so that bound must be preserved.

For $x = 1$, the number of states is $k^{\lfloor \frac{2d-i}{2} \rfloor}$, where the exponent is the number of even integers between $i$ and $2d-1$, so the proof is similar. ☐

Then, we have the following result for the operational state complexity of intersection:

**Lemma 11.** *Let $A$ and $B$ be DFAs that accept $L_{k,d,0}$ and $L_{k,d,1}$ and $m_i$ and $n_i$ the widths of rank $i$ in $A$ and $B$, respectively, for $i \in [2d]$ and $d > 0$. A DFA that recognizes the language $L_{k,d,0} \cap L_{k,d,1}$ needs $\sum_{i=0}^{2d} m_i n_i + 1$ states.*

*Proof.* As stated in Lemma 10, we have $m_i = m_{2d-i} = k^{\lceil \frac{2d-i}{2} \rceil}$ and $n_i = n_{2d-i} = k^{\lfloor \frac{2d-i}{2} \rfloor}$, for $i \in [d,2d]$. Moreover, it is easy to see that

$$L_{k,d,0} \cap L_{k,d,1} = \{ww^{\mathrm{R}} \mid w \in \Sigma^d\},$$

that is, the set of palindromes of even length. A minimal DFA $C$ for this language with set of states $S = S_0 \cup \ldots \cup S_\ell$ must first be able to remember the entire first half of the word, therefore, $|S_i| = k \cdot |S_{i+1}| = k^{2d-i}$ for $i \in [d, 2d-1]$. For the second half, it must check for the repetition of the first, then, $|S_i| = |S_{2d-i}|$, for $i \in [d]$. In fact,

$$|S_i| = m_i n_i = k^{\lceil \frac{2d-i}{2} \rceil} k^{\lfloor \frac{2d-i}{2} \rfloor} = k^{2d-i},$$

as desired. ☐

From Lemmas 9 and 11 we have:

**Theorem 12.** *Given two DFAs $A, B$ for block languages $L_1, L_2 \subseteq \Sigma^\ell$, for $\ell > 0$, $\sum_{i=0}^{\ell} m_i n_i + 1$ states are necessary and sufficient in the worst-case for a DFA that accepts the intersection of $L_1$ and $L_2$, where $m_i$ and $n_i$ are the widths of rank $i$ in $A$ and $B$, respectively, for $i \in [\ell]$.*

For the nondeterministic state complexity, the bounds are the same except that the dead state is not considered. In fact, the family witness languages for the tightness of deterministic state complexity is also a witness for the nondeterministic one.

**Theorem 13.** *Let $A = \langle Q, \Sigma, \delta_1, q_0, \{q_f\}\rangle$ and $B = \langle P, \Sigma, \delta_2, p_0, \{p_f\}\rangle$ be minimal NFAs for two block languages $L_1, L_2 \subseteq \Sigma^\ell$, respectively, for some $\ell > 0$, and such that $|Q| = m$ and $|P| = n$. Let $Q_i$ be the set of states in rank $i$ in $A$ and $m_i = \mathsf{w}(i) = |Q_i|$, for $i \in [\ell]$. Let $P_i$ be the set of states in rank $i$ in $B$ and $n_i = \mathsf{w}(i) = |P_i|$, for $i \in [\ell]$. Additionally, $m = \sum_{i\in[\ell]} m_i$ and $n = \sum_{i\in[\ell]} n_i$.*

*Then, an NFA with $\sum_{i=0}^{\ell} m_i n_i$ states is sufficient to recognize the intersection of both languages and the bound is tight for $k > 1$.*

*Proof.* The fact that $\sum_{i=0}^{\ell} m_i n_i$ states are sufficient follows from the previous discussions. Moreover, this number of states is necessary, as can be noticed by considering the languages $L_{k,d,x}$ given above. Recall the language $L_{k,d,x}$, for some $k > 1$, $d > 0$ and $x \in \{0,1\}$. In fact, it is easy to see that $\mathsf{sc}(L_{k,d,x}) - 1 = \mathsf{nsc}(L_{k,d,x})$, since the NFA for $L_{k,d,x}$ must also be able to remember the same information as the DFA. Then, if $A$ (resp. $B$) is a minimal NFA that recognizes the language $L_{k,d,0}$ (resp. $L_{k,d,1}$), an NFA that recognizes the intersection of both needs exactly $\sum_{i=0}^{\ell} n_i m_i$ states. □

## 3.4 Union

Let $L_1, L_2 \subseteq \Sigma^\ell$ be two block languages, for some $\ell > 0$ and $|\Sigma| = k$, and their respective bitmaps $\mathsf{B}(L_1), \mathsf{B}(L_2)$. The bitmap of the union of $L_1$ and $L_2$ is $\mathsf{B}(L_1) \vee \mathsf{B}(L_2)$.

Let $A = \langle Q \cup \{\Omega_1\}, \Sigma, \delta_1, q_0, \{q_f\}\rangle$ and $B = \langle P \cup \{\Omega_2\}, \Sigma, \delta_2, p_0, \{p_f\}\rangle$ be the minimal DFAs for $L_1$ and $L_2$, respectively, with $|Q| = m$ and $|P| = n$. Again, let $C$ be the product DFA of $A$ and $B$. Because $L_1, L_2$ are finite we know that $m + n$ states can be saved: $m + n - 2$ because the initial states are non returning and 2 more because the final states $(q_f, \Omega_2)$, $(\Omega_1, p_f)$, and $(q_f, p_f)$ can be merged into a single final state. However, again, one only needs to consider pairs of states $(q, p)$ such that $\mathsf{rank}(q) = \mathsf{rank}(p)$, for $q \in Q, p \in P$. Let $Q_i$ be the set of states in rank $i$ in $A$ and $m_i = \mathsf{w}(i) = |Q_i|$, for $i \in [\ell]$. Let $P_i$ be the set of states in rank $i$ in $B$ and $n_i = \mathsf{w}(i) = |P_i|$, for $i \in [\ell]$. Additionally, $m = 1 + \sum_{i\in[\ell]} m_i$ and $n = 1 + \sum_{i\in[\ell]} n_i$ since the dead states $\Omega_j$ do not belong to any rank. We have that

**Lemma 14.** *Given two DFAs $A$ and $B$ for block languages $L_1$ and $L_2$, respectively, a DFA with*

$$\sum_{i=1}^{\ell-1} (m_i n_i + m_i + n_i) + 3$$

*states is sufficient to recognize the union of $L_1$ and $L_2$, where $m_i$ and $n_i$ are the widths of rank $i$ in $A$ and $B$, respectively, for $i \in [\ell]$.*

*Proof.* Let $C$ be the product automaton from $A$ and $B$. As mentioned above, the final states $(q_f, \Omega_2)$, $(\Omega_1, p_f)$ can be merged with $(q_f, p_f)$, and a state $(p, q)$ is only accessible from the initial state if $\mathsf{rank}(p) = \mathsf{rank}(q)$. Therefore, the DFA resulting from trimming $C$ has a single initial state, a final state and a dead state, and also the states $(Q_i \times P_i) \cup (Q_i \times \{\Omega_2\}) \cup (\{\Omega_1\} \times P_i)$, at each rank $i \in [1, \ell - 1]$. Thus, the sufficient number of states follows. □

In fact, the bound is tight for an alphabet with size at least 3.

**Lemma 15.** *Given two DFAs $A$ and $B$ for block languages $L_1$ and $L_2$ over $\Sigma^\ell$, respectively, a DFA with $\sum_{i=1}^{\ell-1}(m_in_i+m_i+n_i)+3$ states is necessary to recognize the union of $L_1$ and $L_2$, where $m_i$ and $n_i$ are the widths of rank $i$ in $A$ and $B$, respectively, for $i \in [\ell]$ and $|\Sigma| > 2$.*

*Proof.* Since $A$ and $B$ are deterministic, $n_{\ell-1}$ and $m_{\ell-1}$, the number of states at rank $\ell-1$ of $A$ and $B$, respectively, are bounded by $k$ and not equal to 0. Analogously, the width of the rank $\ell-1$ of the DFA for the union of $\mathscr{L}(A)$ and $\mathscr{L}(B)$ is also at most $k$. When $k=2$, it is easy to see that the inequality $0 < n_{\ell-1}m_{\ell-1}+n_{\ell-1}+m_{\ell-1} \leq k$ has no solutions.

Now, consider the languages $L_{1,\ell} = \{a,c\}^\ell$ and $L_{2,\ell} = \{b,c\}^\ell$, and let $A$ and $B$ be the DFAs that recognize them, respectively, for some $\ell$ and $\Sigma = \{a,b,c\}$. We have that $\mathsf{sc}(L_{1,\ell}) = \mathsf{sc}(L_{2,\ell}) = \ell+2$ and $n_i = m_i = 1$, for every $i \in [\ell]$. The minimal DFA that recognizes the language $L_{1,\ell} \cup L_{2,\ell}$ requires 3 states at each rank $i \in [1,\ell-1]$: one state when some $a$ has already been read, so the word is in $L_{1,\ell}$; one state when some $b$ has already been read, so the word is in $L_{2,\ell}$; and one state for when only $c$'s have been read, so the DFA still does not know to what particular language it belongs. Then, $\mathsf{sc}(L_{1,\ell} \cup L_{2,\ell}) = \sum_{i=1}^{\ell-1}(n_im_i+n_i+m_i)+3 = 3\ell$. ☐

From Lemmas 14 and 15 we have:

**Theorem 16.** *Given two DFAs $A,B$ for block languages $L_1,L_2 \subseteq \Sigma^\ell$, for $\ell > 0$, $\sum_{i=1}^{\ell-1}(m_in_i+m_i+n_i)+3$ states are sufficient and necessary, if $\Sigma > 2$, in the worst-case for a DFA that accepts the union of $L_1$ and $L_2$, where $m_i$ and $n_i$ are the widths of rank $i$ in $A$ and $B$, respectively, for $i \in [\ell]$.*

For the nondeterministic state complexity, the upper bound is the same as for finite languages over the same alphabet size.

**Theorem 17.** *Let $L_1,L_2 \subseteq \Sigma^\ell$ with $\ell > 0$ and $|\Sigma| = k$, such that $\mathsf{nsc}(L_1) = n$ and $\mathsf{nsc}(L_2) = m$. Then, $\mathsf{nsc}(L_1 \cup L_2) \leq n+m-2$, and this bound is reached.*

*Proof.* Let $L_{1,\ell} = \{a^\ell\}$ and $L_{2,\ell} = \{b^\ell\}$, for some $\ell > 0$ and $\Sigma = \{a,b\}$. We have that $\mathsf{nsc}(L_{1,\ell}) = \mathsf{nsc}(L_{2,\ell}) = \ell+1$ and $\mathsf{nsc}(L_{1,\ell} \cup L_{2,\ell}) = 2\ell$. ☐

## 3.5 Concatenation

Consider two languages $L_1 \subseteq \Sigma^{\ell_1}$ and $L_2 \subseteq \Sigma^{\ell_2}$, for some $\ell_1,\ell_2 > 0$ and $|\Sigma| = k$, with bitmaps $\mathsf{B}(L_1)$ and $\mathsf{B}(L_2)$, respectively. The bitmap for the language $L_1L_2$ is given by replacing each 1 in $\mathsf{B}(L_1)$ by $\mathsf{B}(L_2)$ and each 0 by $0^{k^{\ell_2}}$. This ensures that each word of $L_1L_2$ is obtained by concatenating a word of $L_1$ with a word of $L_2$ and for each word obtained in such way the correspondent bit in $\mathsf{B}(L_1L_2)$ is set to 1.

The deterministic state complexity of the concatenation for block languages coincides with the one for the finite languages when the first operand, $L_1$, has a single final state in its minimal DFA. Therefore, we have the following exact upper bound:

**Theorem 18.** *Let $L_1 \subseteq \Sigma^{\ell_1}$ and $L_2 \subseteq \Sigma^{\ell_2}$, for some $\ell_1,\ell_2 > 0$, be two block languages over a $k$-letter alphabet, where $\mathsf{sc}(L_1) = m$ and $\mathsf{sc}(L_2) = n$. Then, $\mathsf{sc}(L_1L_2) = m+n-2$.*

*Proof.* Let $A$ and $B$ be the minimal DFAs for $L_1$ and $L_2$, respectively. Also, let $C$ be the minimal DFA for $L_1L_2$. Considering the bitmaps for these languages, the width of the rank $i$ of $C$ is $|\mathscr{B}(L_2)|_i$, if $i \in [\ell_2]$, or is $|\mathscr{B}(L_1)|_{i-\ell_2}$, if $i \in [\ell_2+1,\ell_1+\ell_2]$. Then, $C$ saves 2 states by reusing the final state of $A$ for the initial state of $B$ (alternatively, reusing the initial state of $B$ for the final state of $A$) and also by eliminating one of the dead states. ☐

For the nondeterministic state complexity, the same result is expected, coinciding with the state complexity for the finite languages.

**Theorem 19.** *Let $L_1 \subseteq \Sigma^{\ell_1}$ and $L_2 \subseteq \Sigma^{\ell_2}$, for some $\ell_1, \ell_2 > 0$, be two block languages over a k-letter alphabet, where $\mathsf{nsc}(L_1) = m$ and $\mathsf{nsc}(L_2) = n$. Then, $\mathsf{nsc}(L_1L_2) = m+n-1$.*

In fact, any two languages $L_1 \subseteq \Sigma^{\ell_1}$ and $L_2 \subseteq \Sigma^{\ell_2}$ result in a family of witness languages. That is due to the fact that this operation preserves the ranks of the DFAs of the operands.

**Example 4.** *Let $L_{1,\ell_1} = \{a^{\ell_1}\}$ and $L_{2,\ell_2} = \{a^{\ell_2}\}$, for $\ell_1, \ell_2 > 0$ and $\Sigma = \{a\}$. We have that $\mathsf{sc}(L_{1,\ell_1}) = \ell_1 + 2$, $\mathsf{sc}(L_2) = \ell_2 + 2$, and $\mathsf{sc}(L_{1,\ell_1}L_{2,\ell_2}) = \ell_1 + \ell_2 + 2$. We also have $\mathsf{nsc}(L_{1,\ell_1}) = \ell_1 + 1$, $\mathsf{nsc}(L_{2,\ell_2}) = \ell_2 + 1$, and $\mathsf{nsc}(L_{1,\ell_1}L_{2,\ell_2}) = \ell_1 + \ell_2 + 1$.*

## 3.6 Block Complement

Consider a language $L \subseteq \Sigma^\ell$, for some $\ell > 0$ and alphabet of size $k > 0$, and let $\mathsf{B}$ be its bitmap. Now, given a block language $\Sigma^\ell$, we consider block complement language, namely $\Sigma^\ell \setminus L$, also denoted by $\overline{L}^\ell$.

Then, the bitmap of the language $\Sigma^\ell \setminus L$, namely $\overline{\mathsf{B}}$, is given by flipping every bit of $\mathsf{B}$.

**Theorem 20.** *Let $L \subseteq \Sigma^\ell$, with $\ell > 0$, be a block language with $|\Sigma| = k$, such that $\mathsf{sc}(L) = m$. Then, $m - (\ell - 1) \leq \mathsf{sc}(\Sigma^\ell \setminus L) \leq m + (\ell - 1)$.*

*Proof.* The number of states on a rank $i \in [\ell]$ of the minimal DFA for $\Sigma^\ell \setminus L$ is given by the cardinality of $\overline{\mathscr{B}}_i$, the set of the non-null factors of length $k^i$ on the bitmap $\overline{\mathsf{B}}$. If, for some $j \in [k^{\ell-i} - 1]$, we have that $s^i_j = 0 \cdots 0$, which by definition implies that $s^i_j \notin \mathscr{B}_i$, then $\overline{s^i_j} = 1 \cdots 1$ and so $\overline{s^i_j} \in \overline{\mathscr{B}_i}$. Moreover, the complement may also occur. Therefore, $\big| |\mathscr{B}_i| - |\overline{\mathscr{B}_i}| \big| \leq 1$.

Let $L_\ell = \{a^\ell\}$, for $\ell > 0$. As we previously saw on Theorem 8, $\mathsf{sc}(L) = \ell + 1$, and $\mathsf{sc}(\Sigma^\ell \setminus L) = 2\ell$. $\square$

For the nondeterministic state complexity of the block complement operation, we have that the bound meets the one of the complement from the gmeneral case for finite languages considering the determinization cost of block languages. Also, this bound is asymptotically tight for alphabets of size at least 2.

**Lemma 21.** *Let $L \subseteq \Sigma^\ell$ be a block language with $|\Sigma| = k$, such that $L$ is accepted by an m-state NFA. Then, $2^{O(\sqrt{m})}$ states are sufficient for an NFA for $\Sigma^\ell \setminus L$.*

*Proof.* Let $A$ be a NFA for $L$ with $m$ states. The minimal DFA $B$ for $L$ will have at most $2^{O(\sqrt{m})}$ states [9]. Furthermore, the minimal DFA $C$ for $\Sigma^\ell \setminus L$ will have at most $\ell + 1$ more states then $B$, as shown in Theorem 20. The nondeterministic state complexity is trivially bounded by the deterministic state complexity, so the sufficient number of states follows. $\square$

Consider the following family presented by Karhumäki and Okhotin [9]:

$$L_{k,d} = \{ w_0 \cdots w_{2d-1} \mid \exists i \in [d-1] : w_i = w_{i+d} \in \Sigma \setminus \{\sigma_{k-1}\} \}$$

defined over a $k$-ary alphabet $\Sigma = \{\sigma_0, \ldots, \sigma_{k-1}\}$. Informally, this language contains words that can be split into two halves of size $d$, such that there is at least one position in the first half that matches its counterpart in the second one, and it is different than the "*prohibited symbol*" $\sigma_{k-1}$.

**Proposition 2** ( [9]). *For each $k \geq 2$ and $d \geq 2$, the language $L_{k,d}$ is recognized by an NFA with $(k-1)d^2 + 2d$ states.*

**Lemma 22.** *For each $k \geq 2$ and $d \geq 2$, the language $\overline{L}_{k,d}^{2d}$ defined over a k-letter alphabet $\Sigma$ requires at least $k^d$ states on the d-th rank.*

*Proof.* First, notice that $\overline{L}_{k,d}^{2d}$ is the block complement of the language $\overline{L}_{k,d}$ defined above, formally

$$\overline{L}_{k,d}^{2d} = \{ w_0 \cdots w_{2d-1} \mid \forall i \in [d-1] : w_i \neq w_{i+d} \text{ or } w_i = \sigma_{k-1} \}.$$

Let $w_1$ and $w_2$ be two words in $\Sigma^d$ such that $a$ and $b$ are the $i$-th symbols of $w_1$ and $w_2$, respectively, with $a \neq b$ and $i \in [d]$. If $a = \sigma_{k-1}$ then, with $w_3 = \sigma_{k-1}^{i-1} b \sigma_{k-1}^{d-i}$, we have $w_1 w_3 \in \overline{L}_{k,d}^{2d}$ but $w_2 w_3 \notin \overline{L}_{k,d}^{2d}$. If $b = \sigma_{k-1}$ then, with $w_3 = \sigma_{k-1}^{i-1} a \sigma_{k-1}^{d-i}$, we have $w_1 w_3 \notin \overline{L}_{k,d}^{2d}$ but $w_2 w_3 \in \overline{L}_{k,d}^{2d}$. As a consequence, $w_1^{-1}\overline{L}_{k,d}^{2d} \neq w_2^{-1}\overline{L}_{k,d}^{2d}$. Therefore, one state in rank $d$ is needed for each word in $\Sigma^d$.

□

With these results, it is possible to determine that the nondeterministic state complexity for the complement operation given in Lemma 21 is tight.

**Theorem 23.** *Let $m \geq 2$ and $\Sigma$ an alphabet of size $k \geq 2$. Then, there exists a language $L \subseteq \Sigma^\ell$, for some $\ell > 0$, such that* $\mathsf{nsc}(L) = m$ *and* $\mathsf{nsc}(\Sigma^\ell \setminus L) = 2^{\Omega(\sqrt{m})}$.

*Proof.* Consider $d$ as the largest integer for which $(k-1)d^2 + 2d \leq m$. Following the work in [9], we have that

$$d = \left\lfloor \sqrt{\frac{m}{k-1} + \frac{1}{(k-1)^2}} - \frac{1}{k-1} \right\rfloor.$$

Then, $L_{k,d}$ is a language recognized by an $m$-state NFA, while every NFA for $\overline{L}_{k,d}^{\ell}$ requires, by Lemma 22, at least

$$k^d = k^{\left\lfloor \sqrt{\frac{m}{k-1} + \frac{1}{(k-1)^2}} - \frac{1}{k-1} \right\rfloor} \geq k^{\sqrt{\frac{m}{k-1}} - 2} = 2^{\Omega(\sqrt{m})}$$

states, as required. □

## 3.7 Kleene Star and Plus

Let $L \subseteq \Sigma^\ell$, for some $\ell > 0$ and $\mathsf{B}$ its bitmap. From $\mathsf{B}$ one can obtain the minimal DFA for $L$, namely $A = \langle Q, \Sigma, \delta_0, q_0, \{q_f\} \rangle$.

A DFA $B = \langle Q \setminus \{q_f\}, \Sigma, \delta_1, q_0, \{q_0\} \rangle$ recognizes the language $L^\star$ if $\delta_1(q,\sigma) = q_0$, for all $q \in Q$ such that $\mathsf{rank}(q) = 1$ and $\sigma \in \Sigma$, and $\delta_1(q,\sigma) = \delta(q,\sigma)$, for the remaining pairs $(q,\sigma) \in Q \times \Sigma$. That is, the DFA for $L^\star$ is given by substituting all the transitions with final state as the target state to transitions to the initial state. The same applies for the NFA for $L^\star$, as the following theorem states.

**Theorem 24.** *Let $L \subseteq \Sigma^\ell$, with $\ell > 0$, be a block language with* $\mathsf{sc}(L) = n$ *and* $\mathsf{nsc}(L) = m$. *Then,* $\mathsf{sc}(L^\star) = n - 1$ *and* $\mathsf{nsc}(L^\star) = m - 1$.

Moreover, a DFA $C = \langle Q, \Sigma, \delta_2, q_0, \{q_f\} \rangle$ recognizes the language $L^+$ if $\delta_2(q_f, \sigma) = \delta_0(q_0, \sigma)$, for $\sigma \in \Sigma$. Again, the NFA for $L^+$ is given by applying the same changes to the minimal NFA for $L$. And the witness languages coincide with the ones for finite languages, namely $L_\ell = \{a^\ell\}$ , for $\ell > 0$.

**Theorem 25.** *Let $L \subseteq \Sigma^\ell$, with $\ell > 0$. Then,* $\mathsf{sc}(L^+) = \mathsf{sc}(L)$ *and* $\mathsf{nsc}(L^+) = \mathsf{nsc}(L)$.

Table 2: Upper bounds of the state complexity for block languages of words of length $\ell$.

| Block Languages | | | | |
|---|---|---|---|---|
| | sc | $\|\Sigma\|$ | nsc | $\|\Sigma\|$ |
| $L_1 \cup L_2$ | $\sum_{i=1}^{\ell-1}(m_i n_i + m_i + n_i) + 3$ | 3 | $m+n-2$ | 2 |
| $L_1 \cap L_2$ | $\sum_{i=0}^{\ell} m_i n_i + 1$ | 2 | $\sum_{i=0}^{\ell} m_i n_i$ | 2 |
| $L_1 L_2$ | $m+n-2$ | 1 | $m+n-1$ | 1 |
| $\Sigma^\ell \setminus L$ | $m+\ell-1$ | 2 | $O(2^{\sqrt{m}})$ | 2 |
| $L \cup \{w\}$ | $m+\ell-1$ | 2 | $m+\ell-1$ | 2 |
| $L \setminus \{w\}$ | $m+\ell-1$ | 2 | $m+\ell-1$ | 2 |
| $L^*$ | $m-1$ | 1 | $m-1$ | 1 |
| $L^+$ | $m$ | 1 | $m$ | 1 |
| $L^{\mathrm{R}}$ | $2^{\Theta(\sqrt{m})}$ | 2 | $m$ | 1 |

## 4 Conclusions

The complexities obtained for operations on block languages are summarized in Table 2. One can compare these results with the ones for finite languages summarized in Table 1. For the deterministic state complexity, the bounds for Boolean operations on block languages are given using the rank widths and are smaller than the ones for finite languages. It would be interesting to express them as a function of the number of states of the operands (as it is usually done). Moreover, those bounds could be obtained from a direct construction of the minimal DFA for the resulting language considering the bitmaps of the operands. Note that bitwise Boolean operations can be performed to obtain the bitmap factors (i.e., states) in each rank of the resulting DFA. This study will be interesting to pursue in future work. For concatenation and Kleene star the bounds correspond to special cases of the ones for finite languages. Finally, for reversal the results are analogous to the ones for finite languages, but here considering the bounds known for the determinization of block languages. The results for nondeterministic state complexity meet the values known for finite languages except for intersection and the specific operations for block languages (block complement, word addition, and word removal).

## References

[1] Marco Almeida, Nelma Moreira & Rogério Reis (2008): *Exact generation of minimal acyclic deterministic finite automata*. *Int. J. Found. Comput. S.* 19(4), pp. 751–765, doi:10.1142/S0129054108005930.

[2] Cezar Câmpeanu, Karel Culik II, Kai Salomaa & Sheng Yu (2001): *State Complexity of Basic Operations on Finite Languages*. In Oliver Boldt & Helmut Jürgensen, editors: *4th WIA'99*, *LNCS* 2214, Springer-Verlag, pp. 60–70, doi:10.1007/3-540-45526-4_6.

[3] Cezar Câmpeanu & Wing Hong Ho (2004): *The Maximum State Complexity for Finite Languages*. *J. Autom. Lang. Comb.* 9(2-3), pp. 189–202.

[4] Guilherme Duarte, Nelma Moreira, Luca Prigioniero & Rogério Reis (2024): *Block Languages and their Bitmap Representations*. Submitted.

[5] Yuan Gao, Nelma Moreira, Rogério Reis & Sheng Yu (2017): *A Survey on Operational State Complexity*. *Journal of Automata, Languages and Combinatorics* 21(4), pp. 251–310.

[6] Yo-Sub Han & Kai Salomaa (2008): *State Complexity of Union and Intersection of Finite Languages*. Int. J. Found. Comput. Sci. 19(3), pp. 581–595, doi:10.1142/S0129054108005838.

[7] Markus Holzer & Martin Kutrib (2003): *State Complexity of Basic Operations on Nondeterministic Finite Automata*. In Jean-Marc Champarnaud & Denis Maurel, editors: 7th CIAA 2002, LNCS 2608, Springer-Verlag, pp. 148–157, doi:10.1007/3-540-44977-9_14.

[8] Juhani Karhumäki & Jarkko Kari (2021): *Finite automata, image manipulation, and automatic real functions*. In Jean-Éric Pin, editor: Handbook of Automata Theory, European Mathematical Society, pp. 1105–1143, doi:10.4171/AUTOMATA-2/8.

[9] Juhani Karhumäki & Alexander Okhotin (2014): *On the Determinization Blowup for Finite Automata Recognizing Equal-Length Languages*. In R. Freivalds C. S. Calude & K. Iwama, editors: Computing with New Resources - Essays Dedicated to Jozef Gruska, LNCS 8808, Springer, pp. 71–82, doi:10.1007/978-3-319-13350-8_6.

[10] Stavros Konstantinidis, Nelma Moreira & Rogério Reis (2018): *Randomized Generation Of Error Control Codes With Automata And Transducers*. RAIRO 52, pp. 169–184.

[11] Diaconis Persi, Graham R. L. & Kantor William.M. (1983): *The mathematics of perfect shuffles*. Advances in Applied Mathematics 4, pp. 175–196, doi:10.1016/0196-8858(83)90009-X.

[12] Dominique Revuz (1992): *Minimisation of acyclic deterministic automata in linear time*. Theoret. Comput. Sci. 92(1), pp. 181–189, doi:10.1016/0304-3975(92)90142-3.

[13] Kai Salomaa & Sheng Yu (1997): *NFA to DFA Transformation for Finite Languages over Arbitrary Alphabets*. J. Autom. Lang. Comb. 2(3), pp. 177–186.

[14] Sheng Yu, Qingyu Zhuang & Kai Salomaa (1994): *The State Complexities of Some Basic Operations on Regular Languages*. Theor. Comput. Sci. 125(2), pp. 315–328, doi:10.1016/0304-3975(92)00011-F.

# Winning Strategies for the Synchronization Game on Subclasses of Finite Automata[*]

Henning Fernau Carolina Haase Stefan Hoffmann
Universität Trier, Fachbereich IV, Informatikwissenschaften, Trier, Germany hoffmanns.tcs@gmail.com
fernau@uni-trier.de haasec@uni-trier.de

Mikhail Volkov
Institute of Natural Sciences and Mathematics, Ural Federal University, Ekaterinburg, Russia
m.v.volkov@urfu.ru

We exhibit a winning strategy for Synchronizer in the synchronization game on every synchronizing automaton in whose transition monoid the regular $\mathfrak{D}$-classes form subsemigroups.

## 1 Introduction

A complete deterministic finite automaton (DFA) is a pair $(Q,\Sigma)$ of two finite sets equipped with a map $Q\times\Sigma\to Q$ whose image at $(q,a)\in Q\times\Sigma$ is denoted by $q{\cdot}a$. We call $Q$ the *state set* and $\Sigma$ the *input alphabet*. Elements of $Q$ and $\Sigma$ are referred to as *states* and, respectively, *letters*, and for a state $q\in Q$ and a letter $a\in\Sigma$, we refer to $q{\cdot}a$ as the result of the *action of a at* $q\in Q$. The action of letters in $\Sigma$ naturally extends to the action of words over $\Sigma$: if $w=a_1a_2\cdots a_n$ with $a_1,a_2,\dots,a_n\in\Sigma$, then $q{\cdot}w := (\dots((q{\cdot}a_1){\cdot}a_2)\dots){\cdot}a_n$.

A DFA $(Q,\Sigma)$ is called *synchronizing* if there exists a word $w$ over $\Sigma$ whose action brings the DFA to one particular state no matter at which state $w$ is applied: $q{\cdot}w = q'{\cdot}w$ for all $q,q'\in Q$. Any word $w$ with this property is said to be a *reset* word for the automaton.

Synchronizing automata serve as transparent and natural models of error-resistant systems in many applications (coding theory, robotics, testing of reactive systems) and reveal interesting connections with symbolic dynamics, substitution systems, and other parts of mathematics. We refer the reader to chapter [12] of the 'Handbook of Automata Theory' and survey [19] for an introduction to the area and an overview of its state-of-the-art.

The fourth-named author initiated viewing synchronizing automata through the lens of game theory; the motivation for this came from a game-theoretical approach to software testing suggested in [4]. In a synchronization game on a DFA $\mathscr{A}$, two players, Alice (Synchronizer) and Bob (Desynchronizer), take turns choosing letters from the input alphabet of $\mathscr{A}$. Alice who wants to synchronize $\mathscr{A}$ wins when the sequence of chosen letters forms a reset word. Bob aims to prevent synchronization or, if synchronization is unavoidable, to delay it as long as possible. Provided that both players play optimally, the outcome of such a game depends on the automaton only. This raises the problem of classifying synchronizing automata into those on which Alice and, respectively, Bob have a winning strategy. DFAs on which Alice can ensure win are of interest because they are more amenable to synchronization, in a sense. For brevity, we call such DFAs *A-automata*.

[*]Mikhail Volkov was supported by the Ministry of Science and Higher Education of the Russian Federation, project FEUZ-2023-2022.

A few initial results on synchronization games were obtained in [8]. In particular, [8, Theorem 4] provides an algorithm that, given a DFA $\mathscr{A}$ with $n$ states and $k$ input letters, decides who has a winning strategy in the synchronization game on $\mathscr{A}$ in $O(n^2k)$ time. Thus, for any individual DFA, one can determine whether it is an A-automaton. Here, however, we are interested in general conditions ensuring that all synchronizing DFAs of a certain type are A-automata. One such condition was mentioned in [8]: Alice always wins on definite DFAs introduced in [14].

In [7], the first three authors of the present note showed that within two further families of automata considered in the literature—weakly acyclic DFAs and commutative DFAs—every synchronizing DFA is an A-automaton. Here we continue this line of research by designing a winning strategy for Alice that applies to synchronizing automata from yet another family of DFAs. Automata in this family are distinguished by a structure feature of their transition monoids: the regular $\mathfrak{D}$-classes in these monoids form subsemigroups. The set **DS** of all finite semigroups with regular $\mathfrak{D}$-classes being subsemigroups plays a distinguished role in the algebraic theory of regular languages; see [1, Chapter 8]. Therefore, DFAs with transition monoids in **DS** often show up in the literature; see, e.g., [2, 3]. As they seem to have no specific name so far, we coin them DS-automata. Thus, our main result says that Alice can win the synchronization game on every synchronizing DS-automaton. Since the family of DS-automata is extensive and encompasses all the families above (definite, weakly acyclic, and commutative DFAs), this provides a vast generalization of the mentioned results from [8, 7].

Our approach is algebraic as it exploits the structural properties of transition monoids. We have collected all necessary prerequisites from semigroup theory in Section 2 to make the note self-contained, to a reasonable extent. Section 3 presents Alice's winning strategy in synchronization games on synchronizing DS-automata. In Section 4, we relate our result to previously found facts about winning strategies for synchronization games and state two open questions.

## 2 Preliminaries

### 2.1 Transition Monoids and Synchronization

For a DFA $\mathscr{A} = (Q,\Sigma)$, the map $\tau_a \colon Q \to Q$ defined by the rule $q \mapsto q{\cdot}a$ is a transformation on the set $Q$.

**Definition 1.** The *transition monoid* of a DFA $\mathscr{A} = (Q,\Sigma)$ is the submonoid of the monoid of all transformations on the set $Q$ generated by the set $\{\tau_a \mid a \in \Sigma\}$.

We denote the transition monoid of a DFA $\mathscr{A} = (Q,\Sigma)$ by $T(\mathscr{A})$. It is easy to see that any product $\tau_{a_1}\tau_{a_2}\cdots\tau_{a_n}$ is nothing but the transformation $\tau_w$ defined by the rule $q \mapsto q{\cdot}w$ where $w$ stands for the word $a_1a_2\cdots a_n$. Thus, the transition monoid $T(\mathscr{A})$ can alternatively be defined as the monoid of all transformations on the set $Q$ caused by the action of words over $\Sigma$.

If $\mathscr{A} = (Q,\Sigma)$ is a synchronizing DFA and $w$ is a reset word for $\mathscr{A}$, then the transformation $\tau_w$ is a constant map on $Q$, that is, $Q\tau_w = \{q\}$ for a certain $q \in Q$. Thus, the transition monoid of a synchronizing automaton always contains a constant transformation. Conversely, if $\zeta \in T(\mathscr{A})$ is a constant transformation, then any word $w$ with $\tau_w = \zeta$ is a reset word for $\mathscr{A}$ and so $\mathscr{A}$ is synchronizing. We see that synchronization is actually a property of the transition monoid of an automaton rather than the automaton itself: for DFAs $\mathscr{A} = (Q,\Sigma)$ and $\mathscr{A}' = (Q,\Sigma')$ with the same state set but different input alphabets, the equality $T(\mathscr{A}) = T(\mathscr{A}')$ guarantees that $\mathscr{A}'$ is synchronizing if and only if so is $\mathscr{A}$.

We can say a bit more about transition monoids of synchronizing automata, but for this, we first need to recall some concepts of semigroup theory. A nonempty subset $I$ of a semigroup $S$ is called an *ideal* in $S$ if, for all $s \in S$ and $i \in I$, both products $si$ and $is$ lie in $I$. If $I$ and $J$ are two ideals in $S$, their intersection

$I \cap J$ contains any product $ij$ with $i \in I,\ j \in J$. So $I \cap J$ is nonempty, and then it is easy to verify that $I \cap J$ is again an ideal in $S$. Therefore, each finite semigroup $S$ has a least ideal called the *kernel* of $S$ and denoted $\operatorname{Ker} S$.

The following observation is folklore but we provide a proof for the sake of completeness.

**Lemma 1.** *A DFA is synchronizing if and only if the kernel of its transition monoid consists of constant transformations.*

*Proof.* The 'if' part readily follows from the already mentioned fact that if the transition monoid of a DFA contains a constant transformation, then the DFA is synchronizing.

For the 'only if' part, let $\mathscr{A}$ be a synchronizing DFA; then the set $C$ of all constant transformations in the transition monoid $T(\mathscr{A})$ is nonempty. For any $\tau \in T(\mathscr{A})$ and any $\zeta \in C$, we have

$$\tau\zeta = \zeta. \tag{1}$$

Equality (1) implies that the set $C$ is contained in every ideal of the monoid $T(\mathscr{A})$, in particular, in its kernel $\operatorname{Ker} T(\mathscr{A})$. Further, for every $\tau \in T(\mathscr{A})$ and every $\zeta \in C$, the product $\zeta\tau$ is a constant transformation: if $Q\zeta = \{q\}$, then $Q\zeta\tau = \{q\tau\}$. Together with (1), this observation implies that $C$ forms an ideal in $T(\mathscr{A})$. Since $\operatorname{Ker} T(\mathscr{A})$ contains $C$, we have $\operatorname{Ker} T(\mathscr{A}) = C$ by the definition of the kernel. □

## 2.2 Structure of Semigroups in DS

Green [9] defined five important relations on every semigroup $S$, collectively referred to as *Green's relations*, of which we need the following three:

$a \mathfrak{R} b \iff$ either $a = b$ or $a = bs$ and $b = at$ for some $s, t \in S$;

$a \mathfrak{L} b \iff$ either $a = b$ or $a = sb$ and $b = ta$ for some $s, t \in S$;

$a \mathfrak{D} b \iff a \mathfrak{R} c$ and $c \mathfrak{L} b$ for some $c \in S$.

The relations $\mathfrak{R}$ and $\mathfrak{L}$ are obviously equivalencies. The definition of $\mathfrak{D}$ means that $\mathfrak{D}$ is the product of $\mathfrak{R}$ and $\mathfrak{L}$ as binary relations. As observed in [9], $\mathfrak{D}$ is also the product of $\mathfrak{L}$ and $\mathfrak{R}$, and this implies that $\mathfrak{D}$ is the least equivalence containing both $\mathfrak{R}$ and $\mathfrak{L}$.

An element $a$ of a semigroup $S$ is said to be *regular* if $asa = a$ for some $s \in S$. A $\mathfrak{D}$-class $D$ is called *regular* if it contains a regular element. (In this case, every element of $D$ is known to be regular; see [9, Theorem 6].) We denote by **DS** the set of all finite semigroups $S$ such that the regular $\mathfrak{D}$-classes of $S$ are subsemigroups in $S$.

The structure of semigroups **DS** is well understood in terms of their decompositions into some basic blocks. As we will use this structural result, we recall the notions involved.

A *semilattice* is a semigroup satisfying the laws of commutativity $xy = yx$ and idempotency $x^2 = x$.

**Definition 2.** Let $Y$ be a semilattice and $\{S_y\}_{y \in Y}$ a family of disjoint semigroups indexed by the elements of $Y$. A semigroup $S$ is said to be a *semilattice of semigroups* $S_y$, $y \in Y$, if:

(S1) $S = \bigcup_{y \in Y} S_y$;

(S2) each $S_y$ is a subsemigroup in $S$;

(S3) for every $y, z \in Y$ and every $s \in S_y$, $t \in S_z$, the product $st$ belongs to $S_{yz}$.

We say that a semigroup $S$ is *m-nilpotent over its kernel* ($m$ being a positive integer) if every product of $m$ elements of $S$ belongs to $\operatorname{Ker} S$. We call a semigroup *nilpotent over its kernel* if it is $m$-nilpotent over its kernel for some $m$. (To a semigroupist, finite semigroups nilpotent over their kernels are familiar as finite *Archimedean* semigroups.)

The following is a specialization of the equivalence (4c) $\Leftrightarrow$ (1b) in [18, Theorem 3] to finite semigroups[1].

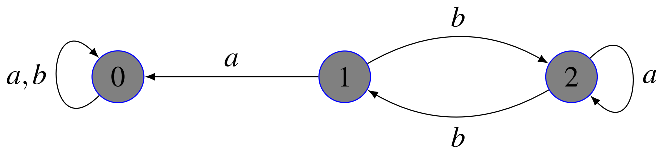
**Lemma 2.** *Every semigroup in* **DS** *is a semilattice of semigroups nilpotent over their kernels.*

## 3 Winning Strategy in Synchronization Games on DS-Automata

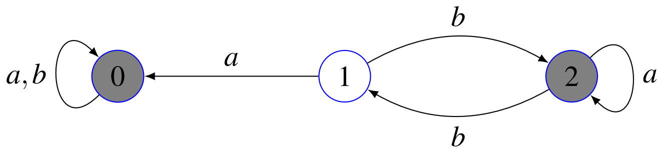
We start with a visual yet rigorous description of the synchronization game under consideration. In this game, two players, Alice and Bob, play on a fixed DFA $\mathscr{A} = (Q, \Sigma)$. At the start, each state in $Q$ holds a token. During the game, some tokens can be removed according to the rules specified in the next paragraph. Alice wins if only one token remains, while Bob wins if he can keep at least two tokens unremoved for an indefinite amount of time.

Alice moves first, and then players alternate moves. The player whose turn is to move proceeds by selecting a letter $a \in \Sigma$. Then, for each state $q \in Q$ that held a token before the move, the token advances to the state $q{\cdot}a$. (In the standard graphical representation of $\mathscr{A}$ as the labelled digraph with $Q$ as the vertex set and the labelled edges of the form $q \xrightarrow{a} q{\cdot}a$, one can visualize the move as follows: all tokens simultaneously slide along the edges labelled $a$.) If several tokens arrive at the same state after this, all of them but one are removed so that when the move is completed, each state holds at most one token.

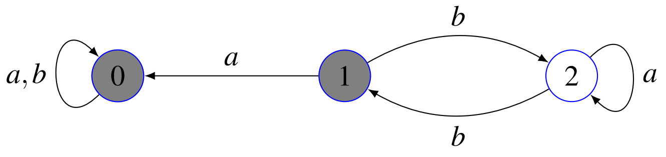
To illustrate, let us look at how the synchronization game might play out on the following DFA in which, initially, each state holds a token (shown in gray).



If Alice chooses the letter $a$ on her first move, the tokens in states 0 and 2 remain due to the loops at these states. The token from state 1 moves to 0 and then is removed because the state 0 is 'occupied'. Hence, the position after Alice's first moves looks as follows:

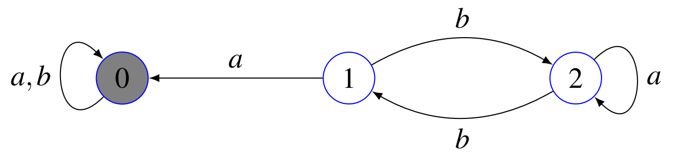


If Bob replies by choosing the letter $b$, the token in state 0 remains while the token from state 2 moves to 1. Here is the position after Bob's reply:



[1]Theorem 3 in [18] deals with semigroups in which every element has a power that belongs to a subgroup. Every finite semigroup has this property.

Now choosing $a$, Alice wins because after the token from state 1 moves to 0, it is removed, and we get the position with only one token:



Notice that Alice won the game described above only because of Bob's unfortunate reply. In fact, Bob has a winning strategy in the synchronization game on this DFA: if he repeats Alice's moves, that is, chooses the same letter Alice chose on her previous move, he can maintain two tokens unremoved forever. Hence, there are simple synchronizing DFAs that are not A-automata.

Recall that a DS-automaton is a DFA whose transition monoid lies in the set **DS** of all finite semigroups with regular $\mathfrak{D}$-classes being subsemigroups. The following is the main result of this note.

**Theorem 3.** *Alice has a winning strategy on every synchronizing DS-automaton.*

*Proof.* Take an arbitrary synchronizing DS-automaton $\mathscr{A} = (Q,\Sigma)$. We denote the transition monoid $T(\mathscr{A})$ by $S$ to lighten the notation. Since $S \in \mathbf{DS}$, by Lemma 2 there is a semilattice $Y$ such that $S$ is a semilattice of semigroups $S_y$, $y \in Y$, where each semigroup $S_y$ is nilpotent over its kernel.

The relation $\leq$ defined by $x \leq y \iff xy = x$ is known (and easy to see) to be a partial order on every semilattice, and so on $Y$. Due to the laws of commutativity and idempotency, the inequalities $xy \leq x$ and $xy \leq y$ hold for all $x,y \in Y$. Since the semilattice $Y$ is finite, it has a least element with respect to this order; denote it by $z$. Then $yz = zy = z$ for every $y \in Y$ whence the semigroup $S_z$ is an ideal in $S$ by item (S3) in Definition 2. Therefore $S_z$ contains the kernel $\operatorname{Ker} S$ of $S$, and therefore, $\operatorname{Ker} S_z \subseteq \operatorname{Ker} S$. (In fact, it is easy to show that the equality $\operatorname{Ker} S_z = \operatorname{Ker} S$ holds, but it is not needed for the present proof.)

Fix a positive integer $m$ such that the semigroup $S_z$ is $m$-nilpotent over its kernel $\operatorname{Ker} S_z$. We show that Alice can win in the synchronization game on $\mathscr{A}$, using the following $m$-round strategy. Denote by $a_i^k$ and $b_i^k$ the $i$-th letters chosen in the $k$-th round by Alice and Bob, respectively. In each round, Alice chooses the first letter $a_1^k$ at random. Then, after each reply of Bob, she checks whether the word $u_i^k := a_1^k b_1^k \cdots a_i^k b_i^k$ causes a transformation in $S_z$. If yes, then Alice starts the next round. If no, the transformation caused by $u_i^k$ lies in some subsemigroup $S_y$ with $y \neq z$ (recall that $S = \bigcup_{y\in Y} S_y$ by item (S1) in Definition 2). For each letter $a \in \Sigma$, denote by $y(a)$ the element of the semilattice $Y$ such that the transformation $\tau_a$ lies in the subsemigroup $S_{y(a)}$. Take any transformation $\tau \in S_z$. By Definition 1, $\tau = \tau_{a_1}\tau_{a_2}\cdots\tau_{a_n}$ for some $a_1,a_2,\ldots,a_n \in \Sigma$ whence by item (S3) in Definition 2, $z = y(a_1)y(a_2)\cdots y(a_n)$. If $y \leq y(a_i)$ for all $i = 1,2,\ldots,n$, then $y \leq y(a_1)y(a_2)\cdots y(a_n) = z$, and this would contradict the choice of $z$ as the least element with respect to $\leq$ and the assumption $y \neq z$. Thus, there must be a letter $a \in \Sigma$ such that $y \not\leq y(a)$, and Alice chooses any such letter $a$ as $a_{i+1}^k$.

By the construction, if $S_y \neq S_z$, then the index $x$ of the subsemigroup $S_x$ containing the transformation caused by the word $u_{i+1}^k$ is strictly less than $y$ in the partially ordered set $(Y;\leq)$. Hence, for each $k$, the number $\ell_k$ of pairs of moves in the $k$-th round does not exceed the maximum length of strictly decreasing chains in $(Y;\leq)$, and the transformation caused by the word $u_{\ell_k}^k$ lies in the subsemigroup $S_z$. Then the transformation $\tau_w$ caused by the word $w := u_{\ell_1}^1 u_{\ell_2}^2 \cdots u_{\ell_m}^m$ is a product of $m$ elements of $S_z$ and so $\tau_w$ belongs to $\operatorname{Ker} S_z$ as the semigroup $S_z$ is $m$-nilpotent over its kernel. Since $\mathscr{A}$ is a synchronizing automaton, the kernel $\operatorname{Ker} S$ of its transition monoid consists of constant transformations by Lemma 1. From the inclusion $\operatorname{Ker} S_z \subseteq \operatorname{Ker} S$ registered above, we conclude that $\tau_w$ is a constant transformation, and so $w$ is a reset word for $\mathscr{A}$. $\square$

## 4 Relations to Earlier Results and Future Work

### 4.1 Corollaries

In the introduction, we mentioned a few previously known families of A-automata. Now we show that all these families consist of DS-automata so their winning strategies are subsumed by that of Theorem 3.

**Definite automata.** This DFA family was introduced by some of the pioneers of automata theory back in 1963 [14]. In [14], the term 'automaton' meant a recognizer, that is, a DFA with a designated initial state and a distinguished set of final states. However, DFAs without initial and final states as defined in the present note also appeared in [14] but under the name 'transition tables'. The following is [14, Definition 13] stated in our terminology and notation.

**Definition 3.** A DFA $(Q,\Sigma)$ is *weakly k-definite* if for every word $w$ of length at least $k$ over $\Sigma$, $q{\cdot}w = q'{\cdot}w$ for all $q,q' \in Q$. A DFA is *k-definite* if it is weakly $k$-definite but not weakly $(k-1)$-definite. A DFA is *definite* if it is $k$-definite for some $k$.

By Definition 3, every definite DFA is synchronizing and any word of length at least $k$ is a reset word for every $k$-definite DFA. Therefore, Alice wins on every definite automaton by selecting her moves at random.

The transition monoid of a definite DFA is nilpotent over its kernel. This fact is implicitly contained in [14] and in the explicit form, it is a part of [20, Theorem 3]. Comparing it with Lemma 2, we see that definite automata constitute a special subfamily of DS-automata. Moreover, for definite automata, Alice's winning strategy from Theorem 3 specializes exactly to the random choice of moves. In fact, a DFA is definite if and only if Alice can win by pure random choices of moves.

**Commutative automata.** A DFA is said *commutative* if its transition monoid is commutative, that is, satisfy the law $xy = yx$. Synchronizing commutative automata were considered in [15, 16, 10]. A simple winning strategy for Alice in synchronization games on such automata was suggested in [7, Theorem 5.2]: Alice must just choose letters spelling a reset word. Hence, as in the previous case, Alice can completely ignore the moves of Bob.

Obviously, on any commutative semigroup, Green's relations $\mathfrak{R}$ and $\mathfrak{L}$ coincide with each other, and hence, with the relation $\mathfrak{D}$. If an element $a$ of a commutative semigroup $S$ is regular, then $a = a^2s$ for some $s \in S$ whence $a \,\mathfrak{R}\, a^2$. By [9, Theorem 7], this ensures that the $\mathfrak{D}$-class containing $a$ is a subsemigroup (and even a subgroup) of $S$. Thus, in all commutative semigroups, regular $\mathfrak{D}$-classes are subsemigroups. Therefore, commutative DFAs are DS-automata, and Theorem 3 applies to commutative synchronizing DFAs.

**Weakly acyclic automata.** Let $\mathscr{A} = (Q,\Sigma)$ be a DFA and $p,q \in Q$. We say that $q$ is *reachable from p in $\mathscr{A}$* if either $p = q$ or there exists a word $w$ over $\Sigma$ such that $q = p{\cdot}w$. The reachability relation of any DFA is reflexive and transitive. A DFA is called *weakly acyclic* if its reachability relation is a partial order.

Various properties of synchronizing weakly acyclic DFAs were considered in [17, 11]. A winning strategy for Alice in synchronization games on such automata was suggested in [7, Theorem 2.3].

By [5, Proposition 6.2] the transition monoid $T(\mathscr{A})$ of any weakly acyclic DFA $\mathscr{A}$ is $\mathfrak{R}$-*trivial*, which means that Green's relation $\mathfrak{R}$ on $T(\mathscr{A})$ coincides with the equality relation. It is well known that regular $\mathfrak{D}$-classes are subsemigroups in any $\mathfrak{R}$-trivial semigroup, but we failed to locate a source where this fact was formulated such that it would be convenient to refer to it. Therefore we state it as a lemma and provide a proof.

**Lemma 4.** *In every $\mathfrak{R}$-trivial semigroup, regular $\mathfrak{D}$-classes are subsemigroups.*

*Proof.* Let $S$ be an $\mathfrak{R}$-trivial semigroup and let $D$ be its regular $\mathfrak{D}$-class. Every element $b \in D$ is regular, so that $b = btb$ for some $t \in S$. Then $b \,\mathfrak{R}\, bt$ whence $b = bt$ since $S$ is $\mathfrak{R}$-trivial. We have $b^2 = btbt = (btb)t = bt = b$. Now if $a \in D$, then $a \,\mathfrak{L}\, b$ since $\mathfrak{D} = \mathfrak{L}$ in every $\mathfrak{R}$-trivial semigroup. By the definition of the relation $\mathfrak{L}$, we have either $a = b$ or $a = sb$ for some $s \in S$. If $a = b$, then $ab = b^2 = b = a$. If $a = sb$, multiplying through on the right by $b$, we get $ab = sb^2 = sb = a$. Thus, $ab = a \in D$ for arbitrary $a, b \in D$, whence $D$ is a subsemigroup. □

Thus, weakly acyclic DFAs are DS-automata, and Theorem 3 applies again.

## 4.2 Open Questions

Theorem 3 generalizes several earlier results on A-automata. Can it be further generalized? This is an interesting question, but it requires specification.

We mentioned in Section 2.1 that synchronization is a property of transition monoids. This is not true for the property of being an A-automaton. Moreover, for every synchronizing DFA $\mathscr{A} = (Q, \Sigma)$, there exists an A-automaton $\mathscr{A}' = (Q, \Sigma')$ such that $T(\mathscr{A}) = T(\mathscr{A}')$. To see this, let $\Sigma' := \Sigma \cup \{c\}$ where the action of the added letter $c$ coincides with the action of a fixed reset word for $\mathscr{A}$. The transformations caused by the letters in $\Sigma'$ generate the same submonoid of the monoid of all transformations on the set $Q$ as do the transformations caused by the letters in $\Sigma$. Still, Alice instantly wins the synchronization game on $\mathscr{A}'$ by choosing the letter $c$ on her first move.

Thus, one cannot hope for a characterization of A-automata in terms of transition monoids. One can, however, try to find new sets $\mathbf{P}$ of finite semigroups such that $\mathbf{DS} \subsetneq \mathbf{P}$ and every synchronizing DFA whose transition monoid lies in $\mathbf{P}$ is an A-automaton.

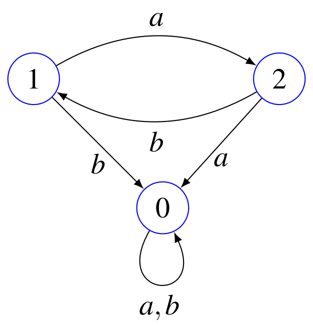
It is easy to verify that the property of being an A-automaton is inherited by subautomata, homomorphic images, and finite direct products. This suggests looking for sets $\mathbf{P}$ closed under corresponding operations with semigroups. A set of finite semigroups closed under forming finite direct products and taking subsemigroups and homomorphic images is called a *pseudovariety*; the set $\mathbf{DS}$ is an example of a pseudovariety. Using the notion of a pseudovariety, we can specify a possible direction towards generalizing Theorem 3 as follows:

**Question 1.** *Is there a pseudovariety* $\mathbf{P}$ *of finite semigroups that strictly contains* $\mathbf{DS}$ *while all synchronizing DFAs with transition monoids in* $\mathbf{P}$ *are A-automata?*

The 5-element *Brandt semigroup* $B_2$ consists of the following five $2 \times 2$-matrices, multiplied according to the usual rule:

$$\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}, \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}.$$

The monoid $B_2^1$ obtained by adding the identity $2 \times 2$-matrix to $B_2$ is the syntactic monoid of the language $(ab)^*$, and hence, the transition monoid of the minimal automaton $\mathscr{M}$ of this language (shown below).



It is known that every pseudovariety of finite semigroups not contained in **DS** must include the semigroup $B_2$ (this fact occurs as Exercise 8.1.6 in [1]; the solution to this exercise follows from the proof of [13, Theorem 3]). Thus, if Bob had a winning strategy on the automaton $\mathscr{M}$, then the answer to Question 1 would be 'No', and moreover, **DS** would be the largest pseudovariety **P** with the property that Alice can win the synchronization game on every synchronizing DFA whose transition monoid lies in **P**. However, it is easy to see that Alice wins on $\mathscr{M}$: she can start with choosing $a$; if Bob replies with $a$, he loses, and if he replies with $b$, Alice wins by choosing $b$.

The fact that $\mathscr{M}$ is an A-automaton, along with other examples of A-automata beyond the family of DS-automata, indicates that Question 1 might have an affirmative answer. We have a candidate pseudovariety to witness such an answer but its definition requires more structure theory of semigroups than assumed here.

Another question of interest concerns the speed of synchronization for A-automata. When we mentioned in the introduction that A-automata seem more amenable to synchronization, we meant that they tend to have short reset words. Indeed, in all examples we know, an A-automaton with $n$ states admits a reset word of length at most $n-1$. For DFAs in the range of Theorem 3, that is, synchronizing DS-automata, this was established in [3, Theorem 2.6]. The DFA $\mathscr{M}$ with 3 states is reset by the words $a^2$ and $b^2$ of length 2 and hence provides another example. These observations lead to the next question.

**Question 2.** *Is each A-automaton with n states reset by a word of length $n-1$?*

Recall that for each $n > 2$, there exist synchronizing DFAs with $n$ states whose shortest reset words have length $(n-1)^2$; see [6, Lemma 1]. It can be verified that none of these 'slowly synchronizing' DFAs are A-automata.

**Acknowledgement.** The authors are grateful to the anonymous referees for their attention and remarks.

## References

[1] Almeida, Jorge: Finite Semigroups and Universal Algebra. Series in Algebra, vol. 3. World Scientific, Singapore (1994)

[2] Almeida, Jorge, Margolis, Stuart, Steinberg, Benjamin, Volkov, Mikhail: Representation theory of finite semigroups, semigroup radicals and formal language theory, Transactions of the American Mathematical Society **361**:3, 1429–1461 (2009). https://doi.org/10.1090/S0002-9947-08-04712-0

[3] Almeida, Jorge, Steinberg, Benjamin: Matrix mortality and the Černý–Pin conjecture. In: Diekert, Volker, Nowotka, Dirk (eds.), Developments in Language Theory, 13th International Conference (DLT 2009), Lecture Notes in Computer Science, vol. 5583, pp. 67–80. Springer, Berlin (2009). https://doi.org/10.1007/978-3-642-02737-6_5

[4] Blass, Andreas, Gurevich, Yuri, Nachmanson, Lev, Veanes, Margus: Play to test. In: Grieskamp, Wolfgang, Weise, Carsten (eds.), Formal Approaches to Software Testing, 5th International Workshop (FATES 2005), Lecture Notes in Computer Science, vol. 3997, pp. 32–46. Springer, Berlin, Heidelberg (2006). `https://doi.org/10.1007/11759744_3`

[5] Brzozowski, Janusz A., Fich, Faith E.: Languages of $\mathscr{R}$-trivial monoids. Journal of Computer and System Sciences **20**:1, 32–49 (1980). `https://doi.org/10.1016/0022-0000(80)90003-3`

[6] Černý, Jan: Poznámka k homogénnym eksperimentom s konečnými automatami. Matematicko-fyzikalny Časopis Slovenskej Akadémie Vied **14**(3): 208–216 (1964) (in Slovak; English translation: A note on homogeneous experiments with finite automata. Journal of Automata, Languages, and Combinatorics **24**:2-4, 123–132 (2019). `https://doi.org/10.25596/jalc-2019-123`)

[7] Fernau, Henning, Haase, Carolina, Hoffmann, Stefan: The synchronization game on subclasses of automata. In: Fraigniaud, Pierre, Uno, Yushi (eds.), 11th International Conference on Fun with Algorithms (FUN 2022), Leibniz International Proceedings in Informatics (LIPIcs), vol. 226, pp. 14:1–14:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl (2022). `https://doi.org/10.4230/LIPIcs.FUN.2022.14`

[8] Fominykh, Fedor M., Martyugin, Pavel V., Volkov, Mikhail V.: P(l)aying for synchronization, International Journal of Foundations of Computer Science **24**:6, 765–780 (2013). `https://doi.org/10.1142/S0129054113400170`

[9] Green, James Alexander: On the structure of semigroups. Annals of Mathematics, Second Series **54**:1, 163–172 (1951). `https://doi.org/10.2307/1969317`

[10] Hoffmann, Stefan: Constrained synchronization and commutativity. Theoretical Computer Science **890**, 147-–170 (2021). `https://doi.org/10.1016/j.tcs.2021.08.030`

[11] Hoffmann, Stefan: Constrained synchronization and subset synchronization problems for weakly acyclic automata, in: Moreira, Nelma, Reis, Rogério (eds.), Developments in Language Theory, 25th International Conference (DLT 2021), Lecture Notes in Computer Science, vol. 12811, pp. 204–216. Springer, Cham (2021). `https://doi.org/10.1007/978-3-030-81508-0_17`

[12] Kari, Jarkko, Volkov, Mikhail: Černý's conjecture and the road colouring problem. In: Pin, Jean-Éric (ed.), Handbook of Automata Theory, vol. I, pp. 525–565. EMS Publishing House, Zürich (2021). `https://doi.org/10.4171/AUTOMATA-1/15`

[13] Margolis, Stuart W. On M-varieties generated by power monoids. Semigroup Forum **22**, 339–353 (1981). `https://doi.org/10.1007/BF02572813`

[14] Perles, Micha, Rabin, Michael O., Shamir, Eliahu: The theory of definite automata. Transactions on Electronic Computers **12**, 233–243 (1963). `https://doi.org/10.1109/PGEC.1963.263534`

[15] Rystsov, Igor: Exact linear bound for the length of reset words in commutative automata. Publicationes Mathematicae Debrecen **48**:3-4, 405–409 (1996)

[16] Rystsov, Igor: Reset words for commutative and solvable automata. Theoretical Computer Science **172**:1–2, 273–279 (1997). `https://doi.org/10.1016/S0304-3975(96)00136-3`

[17] Ryzhikov, Andrew: Synchronization problems in automata without non-trivial cycles. Theoretical Computer Science **787**, 77–88 (2019). `https://doi.org/10.1016/j.tcs.2018.12.026`

[18] Shevrin, Lev N.: On the theory of epigroups. I. Russian Academy of Sciences. Sbornik. Mathematics **82**:2, 485–512 (1995). `https://doi.org/10.1070/SM1995v082n02ABEH003577`

[19] Volkov, Mikhail V.: Synchronization of finite automata. Russian Mathematical Surveys **77**:5, 819–891 (2022). `https://doi.org/10.4213/rm10005e`

[20] Zalcstein, Yechezkel: Locally testable languages. Journal of Computer and System Sciences **6**, 151–167 (1972). `https://doi.org/10.1016/S0022-0000(72)80020-5`

# How to Demonstrate Metalinearness and Regularity by Tree-Restricted General Grammars

Martin Havel
Brno University of Technology,
Faculty of Information Technology,
Brno, Czech republic
ihavelm@fit.vut.cz

Zbyněk Křivka
Brno University of Technology,
Faculty of Information Technology,
Brno, Czech republic
krivka@fit.vut.cz

Alexander Meduna
Brno University of Technology,
Faculty of Information Technology,
Brno, Czech republic
meduna@fit.vut.cz

This paper introduces derivation trees for general grammars. Within these trees, it defines context-dependent pairs of nodes, corresponding to rewriting two neighboring symbols using a non context-free rule. It proves that the language generated by a linear core general grammar with a slow-branching derivation tree is $k$-linear if there is a constant $u$ such that every sentence $w$ in the generated language is the frontier of a derivation tree in which any pair of neighboring paths contains $u$ or fewer context-dependent pairs of nodes. Next, it proves that the language generated by a general grammar with a regular core is regular if there is a constant $u$ such that every sentence $w$ in the generated language is the frontier of a derivation tree in which any pair of neighboring paths contains $u$ or fewer context-dependent pairs of nodes. The paper explains that this result is a powerful tool for showing that certain languages are $k$-linear or regular.

## 1 Introduction

Formal language theory has always intensively struggled to establish conditions under which general grammars generate a proper subfamily of the family of recursively enumerable languages because results like this often significantly simplify proofs that some languages are members of the subfamily. Continuing with this important investigation trend in formal language theory, the present paper establishes another result of this kind based upon a restriction illustrated in Fig. 1 placed upon a graph-based representation of derivations in general grammars.

Concerning general grammars, which generate a proper subfamily of the family of recursively enumerable languages, some results of this kind have been achieved, too. First of all, [9] states that for a grammar, the set of terminal strings generated by left-to-right derivations is context-free. Second, [10] shows that the set of terminal strings generated by two-way derivations is context-free, which is further studied in [4]. Third, [3] demonstrates that a grammar generates a context-free language if the left-hand side of every rule contains only one nonterminal with terminal strings as the only context. Fourth, also [3] shows that if every rule of a general grammar has as its left context a string of terminal symbols at least as long as the right context, then the generated language is context-free. Fifth, [2] demonstrates that a grammar generates a context-free language if the right-hand side of every rule contains a string of terminals longer than any string of terminals between two nonterminals on the left-hand side. For $k$-linear grammar, there is no such study. For regularity, there is the publication [6], which shows regularity only in context-free languages.

Finally, Section 2.3.2 in [13] demonstrates context-freeness based on the tree restriction with context-dependency. We explain and expand the importance of introduced context-dependency (see Fig. 1) to demonstrate metalinearness and regularity.

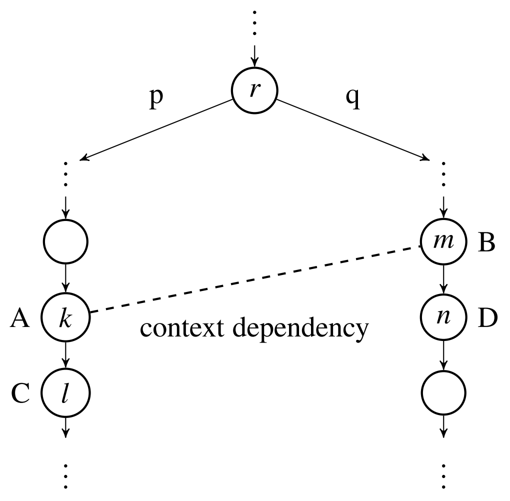


Figure 1: Illustration of context dependency in $t$

To give an insight into the new result achieved in the present paper, some terminology is first needed to be sketched. We introduce a linear core general grammar $G$ if any $p \in P$ has one of these forms,

$$AB \rightarrow CD, A \rightarrow BC, A \rightarrow xEy$$

where $A,B,C,D$ are nonterminals, $E$ is a nonterminal or the empty string, and $x,y$ are strings of terminals.

We define the notion of a derivation tree $t$ graphically representing a derivation in $G$ by analogy with this notion in terms of a $k$-linear grammar (see Definiton 3 or Section 6.2 in [14]). However, in addition, we introduce context-dependent pairs of nodes in $t$ as follows. In $t$, two paths are neighboring if no other path occurs between them. Let $p$ and $q$ be two neighboring paths in $t$. Let $p$ contain a node $k$ with a single child $l$, where $k$ and $l$ are labeled with $A$ and $C$, respectively, and let $q$ contain a node $m$ with a single child $n$, where $m$ and $n$ are labeled with $B$ and $D$, respectively. Let this four-node portion of $t$; consisting of $k$, $l$, $m$, and $n$; graphically represents an application of $AB \rightarrow CD$. Then, $k$ and $m$ are a context-dependent pair of nodes (see Fig. 1).

The main theorem provided in this paper represents a powerful tool to demonstrate that if a linear core general grammar $H$ generates each of its sentences by a derivation satisfying prescribed conditions (specifically, one of these conditions requires that there is a positive integer $u$ and any two nonterminal neighboring paths contain no more than $u$ pairs of context-dependent nodes) then the language generated by $H$ is $k$-linear. Similarly, the following theorems provide a tool to demonstrate membership in the regular language family.

## 2 Preliminaries

We assume that the reader is familiar with graph theory, including labeled ordered trees and their terminology (see [1, 5, 7]) as well as formal language theory (see [12, 14, 15]).

A directed graph $G$ is a pair $G = (V,E)$, where $V$ is a finite set of nodes and $E \subseteq V^2$ is a finite set of edges. For a node $v \in V$ a number of edges of the form $(x,v) \in E$ and a number of edges of the form $(v,y) \in E$, for $x,y \in V$, is called an in-degree of $v$ and an out-degree of $v$, respectively, and denoted by in-d$(v)$, out-d$(v)$. Let $(v_1, v_2, \ldots, v_n)$ be an $n$-tuple of nodes, for some $n \geq 1$, where $v_i \in V$, for $1 \leq i \leq n$, and there exists an edge $(v_k, v_{k+1}) \in E$, for every pair of nodes $v_k, v_{k+1}$, where $1 \leq k \leq n-1$, then, we call it a sequence of length $n$. Let $(v_1, v_2, \ldots, v_n)$ be a sequence of the length $n$, for some $n \geq 1$, where $v_i \neq v_j$, for $1 \leq i \leq n$, $1 \leq j \leq n$, then, we call the sequence a path. Let $(v_1, v_2, \ldots, v_n)$ be a path in $G$, for some $n \geq 1$, and $v_1 = v_n$, then we call it a cycle. A graph $G$ is acyclic iff it contains no cycle.

For a set $W$, card$(W)$ denotes its cardinality. Let $V$ be an alphabet (finite non-empty set). $V^*$ is the set of all strings over V. Algebraically, $V^*$ represents the free monoid generated by $V$ under the operation of concatenation. The unit of $V^*$ is denoted by $\varepsilon$. Set $V^+ = V^* - \{\varepsilon\}$. Algebraically, $V^+$ is thus the free semigroup generated by $V$ under the operation of concatenation. For $w \in V^*$, $|w|$ denotes the length of $w$. The alphabet of $w$, denoted by alph$(w)$, is the set of symbols appearing in $w$. Let $\mathscr{I}$ denote the set of all positive integers.

Let $\Rightarrow$ be a relation over $V^*$. The transitive and transitive and reflexive closure of $\Rightarrow$ are denoted $\Rightarrow^+$ and $\Rightarrow^*$, respectively. Unless explicitly stated otherwise, we write $x \Rightarrow y$ instead of $(x,y) \in \Rightarrow$.

The families of context-free, context-sensitive and recursively enumerable languages are denoted by **CF**, **CS** and **RE**, respectively.

## 3 Definitions and Examples

**Definition 1.** *An* (oriented) tree *is a directed acyclic graph $G = (V, E)$, with a specified node $r \in V$ called the* root *such that* in-d$(r) = 0$, in-d$(x) = 1$, *and there exists a path $(v_1, v_2, \ldots, v_n)$, where $v_1 = r$, $v_n = x$, for some $n \geq 1$, for all $x \in V - \{r\}$. For $v, u \in V$, where $(v, u) \in E$, $v$ is called a* parent *of $u$, $u$ is called a child of $v$, respectively. For $v, u, z \in V$, where $(v, u), (v, z) \in E$, $u$ is called a* sibling *of $z$ and vice versa.*

*A tree is called* labeled, *if there exist a set of labels $\mathscr{L}$ and a total mapping $l : V \to \mathscr{L}$.*

*An* ordered tree *$t$ is a tree, where for every set of siblings there exists a linear ordering. Let $o$ has the children $n_1, n_2, \ldots, n_r$ ordered in this way, where $r \geq 1$. Then $n_1$ is the* leftmost child *of $o$, $n_r$ is the* rightmost child *of $o$ and $n_i$ is the* direct left sibling *of $n_{i+1}$, $n_{i+1}$ is the* direct right sibling *of $n_i$, $1 \leq i \leq r-1$, and for $j < k$, $n_j$ is* left sibling *of $n_k$ and $n_k$ is* right sibling *of $n_j$, $1 \leq j \leq r$, $1 \leq k \leq r$.*

*Let $t$ be a labeled ordered tree, and let $t$ contain node $o$. Let $\alpha = (o, m_1, m_2, \ldots, m_r)$, and $\beta = (o, n_1, n_2, \ldots, n_s)$ be two paths in $t$, for some $r, s \geq 1$, such that $o$ is the parent of $m_1$ and $n_1$, while*

1. *$m_1$ is the direct left sibling of $n_1$;*
2. *$m_i$ is a nonterminal child of $m_{i-1}$, while all its right siblings are terminal siblings, $2 \leq i \leq r-1$, $n_j$ is a nonterminal child of $n_{j-1}$, while all its left siblings are terminal siblings, $2 \leq j \leq s-1$;*
3. *if $m_r$ is a terminal node, then all its siblings are terminal nodes; otherwise, all its right siblings are terminal siblings;*
4. *if $n_s$ is a terminal node, then all its siblings are terminal nodes; otherwise, all its left siblings are terminal siblings;*

*Then, $\alpha$ and $\beta$ are two* nonterminal neighboring paths *in $t$, $\alpha$ is a* left nonterminal neighboring path *to $\beta$, and $\beta$ is a* right nonterminal neighboring path *to $\alpha$.*

Next, we define the notion of a general grammar, also known as that of a type-0 grammar or that of a phrase-structure grammar in the literature.

**Definition 2.** *A* general grammar *(GG) $G$ is a quadruple $G = (V, T, P, S)$, where $V$ is a* total alphabet, *$T \subset V$ is a* terminal alphabet, *$P$ is a finite* set of rules *of the form $x \to y$, where $x, y \in V^*$,* alph$(x) \cap (V - T) \neq \emptyset$, *$S \in V - T$ is a* start symbol. *For every $u, v \in V^*$ and $x \to y \in P$, $uxv \Rightarrow uyv[p]$ or simply*

*$uxv \Rightarrow uyv$ is a* derivation step *of $G$ from $uxv$ to $uyv$ by the rule $x \rightarrow y$, $\Rightarrow$ is the* direct derivation *relation. Let $w_0, w_1, \ldots, w_n \in V^*$, for some $n \geq 1$, such that $w_0 \Rightarrow w_1\,[p1] \Rightarrow \ldots \Rightarrow w_n\,[p_n]$, where $p_i \in P$, for all $i = 1, \ldots, n$, then, $w_0 \Rightarrow^n w_n$; based on $\Rightarrow^n$, we define $\Rightarrow^+$ and $\Rightarrow^*$.*

*A* language *of $G$ is $L(G) = \{w \in T^* \mid S \Rightarrow^* w\}$. $G$ is* propagating *if $A \rightarrow x \in P$ implies $x \neq \varepsilon$. $G$ is* context-free *if $A \rightarrow x \in P$ implies $A \in V - T$. $G$ is* linear core *GG if any $p \in P$ has one of these forms:*

$$AB \rightarrow CD, A \rightarrow BC, A \rightarrow xEy$$

*where $A, B, C, D \in V - T$, $E \in (V - T) \cup \{\varepsilon\}$, $x, y \in T^*$. In what follows, unless explicitly stated otherwise, we assume that every GG is a linear core GG.*
*Similary, $G$ is* left linear core *GG if any $p \in P$ has one of these forms (see [8]):*

$$AB \rightarrow CD, A \rightarrow BC, A \rightarrow xE$$

*where $A, B, C, D \in V - T$, $E \in (V - T) \cup \{\varepsilon\}$, $x \in T^*$.*
*$G$ is GG in the* Kuroda normal form *(KNF) [11] if every rule is one of these forms:*

$$AB \rightarrow CD, A \rightarrow BC, A \rightarrow B, A \rightarrow a, A \rightarrow \varepsilon$$

*where $A, B, C, D \in V - T$, $a \in T$.*

As obvious, all rules of the form of $A \rightarrow B$ can be always removed from $G$ without disturbing $L(G)$. Next, we show that the proposed linear core grammars have the same generative power as GGs.

**Lemma 1.** *A language $L$ is recursively enumerable iff $L = L(G)$, where $G$ is a linear core general grammar.*

*Proof.* Every language $L$ generated by a linear core GG $G$ is recursively enumerable, because every general linear core grammar can be trivially converted to KNF. In other direction, every KNF $G$ is a linear core GG by Definition 2.

□

**Lemma 2.** *A language $L$ is context-sensitive iff $L = L(G)$, where $G$ is a propagating linear core general grammar.*

*Proof.* Every language $L$ generated by propagating linear core GG $G$ is context sensitive, because each rule in $P$, where $P \in G$, is a form of $x \rightarrow y$ and $|x| \leq |y|$.

□

**Definition 3.** *A* linear grammar *$G$ is a GG $G = (V,\ T,\ P,\ S)$, where $P$ contains rules of the form:*

$$A \rightarrow x$$

*where $A \in (V - T)$, $x \in T^*((V - T) \cup \{\varepsilon\})T^*$. A language is* linear *(1-linear) if it can be generated by a linear grammar. The concept of a linear grammar can be generalized: A $k$-*linear grammar *$G$ is a GG $G = (V,\ T,\ P,\ S)$, where $P$ is a finite set of rules of the form:*

$$A \rightarrow x, A \rightarrow xBy, S \rightarrow W$$

*where $A, B \in (V - T)$, $x, y \in T^*$, $W \in (V - (T \cup \{S\}))^k$. A language is said to be $k$-*linear *if it can be generated by a k-linear grammar. A language is said to be* metalinear *if it is k-linear for some positive integer k.*

**Definition 4.** *Let $G = (V, T, P, S)$ be a linear core GG without rules of the form $AB \rightarrow CD$. Let $w \in T^*$ be a string derived from G. A derivation tree for w is a labeled tree $\tau$ such that:*

1. *The root of $\tau$ is labeled with S.*
2. *Each leaf of $\tau$ is labeled with a symbol from $T$.*
3. *Each internal node of $\tau$ is labeled with a symbol from $V$.*
4. *If an internal node v is labeled with $A \in V$ and has children labeled $B_1$, $B_2$, then there exists a rule $A \rightarrow B_1B_2$ in P and, analogically, for the rest of the rules of a linear core GG without rules of the form $AB \rightarrow CD$.*
5. *The yield of $\tau$ (that is, the concatenation of the labels on its leaves), denoted by* $\text{frontier}(\tau)$*, is w.*

**Example 1.** *The following graph (Fig. 2) represents a labeled ordered tree t for a GG in KNF. Since any two distinct nodes have different labels, we refer to their labels below. The root node $\hat{r}$ is a. It has no parent and two children b and c. Then b is a sibling of c and c is a sibling of b. The leftmost child of b is d, while the rightmost is e. The node d is a left sibling of e. The node d is the parent of h, but h has no child, so it is a leaf node. horsm =* $\text{frontier}(t)$*. Consider the node e. The nodes a and b are predecessors of e, while i, j, o, p, and r are descendants of e. The nodes c or d are not in predecessor relation with e, as they are neither predecessors of e, nor descendants of e. The sequence of nodes bejpr is a path in t. The path cglqs is neighboring to bejpr; unlike acglqs, eio or bdh.*

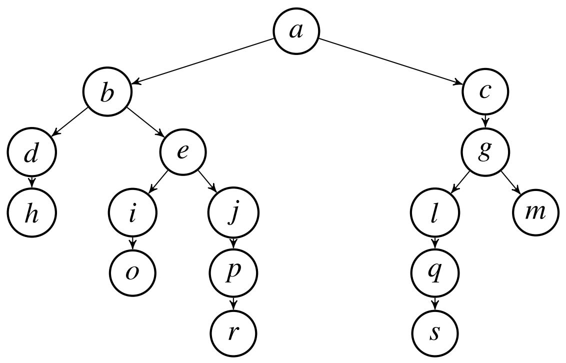


Figure 2: Labeled ordered tree *t*

**Definition 5.** *Let $G = (V, T, P, S)$ be a linear core GG.*

1. *For $p\colon A \rightarrow x \in P$, $A\langle x\rangle$ is the* rule tree *that represents p.*
2. *The* derivation trees *representing the derivations in G are defined recursively as follows:*
   (a) *One-node tree with a node labeled X is the derivation tree corresponding to $X \Rightarrow^0 X$ in G, where $X \in V$. If $X = \varepsilon$, we refer to the node labeled X as* $\varepsilon$-node *(*$\varepsilon$-leaf*); otherwise, we call it* non-$\varepsilon$-node *(*non-$\varepsilon$-leaf*).*
   (b) *Let d be the derivation tree with* $\text{frontier}(d) = uAv$ *representing $X \Rightarrow^* uAv\,[\rho]$ and let $p\colon A \rightarrow x \in P$. The derivation tree that represents*

$$X \Rightarrow^* uAv\,[\rho] \Rightarrow uxv\,[p]$$

   *is obtained by replacing the ith non-$\varepsilon$-leaf in d labeled A, with the rule tree corresponding to p, $A\langle x\rangle$, where $i = |uA|$.*

(c) *Let $d$ be the derivation tree with* frontier$(d) = uABv$ *representing* $X \Rightarrow^* uABv\,[\rho]$ *and let* $p\colon AB \to CD \in P$. *The derivation tree that represents*

$$X \Rightarrow^* uABv\,[\rho] \Rightarrow uCDv\,[p]$$

*is obtained by replacing the ith and $(i+1)$th non-$\varepsilon$-leaf in $d$ labeled $A$ and $B$ with $A\langle C\rangle$ and $B\langle D\rangle$, respectively, where $i = |uA|$.*

3. *A* derivation tree *in $G$ is any tree $t$ for which there is a derivation represented by $t$ (see item 2 in this definition).*

*Note that the figure to illustrate the definition is postponed to Example 2. Moreover, after replacement in 2c, the nodes $A$ and $B$ are the parents of the new leaves $C$ and $D$, respectively, and we say that $A$ and $B$ are* context-dependent, *alternatively speaking, we say that there is a context dependency between $A$ and $B$. In a derivation tree, two nodes are* context-independent *if they are not context-dependent.*

*Then, for any $p\colon A \to x \in P$, ${}_G\triangle(p)$ denotes the rule tree corresponding to $p$. For any $A \Rightarrow^* x\,[\rho]$ in $G$, where $A \in V - T$, $x \in V^*$, and $\rho \in P^*$, ${}_G\triangle(A \Rightarrow^* x\,[\rho])$ denotes one of the derivation trees corresponding to $A \Rightarrow^* x\,[\rho]$. Just like we often write $A \Rightarrow^* x$ instead of $A \Rightarrow^* x\,[\rho]$, we sometimes simplify ${}_G\triangle(A \Rightarrow^* x\,[\rho])$ to ${}_G\triangle(A \Rightarrow^* x)$ in what follows if there is no danger of confusion. Let ${}_G\blacktriangle$ denote the set of all derivation trees in $G$. Finally, by ${}_G\triangle_x \in {}_G\blacktriangle$, we mean a derivation tree whose frontier is $x$, where $x \in L(G)$.*

*If a node is labeled with a terminal, it is called a* terminal node. *If a node is labeled with a nonterminal, it is called a* nonterminal node. *Analogously, we define the notions of a* terminal child, nonterminal child, terminal sibling, nonterminal sibling. *If a node is labeled with a nonterminal and has two* nonterminal node *children, it is called a* branching nonterminal node. *Let $\alpha = (o, m_1, m_2, \ldots, m_r)$ and $\beta = (o, n_1, n_2, \ldots, n_s)$ be two neighboring paths, where $r, s \geq 0$, $\alpha$ is the left neighboring path to $\beta$, and $m_r$ and $n_s$ are terminal nodes. Then, there is a $t$-tuple $\gamma = (g_1, g_2, \ldots, g_t)$ of nodes from $\alpha$ and $t$-tuple $\delta = (h_1, h_2, \ldots, h_t)$ of nodes from $\beta$, where $g_p < g_q$, for $1 \leq p < q \leq t$, $t <$ min$(r,s)$, and $g_i$ and $h_i$ are context-dependent, for $1 \leq i \leq t$. Let $\rho = p_1 p_2 \ldots p_t$ be a string of non-context-free rules corresponding to context dependencies between $\gamma$ and $\delta$. We call $\rho$ the* right context of $\alpha$ *and the* left context of $\beta$ *or the* context of $\alpha$ and $\beta$. *Consider a node $m_i \in \alpha$, where $1 \leq i \leq r$, and two $(t-k+1)$-tuples of nodes $\sigma = (g_k, g_{k+1}, \ldots, g_t)$ and $\varphi = (h_k, h_{k+1}, \ldots, h_t)$, where $k$ is a minimal integer such that $m_i < g_k$. Then, a string of non-context-free rules $\tau = p_k p_{k+1} \ldots p_t$ corresponding to context dependencies between $\sigma$ and $\varphi$ is called the* right descendant context of *$m_i$, for some $1 \leq k \leq t$. Analogously, we define the notion of the* left descendant context *of a node $n_j$ in $\beta$, for some $1 \leq j \leq s$.*

**Definition 6.** *A labeled ordered tree $t$ is* slow-branching *if any of its pairs of nonterminal neighboring paths contains no more than two nonterminal nodes having two nonterminal children and there is no reachable terminal node from nodes of the path between the root and any branching nonterminal node. A slow-branching labeled ordered tree is of* degree *$k$ if it contains $k$ branching nonterminal nodes, $k \geq 1$.*

**Example 2.** *Let $G = (V, T, P, S)$ be a GG, where $V = N \cup T$ such that $N = \{S, X, Y, Z, A_1, A_2, B, C_1, C_2, D_1, D_2, E, F_1, F_2\}$, $T = \{a, b, c, 0, 1\}$, and $P$ contains the following rules:*

*(1)* $S \rightarrow A_1X$

*(2)* $X \rightarrow A_2Y$

*(3)* $Y \rightarrow BZ$

*(4)* $Z \rightarrow C_1C_2$

*(5)* $A_1 \rightarrow aA_1$

*(6)* $A_2 \rightarrow A_2a$

*(7)* $B \rightarrow bBc$

*(8)* $C_1 \rightarrow aC_1$

*(9)* $C_2 \rightarrow C_2b$

*(10)* $A_1A_2 \rightarrow D_1D_2$

*(11)* $B \rightarrow E$

*(12)* $C_1C_2 \rightarrow F_1F_2$

*(13)* $D_1 \rightarrow 0D_1$

*(14)* $D_2 \rightarrow D_21$

*(15)* $E \rightarrow 0E1$

*(16)* $F_1 \rightarrow 0F_1$

*(17)* $F_2 \rightarrow F_21$

*(18)* $D_1 \rightarrow \varepsilon$

*(19)* $D_2 \rightarrow \varepsilon$

*(20)* $E \rightarrow \varepsilon$

*(21)* $F_1 \rightarrow \varepsilon$

*(22)* $F_2 \rightarrow \varepsilon$

*A graph representing ${}_G\triangle(S \Rightarrow^* aaa0011a0011b)$ is illustrated in Fig. 3 and illustrate slow-branchingness. The graph is slow-branching since it has exactly k branching nodes. Those are $S, X, Y, Z$. That any of its pairs of nonterminal neighboring paths contains no more than two nonterminal nodes having two nonterminal children and there is no reachable terminal node from nodes of the path between the root and any branching nonterminal node. Observe that terminal nodes, denoted by square, do not influence any condition.*

*Let us note that dashed lines and numbers contour only denote the context dependencies, and applied non-context-free rules, respectively, and are not part of the derivation tree. The pairs of context-dependent nodes are linked with dashed lines, all the other nodes are context-independent.*

*Since $aaa0011a0011b = \text{frontier}({}_G\triangle_{aaa0011a0011b})$, all leaves are terminal nodes. Every other node is a nonterminal node.*

*For a pair of neighboring paths $\alpha = SA_1A_1D_1\varepsilon$ and $\beta = SXA_2A_2A_2D_2\varepsilon$, a string $\rho = 10$ is their context, it is the left context of $\beta$ and the right context of $\alpha$.*

## 4 Results

**Theorem 1.** *A language L is k-linear iff there is a constant $k \geq 0$, constant $u \geq 0$ and a linear core general grammar G such that $L = L(G)$ and for every $x \in L(G)$, there is a slow-branching tree of degree k denoted by ${}_G\triangle_x \in {}_G\blacktriangle$ that both following satisfies:*

1. *any two nonterminal neighboring paths contain no more than u pairs of context-dependent nodes;*
2. *all pairs of nodes occurring in non-neighboring paths are context-independent.*

*Proof. Construction.* Consider any $u \geq 0$. Let $G = (V, T, P, S)$ be a GG such that $L(G) = L$. Set $N = V - T$. Let $P_{cs} \subseteq P$ denote the set of all non-context-free rules of $G$. Set

$$N' = \{A_{l|r} \mid A \in N,\ l, r \in (P_{cs} \cup \{\varepsilon\})^u\}.$$

Construct a grammar $G' = (V', T, P', S_{\varepsilon|\varepsilon})$, where $V' = N' \cup T$. Set $P' = \emptyset$. Construct $P'$ by performing (I) through (III) given next.

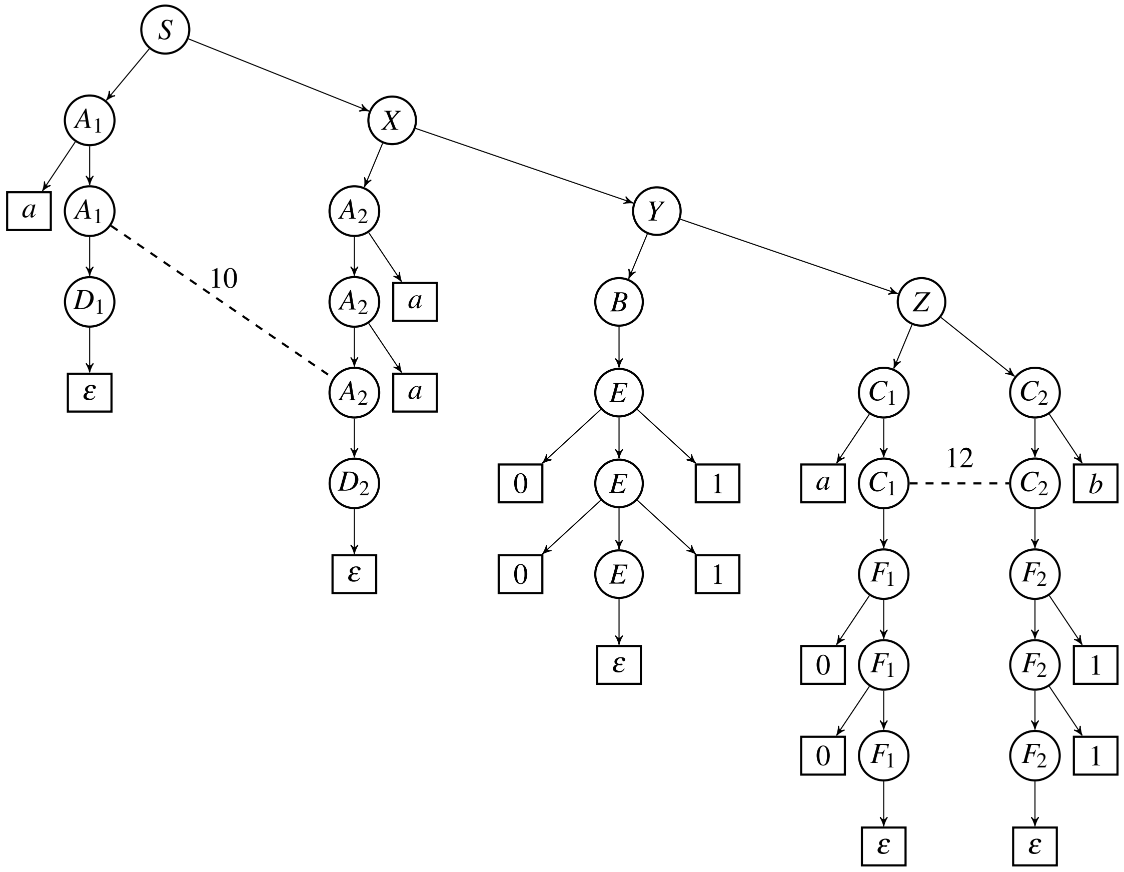


Figure 3: ${}_{G}\triangle_{aaa0011a0011b}$

(I) For all $A \rightarrow xEy \in P$, $A \in N$, $E \in N \cup \{\varepsilon\}$, $x,y \in T^*$, and $l,r \in (P_{cs} \cup \{\varepsilon\})^u$, if $E \in \{\varepsilon\}$ then add $A_{\varepsilon|\varepsilon} \rightarrow xy$ to $P'$ else add $A_{l|r} \rightarrow xE_{l|r}y$ to $P'$;

(II) for all $A \rightarrow BC \in P$, where $A,B,C \in N$, and $r,l,x \in (P_{cs} \cup \{\varepsilon\})^u$, add $A_{l|r} \rightarrow B_{l|x}C_{x|r}$ to $P'$;

(III) for all $p\colon AB \rightarrow CD \in P, A,B,C,D \in N, x,z \in (P_{cs} \cup \{\varepsilon\})^u$, and $y \in (P_{cs} \cup \{\varepsilon\})^{u-1}$, add $A_{x|py} \rightarrow C_{x|y}$ and $B_{py|z} \rightarrow D_{y|z}$ to $P'$.

*Basic idea.* Notice nonterminal symbols. Since every pair of neighboring paths of $G$ contains a limited number of context-dependent nodes, all of its context-dependencies are encoded in nonterminals. $G'$ nondeterministically decides about all context-dependencies while introducing a new pair of neighboring paths by rules from (II). A new pair of neighboring paths is introduced with every application of

$$A_{l|r} \rightarrow B_{l|x}C_{x|r},$$

where $x$ encodes a new descendant context. Context dependencies are realized later by context-free rules from (III).

Since $P'$ contains no non-context-free rule and $G'$ is context-free. Next, we prove $L(G) = L(G')$ by establishing Claims 1 through 3. Define the new homomorphism $\gamma\colon V' \rightarrow V$, $\gamma(A_{l|r}) = A$, for $A_{l|r} \in N'$, and $\gamma(a) = a$ otherwise.

**Claim 1.** *If $S \Rightarrow^m w$ in $G$, where $m \geq 0$ and $w \in V^*$, then $S_{\varepsilon|\varepsilon} \Rightarrow^* w'$ in $G'$, where $w' \in V'^*$ and $\gamma(w') = w$.*

*Proof.* We prove this by induction on $m \geq 0$.

*Basis.* Let $m = 0$. That is $S \Rightarrow^0 S$ in $G$. Clearly, $S_{\varepsilon|\varepsilon} \Rightarrow^0 S_{\varepsilon|\varepsilon}$ in $G'$, where $\gamma(S_{\varepsilon|\varepsilon}) = S$, so the basis holds.

*Induction Hypothesis.* Suppose that there exists $n \geq 0$ such that Claim 1 holds for all $0 \leq m \leq n$.

*Induction Step.* Let $S \Rightarrow^{n+1} w$ in $G$. Then, $S \Rightarrow^n v \Rightarrow w$, where $v \in V^*$, and there exists $p \in P$ such that $v \Rightarrow w\,[p]$. By the induction hypothesis, $S_{\varepsilon|\varepsilon} \Rightarrow^* v'$, where $\gamma(v') = v$, in $G'$. Next, we consider the following three forms of $p$.

(I) Let $p\colon A \to xEy \in P$, for some $A \in N$, $E \in N \cup \{\varepsilon\}$, $x, y \in T^*$.
If there is no nonterminal on the right-hand side of the rule, it implies that left descendant context and a right descendant context of $A$ is $\varepsilon$, then, by the construction of $G'$, there exists a rule $p'\colon A_{\varepsilon|\varepsilon} \to xy \in P'$, where $A_{\varepsilon|\varepsilon} \in v'$. Otherwise, suppose $l$ and $r$ are a left descendant context and a right descendant context of $A$. By the construction of $G'$, there exists a rule $p'\colon A_{l|r} \to xE_{l|r}y \in P'$, where $A_{l|r} \in v'$. Then, there exists a derivation $v' \Rightarrow w'\,[p']$ in $G'$, where $\gamma(w') = w$.

(II) Let $p\colon A \to BC \in P$, for some $A, B, C \in N$. Without any loss of generality, suppose $l$ and $r$ are a left descendant context and a right descendant context of $A$, and $x \in (P_{cs} \cup \{\varepsilon\})^u$ is a context of neighboring paths beginning at this node. By the construction of $G'$, there exists a rule $p'\colon A_{l|r} \to B_{l|x}C_{x|r} \in P'$, where $A_{l|r}, B_{l|x}, C_{x|r} \in v'$. Then, there exists a derivation $v' \Rightarrow w'\,[p']$ in $G'$, where $\gamma(w') = w$.

(III) Let $p\colon AB \to CD \in P$, for some $A, B, C, D \in N$. By the assumption stated in Theorem 1, $A$ and $B$ occur in two neighboring paths denoted by $\alpha$ and $\beta$, respectively. Without any loss of generality, suppose that a context of $\alpha$ and $\beta$ is a string $c \in (P_{cs} \cup \varepsilon)^u$, where $c = pcd$, and $l$ is a left descendant context, $r$ is a right descendant context of $A$, $B$, respectively. By the construction of $G'$, there exist two rules
$$p'_l\colon A_{l|pcd} \to C_{l|cd},\;\; p'_r\colon B_{pcd|r} \to D_{cd|r} \in P',$$
where $A_{l|pcd}, C_{l|cd}, B_{pcd|r}, D_{cd|r} \in V'$. Then, there exists a derivation $v' \Rightarrow^2 w'\,[p'_l p'_r]$ in $G'$, where $\gamma(w') = w$.

Notice (III). The preservation of the context is achieved by nonterminal symbols. Since the stored context is reduced symbol by symbol from left to right direction in both $\alpha$ and $\beta$, $G'$ simulates the applications of non-context-free rules of $G$.

We covered all possible forms of $p$, so the claim holds. □

**Claim 2.** *Every $x \in L(G')$ can be derived in $G'$ as follows.*
$$S_{\varepsilon|\varepsilon} = x_0 \Rightarrow^{d_1} x_1 \Rightarrow^{d_2} x_2 \Rightarrow^{d_3} \cdots \Rightarrow^{d_{h-1}} x_{h-1} \Rightarrow^{d_h} x_h = x,$$
*for some $h \geq 0$, where $d_i \in \{1, 2\}$, $1 \leq i \leq h$, so that*

1. *if $d_i = 1$, then $x_{i-1} = uA_{l|r}v$, $x_i = uzv$, $x_{i-1} \Rightarrow x_i\,[A_{l|r} \to z]$, where $u, v \in V'^*$, $z \in \{E_{l|r}, B_{l|r}, C_{l|x}D_{x|r}, x, y\}$, for some $A_{l|r}, B_{l|r}, C_{l|x}, D_{x|r} \in N'$, $E_{l|r} \in (N' \cup \{\varepsilon\})$, $x, y \in T^*$;*

2. *if $d_i = 2$, then $x_{i-1} = uA_{x|py}B_{py|z}v$, $x_i = uC_{x|y}D_{y|z}v$, and*
$$uA_{x|py}B_{py|z}v \Rightarrow uC_{x|y}B_{py|z}v\;[A_{x|py} \to C_{x|y}] \Rightarrow uC_{x|y}D_{y|z}v\;[B_{py|z} \to D_{y|z}],$$
*for some $u, v \in V'^*$ and $A_{x|py}, B_{py|z}, C_{x|y}, D_{y|z} \in N'$.*

*Proof.* Since $G'$ is context-free, without any loss of generality in every derivation of $G'$ we can always reorder applied rules to satisfy Claim 2. □

**Claim 3.** *Let $S_{\varepsilon|\varepsilon} \Rightarrow^{d_1} x_1 \Rightarrow^{d_2} \cdots \Rightarrow^{d_{m-1}} x_{m-1} \Rightarrow^{d_m} x_m$ in $G'$ be a derivation that satisfies Claim 2, for some $m \geq 0$. Then, $S \Rightarrow^* w$ in $G$, where $\gamma(x_m) = w$.*

*Proof.* We prove this by induction on $m \geq 0$.

*Basis.* Let $m = 0$. That is $S_{\varepsilon|\varepsilon} \Rightarrow^0 S_{\varepsilon|\varepsilon}$ in $G'$. Clearly, $S \Rightarrow^0 S$ in $G$. Since $\gamma(S_{\varepsilon|\varepsilon}) = S$, the basis holds.

*Induction Hypothesis.* Suppose that there exists $n \geq 0$ such that Claim 3 holds for all $0 \leq m \leq n$.

*Induction Step.* Let $S_{\varepsilon|\varepsilon} \Rightarrow^{d_1} x_1 \Rightarrow^{d_2} \cdots \Rightarrow^{d_{n-1}} x_{n-1} \Rightarrow^{d_n} x_n \Rightarrow^{d_{n+1}} x_{n+1}$ in $G'$ be a derivation that satisfies Claim 2. By the induction hypothesis, $S \Rightarrow^* v$, $v \in V^*$, where $\gamma(x_n) = v$, in $G$. Divide the proof into two parts according to $d_{n+1}$.

(A) Let $d_{n+1} = 1$. By the construction of $G'$, there exists a rule $p' \in P'$ such that $x_n \Rightarrow^{d_{n+1}} x_{n+1}\ [p']$. Next, we consider the following two forms of $p'$.

 (I) Let $p'\colon A_{l|r} \to xE_{l|r}y$ or $p'\colon A_{\varepsilon|\varepsilon} \to xy \in P'$, for some $A \in N$, $E \in N$, $x, y \in T^*$ and $l, r \in (P_{cs} \cup \{\varepsilon\})^u$. By the construction of $G'$, rule $p'$ was introduced by some rule $p\colon A \to xEy \in P$ or $p\colon A \to xy \in P$, respectively. Then, there exists a derivation $v \Rightarrow w\,[p]$, where $\gamma(x_{n+1}) = w$.

 (II) Let $p'\colon A_{l|r} \to B_{l|x}C_{x|r} \in P'$, for some $A, B, C \in N$ and $l, r, x \in (P_{cs} \cup \{\varepsilon\})^u$. By the construction of $G'$, rule $p'$ was introduced by some rule $p\colon A \to BC \in P$. Then, there exists a derivation $v \Rightarrow w\,[p]$, where $\gamma(x_{n+1}) = w$.

(B) Let $d_{n+1} = 2$. Then, $x_n \Rightarrow^{d_{n+1}} x_{n+1}$ is equivalent to

$$u_1 A_{x|py} B_{py|z} u_2 \Rightarrow u_1 C_{x|y} B_{py|z} u_2\ [p'_1] \Rightarrow u_1 C_{x|y} D_{y|z} u_2\ [p'_2],$$

where $x_n = u_1 A_{x|py} B_{py|z} u_2$, $x_{n+1} = u_1 C_{x|y} D_{y|z} u_2$, and

$$p'_1\colon A_{x|py} \to C_{x|y},\ p'_2\colon B_{py|z} \to D_{y|z} \in P',$$

for some $u_1, u_2 \in V'^*$ and $A_{x|py}$, $B_{py|z}$, $C_{x|y}$, $D_{y|z} \in N'$. By the construction of $G'$, rules $p'_1$ and $p'_2$ were introduced by some rule $p\colon AB \to CD \in P$, Then, there exists a derivation $v \Rightarrow w\,[p]$, where $\gamma(x_{n+1}) = w$.

We covered all possibilities, so the claim holds. □

Observe that respective the derivation trees of the constructed context-free $G'$ remain slow-branching.

**Claim 4.** *The grammar $G'$ is k-linear.*

*Proof.* In construction (III) we replace the rules of the form $AB \to CD$ with the rules of the form $A \to B$, where $A, B, C, D \in N$. Therefore, only the rules that are allowed to occur in the derivation $G'$ before the rules of the form $A \to BC$ are the rules of the form $A \to B$. Rules of the form $A \to B$ before the rules of the form $A \to BC$ can be omitted by the trivial transformation of $G'$, similar to the algorithm on elimination of unit productions from Section 5 in [11]. Therefore, the grammar $G'$ is $k$-linear. □

By Claim 4 $G'$ is $k$-linear. By Claims 1 and 3, $S \Rightarrow^* w$ in $G$ iff $S_{\varepsilon|\varepsilon} \Rightarrow^* w'$ in $G'$, where $\gamma(w') = w$. If $S \Rightarrow^* w$ in $G$ and $w \in T^*$, then $w \in L(G)$. Since $\gamma(w') = w' = w$, for $w \in T^*$, $w' \in L(G')$. Therefore, $L(G) = L(G')$ and Theorem 1 hold. □

Consider Theorem 1. Observe that the 2nd condition is superfluous whenever $G$ is propagating.

**Theorem 2.** *A language $L$ is $k$-linear iff there is a constant $k \geq 0$, constant $u \geq 0$ and a propagating linear core general grammar $G$ such that $L = L(G)$ and for every $x \in L(G)$, there is a slow-branching tree of degree $k$ $\triangle_x \in {}_G\blacktriangle$, where any two nonterminal neighboring paths contain no more than $u$ pairs of context-dependent nodes.*

*Proof.* Prove this by analogy with the proof of Theorem 1. □

**Theorem 3.** *A language $L$ is regular iff there is a constant $u \geq 0$ and a left linear core general grammar $G$ such that $L = L(G)$ and for every $x \in L(G)$, there is a tree $\triangle_x \in {}_G\blacktriangle$ that satisfies:*

1. *any two nonterminal neighboring paths contain no more than $u$ pairs of context-dependent nodes;*
2. *out of neighboring paths, any pair of nodes is context-independent.*

*Proof.* Prove this by analogy with the proof of Theorem 1. □

**Theorem 4.** *A language $L$ is regular iff there is a constant $u \geq 0$ and a propagating left linear core general grammar $G$ such that $L = L(G)$ and for every $x \in L(G)$, there is a tree $\triangle_x \in {}_G\blacktriangle$, where any two nonterminal neighboring paths contain no more than $u$ pairs of context-dependent nodes.*

*Proof.* Prove this by analogy with the proof of Theorem 1. □

## 5 Use

In this section, we explain how to apply the results achieved in the previous section in order to demonstrate the metalinearness (or regularity) of a language, $L$. As a rule, this demonstration follows the next three-step proof scheme for metalinearness.

1. Construct a linear core GG $G$.
2. Prove $L(G) = L$.
3. Prove that $G$ satisfies conditions from Theorem 2 or Theorem 1 depending on whether $G$ is context-sensitive.

For regularity, we use a similar three-step scheme as following.

1. Construct a left linear core GG $G$.
2. Prove $L(G) = L$.
3. Prove that $G$ satisfies conditions from Theorem 3 or Theorem 4 depending on whether $G$ is context-sensitive.

Reconsider the grammar $G$ from Example 2. Following the proof scheme sketched above, we next prove that $L(G)$ is $k$-linear. Without any loss of generality, every terminal derivation of $G$ can be divided into the following 5 phases, where each rule may be used only in a specific phase:

(a) 1–4 (b) 5–9 (c) 10–12 (d) 13–17 (e) 18–22

Next, we describe these phases in greater detail.

(a) First, we generate the following string by rules 1 though 4.

$$A_1A_2BC_1C_2$$

Possibly applicable rules from (b) and (c) may be postponed to the next phases without affecting the derivation, since the rules in the previous phases cannot rewrite the nonterminals of the following phases.

(b) The rules (5) through (9) are context-free rules and nonterminals on the left-hand side of the rule are the same as on the right-hand side of the rule. Therefore, they are grouped into (b), since they only generate terminals. Possibly applicable rules from (c) may be postponed for the phase (c) without affecting the derivation since the rules in the previous phases cannot rewrite nonterminals from the following phases.

$$a^*A_1A_2a^*b^*Bc^*a^*C_1C_2b^*.$$

(c) The rules 10 and 12 are non context-free rules. The rules 10 through 12 are all rules without generating terminals. For the same reason as in (a) rules 1 to 4 from the phases (d) and (e) can be postponed to respective phases.

$$a^*D_1D_2a^*b^*Ec^*a^*F_1F_2b^*.$$

(d) The rules 13 through 17 are alike rules in (b)

$$a^*0^*D_1D_21^*a^*b^*0^*E1^*c^*a^*0^*F_1F_21^*b^*.$$

(e) Since rules 18 and 22 are erasing rules and they can always be postponed until the end of any successful derivation.

$$a^*0^*1^*a^*b^*0^*1^*c^*a^*0^*1^*b^*.$$

Grammar $G$ is obviously a linear core GG.

Only rules in the step (a) include branching of nonterminals, no terminals are generated and the branching in the step (a) is a slow-branching since the degree derivation tree is 4 and, therefore, $u$ is always 4. Therefore, the slow-branching condition is fulfilled.

Let us now show that for any $x \in L(G)$, there is ${}_G\triangle_x \in {}_G\blacktriangle$, where any two neighboring paths contain no more than a one pair of context-dependent nodes.

Every pair of context-dependent nodes in ${}_G\triangle_x$ corresponds to one non-context-free rule in $S \Rightarrow^* x$. Consider the five phases sketched above. Observe that all phases except (c) contain only non context-free rules, so we only have to investigate (c). On the other hand, (c) contain no rule of the form $A \to BC$, thus the number of neighboring paths remains unchanged.

In (c) rule 10 and 12 introduce context dependency between two pairs of neighboring paths. After the application of these two rules, we cannot reach the nonterminals again on the left-hand side of rules 10 and 12. Therefore, these context-dependencies can occur only once between a pair of neighboring paths.

No other non-context-free rule is applied; therefore, no other context-dependent pair of nodes can occur. Then, every pair of neighboring paths may contain at most one context-dependent pair of nodes introduced in phase (c).

Since $G$ is a linear core GG, where for every $x \in L(G)$, there is ${}_G\triangle_x \in {}_G\blacktriangle$, where any two neighboring paths contain no more than one pair of context-dependent nodes, by Theorem 1, $L(G)$ is $k$-linear.

Unfortunately, although we are able to transform any GG into KNF and, that is, linear core GG, the question whether the conditions in Theorems 1 through 4 are satisfied is obviously undecidable. To invent an algorithm that gives at least approximate results is part of a future research.

## 6 Final Remarks and Open Problems

Before closing this paper, we bring the reader's attention to an open question. More specifically, consider a more lenient definition of slow-branching tree as follows.

**Definition 7.** *A labeled ordered tree $t$ is* slow-branching *if any of its pairs of nonterminal neighboring paths contains no more than two nonterminal nodes having two nonterminal children. A slow-branching labeled ordered tree is of* degree *$k$ if it contains $k$ branching nonterminal nodes, $k \geq 1$.*

It is obvious that the newly provided Definition 7 is insufficient to prove that a grammar restricted by a slow-branching derivation tree is $k$-linear. However, it is possible to apply different restrictions to Definition 7 with its own advantages or demonstrate similar result to Theorem 1 to prove that it is $k$-linear. Such a discovery would require further studies.

## Acknowledgments

This work was supported by the BUT grant FIT-S-23-8209.

## References

[1] Alfred Aho & Jeffrey Ullman (1972): *The Theory of Parsing, Translation, and Compiling*. Prentice-Hall, Series in Automatic Computation.

[2] Brenda Baker (1974): *Non-context-free grammars generating context-free languages*. Information and Control 24(3), pp. 231–246, doi:10.1016/S0019-9958(74)80038-0.

[3] Ronald Vernon Book (1972): *Terminal Context in Context-Sensitive Grammars*. SIAM J. Comput. 1(1), p. 20–30, doi:10.1137/0201003.

[4] Ronald Vernon Book (1973): *On the structure of context-sensitive grammars*. International Journal of Computer & Information Sciences 2, p. 129–139, doi:10.1007/BF00976059.

[5] Thomas Cormen, Charles Leiserson & Ronald Rivest (2002): *Introduction to Algorithms*. McGraw-Hill.

[6] Andrzej Ehrenfeucht, David Haussler & Grzegorz Rozenberg (1983): *On regularity of context-free languages*. Theoretical Computer Science 27(3), pp. 311–332, doi:10.1016/0304-3975(82)90124-4.

[7] Michael Harrison (1978): *Introduction to Formal Language Theory*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[8] Sige-Yuki Kuroda (1964): *Classes of languages and linear-bounded automata*. Information and Control 7(2), pp. 207–223, doi:10.1016/S0019-9958(64)90120-2.

[9] Gethin Matthews (1964): *A note on asymmetry in phrase structure grammars.* Information and Control 7(3), pp. 360–365, doi:10.1016/S0019-9958(64)90406-1.

[10] Gethin Matthews (1967): *Two-way languages.* Information and Control 10(2), pp. 111–119, doi:10.1016/S0019-9958(67)80001-9.

[11] Alexander Meduna (2000): *Automata and languages: theory and applications.* Springer-Verlag, Berlin, Heidelberg, doi:10.1007/978-1-4471-0501-5.

[12] Alexander Meduna (2014): *Formal Languages and Computation: Models and Their Applications.* Taylor & Francis, New York, doi:10.1201/b16376.

[13] Alexander Meduna & Ondřej Soukup (2017): *Modern Language Models and Computation: Theory with Applications.* Springer US, doi:10.1007/978-3-319-63100-4.

[14] Grzegorz Rozenberg & Arto Salomaa, editors (1997): *Handbook of Formal Languages.* Springer, doi:10.1007/978-3-642-59136-5.

[15] Arto Salomaa (1973): *Formal Languages.* Academic Press, London.

# Non-Global Parikh Tree Automata

Luisa Herrmann
Computational Logic Group, TU Dresden, Germany
ScaDS.AI Center for Scalable Data Analytics and Artificial Intelligence Dresden/Leipzig, Germany
luisa.herrmann@tu-dresden.de

Johannes Osterholzer
secunet Security Networks AG, Germany
johannes.osterholzer@gmail.com

Parikh (tree) automata are an expressive and yet computationally well-behaved extension of finite automata – they allow to increment a number of counters during their computations, which are finally tested by a semilinear constraint. In this work, we introduce and investigate a new perspective on Parikh tree automata (PTA): instead of testing one counter configuration that results from the whole input tree, we implement a non-global automaton model. Here, we copy and distribute the current configuration at each node to all its children, incrementing the counters pathwise, and check the arithmetic constraint at each leaf. We obtain that the classes of tree languages recognizable by global PTA and non-global PTA are incomparable. In contrast to global PTA, the non-emptiness problem is undecidable for non-global PTA if we allow the automata to work with at least three counters, whereas the membership problem stays decidable. However, for a restriction of the model, where counter configurations are passed in a linear fashion to at most one child node, we can prove decidability of the non-emptiness problem.

## 1 Introduction

Finite automata are one of the most fundamental computation models in theoretical computer science and have been generalized to many structures that go beyond words, such as trees [18]. However, they are not sufficient when arithmetic properties (such as two symbols occurring equally often) have to be ensured. For this reason, there are numerous approaches to extending automata with counting mechanisms. In the area of tree automata, although less studied than extensions of word automata, there are (among others) two approaches that come into consideration: On the one hand, there are *pushdown tree automata* [7], which extend pushdown automata to trees and thus recognize context-free tree languages, as well as their restriction *counter tree automata*. And on the other hand, Parikh tree automata [14, 13] have been considered: during their computations, they allow to increment a number of counters in each step. These counters are finally tested to satisfy a semilinear constraint. The calculation principles for the counting mechanisms work orthogonally in both approaches – while pushdown tree automata split their computations at each node and execute them pathwise, Parikh tree automata allow a global view: their counters are incremented over the whole input tree before their membership in a semilinear set is tested. Thus, in the remaining work we will refer to this model as *global Parikh tree automata* (GPTA).

One motivation for the development and investigation of Parikh automata in recent years is the specification and verification of systems that fall outside the scope of regular languages. For such applications, tree automata are also interesting, as they are more suitable to model non-determinism and parallel processes than word automata. However, we think that a non-global view would be interesting for this case in particular: with GPTA, requirements such as "the same arithmetic property applies in every alternative path" cannot be modeled.

For this reason and inspired by the computation strategy of pushdown tree automata, we introduce here an alternative, non-global definition of Parikh tree automata (PTA): we define a model which copies and distributes the current counter configuration at each node to all its children, thus increments the counters pathwise, and finally checks at each leaf node whether the obtained configuration is contained in a semilinear set. In this way, PTAs are able to test arithmetic properties of tree paths.

**Contributions** In this work, we start the investigation of non-global PTA, especially from the perspective of their expressiveness and decidability:

- We generalize GPTA, which have so far only been considered for complete binary trees, to trees over arbitrary ranked alphabets and prove an exchange lemma (Lemma 3): This lemma, originally shown for Parikh word automata [1, Lemma 1], states that certain parts in computations of GPTA can be rearranged. Thus, it can be used to show the limits of their expressive power.
- The exchange lemma is used to show that there are languages that are recognized by non-global PTA but not by GPTA. The converse of this statement is also shown, and thus we obtain that the language classes of these two models are incomparable (Theorem 1).
- We prove that, in contrast to GPTA, non-emptiness is undecidable for PTA with at least three counters (Theorem 2). This follows from a simulation of the computations of two counter machines. On the other hand, membership is decidable for arbitrary PTA (Theorem 4).
- We introduce a restriction on the computation mechanism for PTA: *linear* PTA may at each node only pass the current counter configuration to one child tree, the computations in all other child trees start again with all counters zero. With this restriction, we can limit the number of such "non-reset" paths that must at least occur if the language of a linear PTA is non-empty (Lemma 5). Thereby, we can show that non-emptiness becomes decidable (Theorem 3).

**Related work** Since their introduction in [14, 15], Parikh automata have been studied in many works from a variety of perspectives, cf. for example [1, 2]; recently there have also been extensions for infinite words [8, 6] and infinite trees [11]. It is known that Parikh automata correspond to a special form of *vector addition systems with states* (VASS) over integers, so-called $\mathbb{Z}$-VASS [9]. For VASS, in [3] a definition for alternation was provided by using branching – in this sense, PTA could be seen as a formulation of *alternating $\mathbb{Z}$-VASS*.

The idea of limiting the counter flow in a computation to linear paths originates from *linear pushdown tree automata* [4] and has later also been extended to *tree automata with storage* [10].

## 2 Preliminaries

We denote by $\mathbb{N}$ the set of *natural numbers* including 0 and set $[n] = \{1,\dots,n\}$ for each $n \in \mathbb{N}$. Given a finite set $A$, we denote its *cardinality*, i.e., the number of its elements, by $|A|$. For each $k \in \mathbb{N}$ and $a_1,\dots,a_k \in A$, we call $w = a_1 \dots a_k$ a *word over* $A$ and say that its *length* is $k$. We let $A^k$ be the set of all words over $A$ of length $k$ and set $A^* = \bigcup_{n\in\mathbb{N}} A^k$. Given some $w \in A^*$, we refer to its length by $|w|$ and the word of length 0 will be denoted by $\varepsilon$. We let $\sqsubseteq$ denote the *prefix order* on $A$: for words $w_1, w_2 \in A$ we have $w_1 \sqsubseteq w_2$ if $w_2 = w_1 u$ for some $u \in A^*$. The *lexicographic order* on $\mathbb{N}^*$ will be denoted by $\leq_{\text{lex}}$, and is defined for every $u$, $v \in \mathbb{N}^*$ such that $u \leq_{\text{lex}} v$ whenever *(i)* either $u \sqsubseteq v$, or *(ii)* there are $x$, $y$, $z \in \mathbb{N}^*$ and $n$, $m \in \mathbb{N}$ such that $u = xny$, $v = xmz$ and $n < m$.

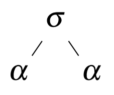
**Alphabets, trees, and tree languages.** A *ranked set* is a tuple $(\Sigma, \mathrm{rk})$ where $\Sigma$ is a set (its elements called *symbols* or *labels*) and $\mathrm{rk}\colon \Sigma \to \mathbb{N}$ is a function assigning to each symbol in $\Sigma$ a natural number, its *rank*. We often assume rk implicitly and only write $\Sigma$ instead of $(\Sigma, \mathrm{rk})$. For each $n \in \mathbb{N}$, by $\Sigma^{(n)}$ we mean $\mathrm{rk}^{-1}(n)$ and we write $\sigma^{(n)}$ in order to say that $\sigma \in \Sigma^{(n)}$. We say that a ranked set $(\Sigma, \mathrm{rk})$ is a *ranked alphabet* if the set $\Sigma$ is finite.[1]

Now let $\Sigma$ be a ranked set and $H$ a set. The set $T_\Sigma(H)$ of *trees* (over $\Sigma$ and indexed by $H$) is defined to be the smallest set $T$ such that *(i)* $H \subseteq T$ and *(ii)* for each $n \in \mathbb{N}$, $\sigma \in \Sigma^{(n)}$, and $\xi_1, \dots, \xi_n \in T$ we have $\sigma(\xi_1, \dots, \xi_n) \in T$. If $H = \emptyset$, we simply write $T_\Sigma$ instead of $T_\Sigma(H)$. As usual, we denote the tree $\alpha()$ by $\alpha$ for each $\alpha \in \Sigma^{(0)}$ and we often write monadic trees of the form $\gamma_1(\gamma_2(\dots\gamma_k(\#)\dots))$, where $\gamma_1, \dots, \gamma_k \in \Sigma^{(1)}$, as words $\gamma_1 \dots \gamma_k\#$. Each subset $L \subseteq T_\Sigma$ is called a *tree language*.

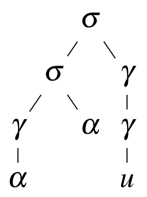
Let $\xi$, $\zeta \in T_\Sigma(H)$. We let $\mathrm{pos}(\xi) \subseteq \mathbb{N}^*$ denote the *set of positions* of $\xi$, defined in the usual way: for every $\alpha \in \Sigma^{(0)} \cup H$ we let $\mathrm{pos}(\alpha) = \{\varepsilon\}$ and for every $n \geq 1$, $\sigma \in \Sigma^{(n)}$, and $\xi_1, \dots, \xi_n \in T_\Sigma(H)$ we let $\mathrm{pos}(\sigma(\xi_1, \dots, \xi_n)) = \{\varepsilon\} \cup \{i\rho \mid i \in [n], \rho \in \mathrm{pos}(\xi_i)\}$. Furthermore, $|\xi| = |\mathrm{pos}(\xi)|$ stands for the *size* of $\xi$, and, given a position $\rho \in \mathrm{pos}(\xi)$, we denote by $\xi(\rho)$ the *label of $\xi$ at position $\rho$* and by $\xi_{|\rho}$ the *subtree of $\xi$ at position $\rho$*, respectively. Let $\xi[\zeta]_\rho$ designate the tree that results from $\xi$ by replacing the subtree rooted at $\rho$ by $\zeta$. We let $\mathrm{ht}(\xi) = \max\{|\rho| \mid \rho \in \mathrm{pos}(\xi)\}$ and $\mathrm{sub}(\xi) = \{\xi_{|\rho} \mid \rho \in \mathrm{pos}(\xi)\}$. Given positions $\rho_1, \rho_2 \in \mathrm{pos}(\xi)$ we say that the subtrees $\xi_{|\rho_1}$ and $\xi_{|\rho_2}$ are *independent* if $\rho_1 \not\sqsubseteq \rho_2$ and $\rho_2 \not\sqsubseteq \rho_1$.

A *path* $\pi$ is a sequence $\pi = \rho_1 \dots \rho_n$ of positions $\rho_1, \dots, \rho_n \in \mathrm{pos}(\xi)$ such that for each $i \in [n-1]$ we have $\rho_{i+1} = \rho_i k$ for some $k \in [\mathrm{rk}(\xi(\rho_i))]$. The *path word* of $\pi$ is given by $\xi(\pi) = \xi(\rho_1) \dots \xi(\rho_n)$. We say that $\pi$ is a *complete path* (or c-path) if $\rho_1 = \varepsilon$ and $\xi(\rho_n) \in \Sigma^{(0)}$; the set of all complete paths of $\xi$ is denoted by $\mathrm{paths}(\xi)$.

**Example 1.** Consider the ranked alphabet $\Sigma = \{\sigma^{(2)}, \gamma^{1}, \alpha^{(0)}\}$ as well as the set $H = \{u\}$. Then $\sigma(\alpha, \alpha)$ is a tree in $T_\Sigma$ and $\xi = \sigma(\sigma(\gamma(\alpha), \alpha), \gamma(\gamma(u)))$ is a tree in $T_\Sigma(H)$. As mentioned above, we will also sometimes write $\sigma(\sigma(\gamma\alpha, \alpha), \gamma\gamma u)$ for $\xi$. We have $\mathrm{pos}(\xi) = \{\varepsilon, 1, 11, 111, 12, 2, 21, 211\}$, $\xi(\varepsilon) = \sigma$ and $\xi(21) = \gamma$, and $\xi_{|2} = \gamma(\gamma(u))$. The trees in this example can be represented graphically as



and



,

respectively. ◁

**Contexts, spines, and composition.** Let $X = \{x_1, x_2, \dots\}$ be a fixed set of *variables* that is disjoint from every other set in this work and let $X_n = \{x_1, \dots, x_n\}$. Now let $H$ be a set, $k \geq 1$ and $\xi \in T_\Sigma(H \cup X_k)$. We call $\xi$ a *context* if *(a)* there is exactly one $\rho_i \in \mathrm{pos}(\xi)$ (in the further denoted by $\mathrm{pos}_{x_i}(\xi)$) with $\xi(\rho_i) = x_i$ for each $i \in [k]$ and *(b)* for each $i_1, i_2 \in [k]$, if $i_1 < i_2$, then $\mathrm{pos}_{x_{i_1}}(\xi) \leq_{\mathrm{lex}} \mathrm{pos}_{x_{i_2}}(\xi)$. The set of all such *contexts over $\Sigma$ and $H$* will be denoted by $C_\Sigma(H, X_k)$ (or by $C_\Sigma(X_k)$ if $H = \emptyset$).

The *composition* $\zeta \cdot \xi$ of a context $\zeta \in C_\Sigma(H, X_1)$ and a tree $\xi \in T_\Sigma(H \cup X)$ replaces $x_1$ in $\zeta$ by $\xi$. This operation can be transferred to arbitrary $k \geq 2$, trees $\xi \in T_\Sigma(H \cup X_k)$ and $\xi_1, \dots, \xi_k \in T_\Sigma(H)$: we let $\xi[\xi_1, \dots, \xi_k]$ stand for the tree $\zeta$ obtained from $\xi$ by replacing each occurrence of $x_i$ by $\xi_i$ for each $i \in [k]$.

[1]Most tree languages in this paper will have labels from some finite ranked alphabet. However, we have to allow infinite label sets for one definition.

Given a tree $\xi \in T_\Sigma$ and a path $\rho_1 \dots \rho_n$, we let the $(\rho_1, \rho_n)$-*spine of* $\xi$, denoted by $\xi^{[\rho_1,\rho_n]}$, be the context $\zeta \in C_\Sigma(X_k)$ for some $k \in \mathbb{N}$ containing exactly the path $\rho_1 \dots \rho_n$, i.e., such that there are a context $\zeta' \in C_\Sigma(X_1)$ and trees $\xi_1, \dots, \xi_k \in T_\Sigma$ with $\xi = \zeta' \cdot (\zeta[\xi_1, \dots, \xi_k])$ and for each $\rho \in \mathbb{N}^*$ we have $\rho \in \mathrm{pos}(\zeta) \setminus \{\mathrm{pos}_{x_i}(\zeta) \mid i \in [k]\}$ if and only if $\rho_1 \rho = \rho_j$ for some $j \in [n]$.

**Example 2.** Consider the positions $\rho_1 = 1$ and $\rho_2 = 11$ of the tree $\xi$ from Example 1. Then the $(\rho_1, \rho_2)$-spine of $\xi$ is $\xi^{[\rho_1,\rho_2]} = \sigma(\gamma(x_1), x_2)$, and we have $\xi = \sigma(x_1, \gamma\gamma u) \cdot (\sigma(\gamma(x_1), x_2)[\alpha, \alpha])$. $\lhd$

**Semilinear sets.** Let $s \geq 1$. We denote by $\mathbf{0}_s$ the zero-vector $\mathbf{0}_s = (0,...,0) \in \mathbb{N}^s$ of *dimension* $s$. If $s$ is clear from the context, we often only write $\mathbf{0}$. A set $C \subseteq \mathbb{N}^s$, $s \geq 1$, is *linear* if it is of the form $C = \{d_0 + \sum_{i \in [l]} m_i d_i \mid m_1, \dots, m_l \in \mathbb{N}\}$ for some $l \in \mathbb{N}$ and vectors $d_0, \dots, d_l \in \mathbb{N}^s$. Any finite union of linear sets is called *semilinear*.

**Lemma 1** ([5, Theorem 1.2 and 1.3])**.** *Given a semilinear set $C \subseteq \mathbb{N}^s$ and a vector $d \in \mathbb{N}^s$, it is decidable whether $d \in C$.*

**Parikh string automata** Let $s \geq 1$. A *Parikh string automaton* of dimension $s$ ($s$-PA) is a tuple $\mathscr{A} = (Q, \Sigma, q_0, \Delta, F, C)$ where $Q$ is a finite set of states, $\Sigma$ is a (string) alphabet, $q_0 \in Q$ (initial state), $F \subseteq Q$ (final states), $\Delta$ is a finite set of transitions of the form $(q, a, d, q')$ for $q, q' \in Q$, $a \in \Sigma$, $d \in \mathbb{N}^s$, and $C$ is a semi-linear set over $\mathbb{N}^s$. Let $w \in \Sigma^*$. A *run* of $\mathscr{A}$ on $w$ is a sequence

$$(p_0, a_1, d_1, p_1)(p_1, a_2, d_2, p_2) \dots (p_{n-1}, a_n, d_n, p_n)$$

of transitions such that $p_0 = q_0$, $a_1 \dots a_n = w$, $p_n \in F$, and $(d_1 + \dots + d_n) \in C$. The set of all runs of $\mathscr{A}$ on $w$ is denoted $\mathrm{Run}_{\mathscr{A}}(w)$ and the *language recognized by* $\mathscr{A}$ is the set $\mathscr{L}(\mathscr{A}) = \{w \in \Sigma^* \mid \mathrm{Run}_{\mathscr{A}}(w) \neq \emptyset\}$.

**Lemma 2** ([15, Property 6])**.** *Given a Parikh string automaton $\mathscr{A}$, it is decidable whether $\mathscr{L}(\mathscr{A}) \neq \emptyset$.*

## 3 Global Parikh Tree Automata

Let us recall the definition of (global) Parikh tree automata from [14, 13]. Note that we use here a slight variation of the original version: In [14, 13], only full binary trees were considered and, thus, the number of successor states in each transition was fixed to two (also for leaf nodes). In this paper, we extend Parikh tree automata to arbitrary ranked trees in the usual way – it is not hard to see that for alphabets containing only binary and nullary symbols, both formalisms are equivalent.

**Extended Parikh map.** Given a ranked alphabet $\Sigma$ and some finite $D \subseteq \mathbb{N}^s$ for $s \geq 1$, the automaton model works with symbols from $\Sigma \times D$. Thus, we use the *projections* $\cdot_\Sigma : \Sigma \times D \to \Sigma$ with $(a,d)_\Sigma = a$ and $\cdot_D : \Sigma \times D \to D$ with $(a,d)_D = d$, extended to trees in the obvious way. Moreover, the *extended Parikh map* $\Psi\colon T_{\Sigma \times D} \to \mathbb{N}^s$ is defined for each tree $\xi \in T_{\Sigma \times D}$ by $\Psi(\xi) = \sum_{\rho \in \mathrm{pos}(\xi)} (\xi(\rho))_D$.

**Global Parikh tree automata.** Let $m \geq 1$. A *global Parikh tree automaton* of dimension $m$ ($m$-GPTA) is a tuple $\mathscr{A} = (Q, \Sigma, D, q_0, \Delta, C)$ where $Q$ is a finite set of states, $\Sigma$ is a ranked alphabet, $D \subset \mathbb{N}^m$ is finite, $q_0 \in Q$ is the initial state, $C \subseteq \mathbb{N}^m$ is a semilinear set, and $\Delta$ is a finite set of transitions of the form

$$q \to \langle \sigma, d \rangle (q_1, \dots, q_n)$$

where $n \in \mathbb{N}$, $\sigma \in \Sigma^{(n)}$, $d \in D$, and $q, q_1, \dots, q_n \in Q$.

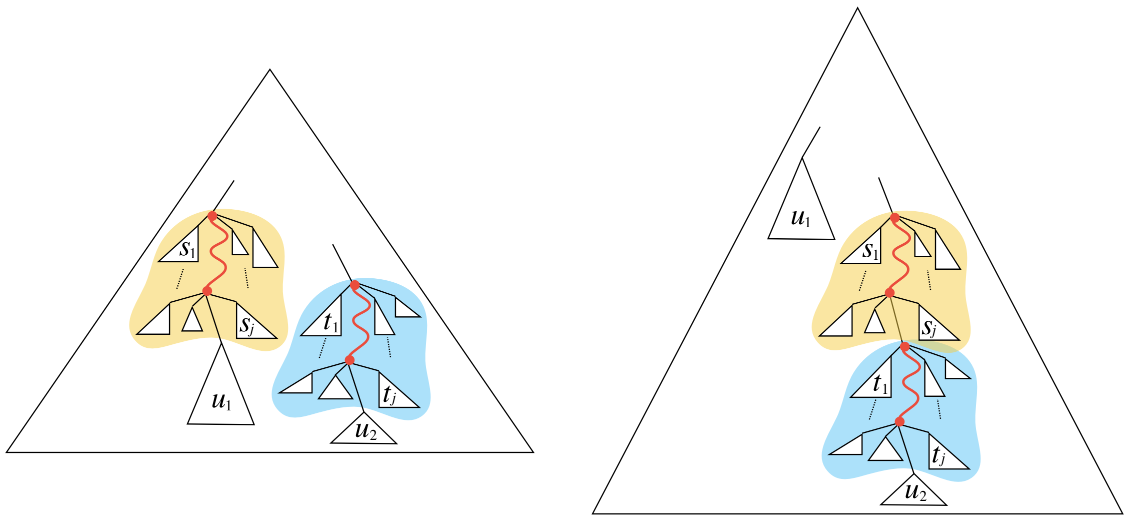


Figure 1: The tree $\xi$ divided as in Lemma 3 (1.) and its reordering as in (2.) where the red spine corresponds to $\zeta_2$.

Given a tree $\zeta \in T_{\Sigma \times D}$, a *run of $\mathscr{A}$ on $\zeta$* is a mapping $r\colon \mathrm{pos}(\zeta) \to Q$ such that for each $\rho \in \mathrm{pos}(\zeta)$, $r(\rho) \to \zeta(\rho)(r(\rho 1), \ldots, r(\rho n))$ with $n = \mathrm{rk}(\zeta(\rho))$ is in $\Delta$. We say that a run $r$ is *successful* if $r(\varepsilon) = q_0$ and $\Psi(\zeta) \in C$; we denote the set of all successful runs of $\mathscr{A}$ on $\zeta$ by $\mathrm{Run}_{\mathscr{A}}(\zeta)$. Then the *language of $\mathscr{A}$*, denoted by $\mathscr{L}(\mathscr{A})$ is the set $\mathscr{L}(\mathscr{A}) = \{\xi \in T_\Sigma \mid \exists \zeta \in T_{\Sigma \times D}$ with $(\zeta)_\Sigma = \xi$ and $\mathrm{Run}_{\mathscr{A}}(\zeta) \neq \emptyset\}$.

## 3.1 Pumping-style Exchange Lemma for GPTA

For Parikh automata, a classical pumping lemma that cuts out or iterates parts of a computation is not known – missing or additional parts in a computation would change the extended Parikh image and thus affect acceptance. However, since the final counter configuration is a global result of the entire computation, parts of the computation can be rearranged without changing the extended Parikh image. This was shown in [1, Lemma 1] for the string case and is generalized here to the tree case. This result will be useful later to show that certain tree languages recognizable by non-global Parikh tree automata are not GPTA-recognizable. Note that a crucial part of the extension is that computation parts from independent subtrees are reordered. This allows us to distinguish path counting from global counting using the exchange lemma. Figure 1 is a graphical representation of the following lemma.

**Lemma 3.** *Let $L$ be a GPTA-recognizable tree language. Then there exist constants $l, p > 0$ such that for each tree $\xi \in L$ with at least $l$ pairwise independent subtrees of height at least $p$ there exists $k \geq 0$, contexts $\zeta_1 \in C_\Sigma(X_2), \zeta_2 \in C_\Sigma(X_{k+1})$ with $0 < \mathrm{ht}(\zeta_2) < p$, trees $s_1, \ldots, s_k$, $t_1, \ldots, t_k$, $u_1, u_2 \in T_\Sigma$, and $j \in [k+1]$ such that*

1. $\xi = \zeta_1[\zeta_2[s_1, \ldots, s_{j-1}, x_1, s_j, \ldots, s_k] \cdot u_1, \zeta_2[t_1, \ldots, t_{j-1}, x_1, t_j, \ldots, t_k] \cdot u_2]$,
2. $\zeta_1[u_1, \zeta_2[s_1, \ldots, s_{j-1}, x_1, s_j, \ldots, s_k] \cdot \zeta_2[t_1, \ldots, t_{j-1}, x_1, t_j, \ldots, t_k] \cdot u_2] \in L$*, and*
3. $\zeta_1[\zeta_2[s_1, \ldots, s_{j-1}, x_1, s_j, \ldots, s_k] \cdot \zeta_2[t_1, \ldots, t_{j-1}, x_1, t_j, \ldots, t_k] \cdot u_1, u_2] \in L$*.*

*Proof.* Let $\mathscr{A} = (Q,\Sigma,D,q_0,\Delta,C)$ be a GPTA with $\mathscr{L}(\mathscr{A}) = L$ and let $p = |Q|+1$. Further, let $N$ be the maximal rank of symbols in $\Sigma$. In order to define $l$, we build from the transitions of $\mathscr{A}$ a graph $G$ labeled by elements of $\Delta \times N$ as follows: We let $G = (V,E)$ with $V = Q$ and $E \subseteq Q \times (\Delta \times [N]) \times Q$ such that $(q,\langle\tau,i\rangle,q') \in E$ if and only if $\tau = q \to \langle\sigma,d\rangle(q_1,\dots,q_n)$, $i \le n$, and $q_i = q'$. Now let $l'$ be the number of cycles in $G$, i.e., the number of sequences $(f_0,u_1,f_1)(f_1,u_2,f_2)...(f_{k-1},u_k,f_k)$ for $k \le p$ such that $(f_{i-1},u_i,f_i) \in E$ for each $i \in [k]$, $f_0 = f_k$, and there are no $i,j \in [k]$ such that $i \ne j$ and $f_i = f_j$. Then $l = l'+1$.

Now consider $\xi \in L$ that fulfills the requirements of the statement. Then there exists a tree $t \in T_{\Sigma\times D}$ with $(t)_\Sigma = \xi$ and a successful run $r \in \mathrm{Run}_{\mathscr{A}}(t)$. By our requirement for $\xi$, also $t$ contains $l$ independent subtrees of height at least $p$. Note that because of their height, each of these subtrees contains a cycle. By the definition of $l$, we can apply the pigeonhole principle and obtain that there is a pair of paths that contain the same cycle: there has to be some $1 \le h \le p$ and two paths $\rho^1_1\dots\rho^1_h$ and $\rho^2_1\dots\rho^2_h$ in independent subtrees of $t$ that induce transition cycles which coincide. Formally, for each $i \in [2]$, let

$$w_i = (r(\rho^i_1),\tilde\tau(\rho^i_1),r(\rho^i_2))\dots(r(\rho^i_{h-1}),\tilde\tau(\rho^i_{h-1}),r(\rho^i_h)),$$

where $\tilde\tau(\rho^i_j) = \langle r(\rho^i_j) \to t(\rho^i_j)(r(\rho^i_j1),\dots,r(\rho^i_jn)),\mu\rangle$, and $\mu \in \mathbb{N}$ such that $\rho^i_{j+1} = \rho^i_j\mu$. Both $w_1$ and $w_2$ are cycles in $G$, thus $r(\rho^i_1) = r(\rho^i_h)$, and $w_1 = w_2$.

Now let $\zeta_1 \in C_\Sigma(X_2)$ such that $\xi = \zeta_1[\xi_{|\rho^1_1},\xi_{|\rho^2_1}]$ and let $\zeta_2 = \xi^{[\rho^1_1,\rho^1_{h-1}]} = \xi^{[\rho^2_1,\rho^2_{h-1}]}$. Clearly, there is some $k \in \mathbb{N}$, $j \in [k+1]$, and trees $s_1,\dots,s_k,t_1,\dots,t_k \in T_\Sigma$ such that $\xi_{|\rho^1_1},\xi_{|\rho^2_1}$ can be written as

$$\xi_{|\rho^1_1} = \zeta_2[s_1,\dots,s_{j-1},x_1,s_j,\dots,s_k]\cdot\xi_{|\rho^1_h} \qquad \text{and} \qquad \xi_{|\rho^2_1} = \zeta_2[t_1,\dots,t_{j-1},x_1,t_j,\dots,t_k]\cdot\xi_{|\rho^2_h}.$$

By letting $u_1 = \xi_{|\rho^1_h}$ and $u_2 = \xi_{|\rho^2_h}$ we obtain

$$\xi = \zeta_1[\zeta_2[s_1,\dots,s_{j-1},x_1,s_j,\dots,s_k]\cdot u_1,\zeta_2[t_1,\dots,t_{j-1},x_1,t_j,\dots,t_k]\cdot u_2]$$

which corresponds to item (1.) of the statement. Note that we can subdivide $t$ in exactly the same building blocks as $\xi$: there are $\delta_1 \in C_{\Sigma\times D}(X_2)$, $\delta_2 = t^{[\rho^1_1,\rho^1_{h-1}]}$, and $\hat s_1,\dots,\hat s_k,\hat t_1,\dots,\hat t_k$ such that $\mathrm{pos}(\delta_1) = \mathrm{pos}(\zeta_1)$, $\mathrm{pos}(\hat s_i) = \mathrm{pos}(s_i)$, $\mathrm{pos}(\hat t_i) = \mathrm{pos}(t_i)$ for each $i \in [k]$, and we have

$$t = \delta_1[\delta_2[\hat s_1,\dots,\hat s_{j-1},x_1,\hat s_j,\dots,\hat s_k]\cdot t_{|\rho^1_h},\delta_2[\hat t_1,\dots,\hat t_{j-1},x_1,\hat t_j,\dots,\hat t_k]\cdot t_{|\rho^2_h}].$$

For item (2.) we need to argue that the reordering

$$\xi' = \zeta_1[u_1,\zeta_2[s_1,\dots,s_{j-1},x_1,s_j,\dots,s_k]\cdot\zeta_2[t_1,\dots,t_{j-1},x_1,t_j,\dots,t_k]\cdot u_2]$$

of $\xi$ can be recognized by $\mathscr{A}$, too. To show this, we construct from $r$ a computation $r'$ on the corresponding reordering $t'$ of $t$ given by

$$t' = \delta_1[t_{|\rho^1_h},\delta_2[\hat s_1,\dots,\hat s_{j-1},x_1,\hat s_j,\dots,\hat s_k]\cdot\delta_2[\hat t_1,\dots,\hat t_{j-1},x_1,\hat t_j,\dots,\hat t_k]\cdot t_{|\rho^2_h}]$$

as follows:

- for each $\rho \in \mathrm{pos}(\delta_1)\setminus\{\mathrm{pos}_{x_1}(\delta_1),\mathrm{pos}_{x_2}(\delta_1)\}$ we let $r'(\rho) = r(\rho)$,
- for each $\rho \in \mathrm{pos}(t_{|\rho^1_h})$ we let $r'(\rho^1_1\rho) = r(\rho^1_h\rho)$,
- for each $\rho \in \mathrm{pos}(\delta_2[\hat s_1,\dots,\hat s_{j-1},x_1,\hat s_j,\dots,\hat s_k])\setminus\{\mathrm{pos}_{x_1}(\delta_2[\hat s_1,\dots,\hat s_{j-1},x_1,\hat s_j,\dots,\hat s_k])\}$ we let $r'(\rho^2_1\rho) = r(\rho^1_1\rho)$, and
- for all $\rho \in \mathrm{pos}(\delta_2[\hat t_1,\dots,\hat t_{j-1},x_1,\hat t_j,\dots,\hat t_k]\cdot t_{|\rho^2_h})$ we let $r'(\rho^2_h\rho) = r(\rho^2_1\rho)$.

It remains to argue that $r'$ is successful on $t'$. But this is easy to see: as we only cut out and inserted a part of the tree at positions which carry the same state, all transitions are still applicable. Finally, as $\Psi(t') = \Psi(t)$, we obtain $r' \in \mathrm{Run}_{\mathscr{A}}(t')$ and, thus, $\xi' \in \mathscr{L}(\mathscr{A})$.

The proof of item (3.) is analogous. □

## 4 Non-Global Parikh Tree Automata

Now we define a non-global variant of Parikh tree automata in which not an extended Parikh image of a whole input tree is computed, but counter vectors that occur in computations *(i)* are added up pathwise and *(ii)* it is tested at each leaf node whether the resulting counter configuration is contained in the semilinear set $C$ of the automaton.

Here we do not consider counter vectors as additional labelings that we guess beforehand, but use them in the transitions as operations which can differ per path, similar as it is done the case of pushdown tree automata. Therefore, a transition that reads a $k$-ary symbol can send $k$ different vectors to the different subtrees. Additionally, we allow a reset operation $\circlearrowleft$ that sets each counter configuration back to $\mathbf{0}$. This operation will be needed later to define Parikh tree automata that pass the current counter configuration of each node to exactly one child node instead of copying it to all children. Such a reset operation has also been introduced in the context of tree automata with storage to define a linear model [10] and was used for an extension of Parikh string automata (over infinite words) [6].

**Non-global Parikh tree automata.** Let $m \geq 1$. A *(non-global) Parikh tree automaton of dimension $m$ with reset operation* ($m$-PTAR) is a tuple $\mathscr{A} = (Q, \Sigma, q_0, \Delta, C)$ where $Q$ is a finite set of states, $\Sigma$ is a ranked alphabet, $q_0 \in Q$ is the initial state, $C \subseteq \mathbb{N}^m$ is a semilinear set, and $\Delta$ is a finite set of transitions of the form

$$q \to \sigma(q_1(d_1), \ldots, q_n(d_n))$$

where $n \in \mathbb{N}$, $\sigma \in \Sigma^{(n)}$, $q, q_1, \ldots, q_n \in Q$, and $d_1 \ldots d_n \in (\mathbb{N}^m \cup \{\circlearrowleft\})$.

The semantics of an $m$-PTAR $\mathscr{A} = (Q, \Sigma, q_0, \Delta, C)$ is defined as follows. We denote by ID the set $Q \times \mathbb{N}^m$ of *automaton configurations*, each consisting of a state and a *counter configuration* from $\mathbb{N}^m$. For each transition $\tau \in \Delta$ we let $\Rightarrow^\tau$ be the binary relation on the set $T_\Sigma(\mathrm{ID})$ such that for each $\zeta_1, \zeta_2 \in T_\Sigma(\mathrm{ID})$ we have

$$\zeta_1 \Rightarrow^\tau \zeta_2$$

if there are $\hat{\zeta} \in C_\Sigma(\mathrm{ID}, X_1)$, $\hat{\zeta}_1, \hat{\zeta}_2 \in T_\Sigma(\mathrm{ID})$ such that $\zeta_1 = \hat{\zeta} \cdot \hat{\zeta}_1$, $\zeta_2 = \hat{\zeta} \cdot \hat{\zeta}_2$, and either

- $\tau = q \to \sigma(q_1(d_1), \ldots, q_n(d_n))$ for some $n \geq 1$, $\hat{\zeta}_1 = (q, w)$, and $\hat{\zeta}_2 = \sigma((q_1, w_1), \ldots, (q_n, w_n))$ where, for each $i \in [n]$, $w_i = w + d_i$ if $d_i \neq \circlearrowleft$ and $w_i = \mathbf{0}$ otherwise, or
- $\tau = q \to \alpha$ for some $\alpha \in \Sigma^{(0)}$, $\hat{\zeta}_1 = (q, w)$, $w \in C$, and $\hat{\zeta}_2 = \alpha$.

The *computation relation of $\mathscr{A}$* is the binary relation $\Rightarrow_{\mathscr{A}} = \bigcup_{\tau \in \Delta} \Rightarrow^\tau$. A *computation* is a sequence $t = \zeta_0 \Rightarrow^{\tau_1} \zeta_1 \ldots \Rightarrow^{\tau_n} \zeta_n$ (sometimes abbreviated as $\zeta_0 \Rightarrow^{\tau_1 \cdots \tau_n} \zeta_n$) such that $n \in \mathbb{N}$, $\zeta_0, \ldots, \zeta_n \in T_\Sigma(\mathrm{ID})$, $\tau_1, \ldots, \tau_n \in \Delta$, and $\zeta_{i-1} \Rightarrow^{\tau_i} \zeta_i$ for each $i \in [n]$. We say that the *length* of $t$ is $n$. We call $t$ *successful on* $\xi \in T_\Sigma$ if $\zeta_0 = (q_0, \mathbf{0})$ and $\zeta_n = \xi$; the set of all successful computations of $\mathscr{A}$ on $\xi$ is denoted by $\mathrm{comp}_{\mathscr{A}}(\xi)$. The *language recognized by $\mathscr{A}$* is the set $\mathscr{L}(\mathscr{A}) = \{\xi \in T_\Sigma \mid \mathrm{comp}_{\mathscr{A}}(\xi) \neq \emptyset\}$.

Now we consider an example showing the capability of non-global Parikh tree automata to check a semi-linear property for each path in a tree.

**Example 3.** Let $\Sigma = \{a^{(2)}, b^{(2)}, \#^{(0)}\}$. We consider the tree language $L_{ab}$ containing all trees $\xi$ such that the word of labels of each complete path in $\xi$ is of the form $a^n b^n \#$ for some $n \geq 1$, i.e.,

$$L_{ab} = \{\xi \in T_\Sigma \mid \forall \pi \in \mathrm{paths}(\xi) : \xi(\pi) \in \{a^n b^n \# \mid n \geq 1\}\}.$$

This tree language can by recognized by the 2-PTA $\mathscr{A} = (Q, \Sigma, q_a, \Delta, C)$ where $Q = \{q_a, q_b\}$, $C = \{(i,i) \mid i \geq 1\}$, and $\Delta$ contains the transitions

$$\tau_{a,1}\colon\; q_a \to a(q_a(1,0), q_a(1,0)) \qquad \tau_{a,2}\colon\; q_a \to b(q_b(0,1), q_b(0,1))$$

and

$$\tau_{b,1}\colon\; q_b \to b(q_b(0,1), q_b(0,1)) \qquad \tau_{b,2}\colon\; q_b \to \#\,.$$

The intuition behind this automaton is quite easy: for each $a$ it reads, the first counter component is increased by 1 and for each $b$ it reads, the second counter component is increased by 1. Finally, # can only be computed if the value of the first and the second component is equal. This process becomes clear if we look at a part of some computation for $a(b(\#,\#), b(\#,\#)) \in L_{ab}$: let us consider a computation of the form

$$\begin{aligned}(q_a,(0,0)) &\Rightarrow^{\tau_{a,1}} a\big((q_a,(1,0)),(q_a,(1,0))\big)\\ &\Rightarrow^{\tau_{a,2}} a\big(b((q_b,(1,1)),(q_b,(1,1))),(q_a,(1,0))\big)\\ &\Rightarrow^{\tau_{b,2}} a\big(b(\#,(q_b,(1,1))),(q_a,(1,0))\big) \Rightarrow^{*} a(b(\#,\#),b(\#,\#))\,.\end{aligned}$$

Observe that in the third step of the computation, from the automaton configuration $(q_b,(1,1)))$ the application of $\tau_{b,2}$ to compute the leaf # is allowed only because $(1,1) \in C$. ◁

*Remark* 1. We note that the tree language $L_{ab}$ is a *context-free tree language*: it is a simple observation that the set of path words occurring in its trees is context-free and, thus, $L_{ab}$ can be recognized by a pushdown tree automaton. However, we can easily extend Example 3 to paths of the form $a^n b^n c^n\#$ by using a third counter – the resulting tree language would not be context-free anymore.

## 4.1 Restrictions of PTAR

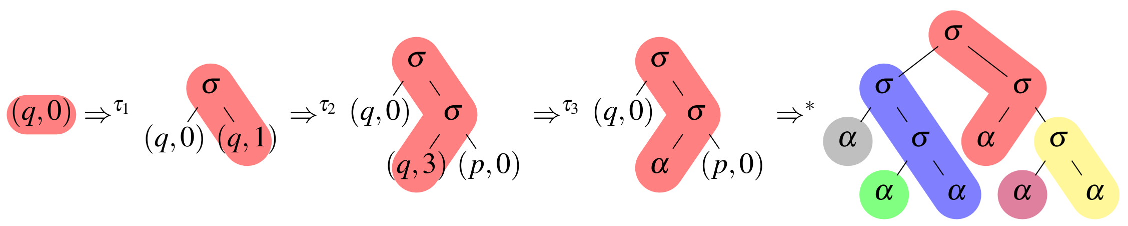
If $\circlearrowleft$ does not occur in the transitions of $\mathscr{A}$, we call $\mathscr{A}$ an $m$-PTA. Moreover, we say that $\mathscr{A}$ is *linear* if for each transition $q \to \sigma(q_1(d_1),\ldots,q_n(d_n))$ in $\Delta$ there is at most one $i \in [n]$ with $d_i \in \mathbb{N}^m$ and for all $j \neq i$ we have $d_j = \circlearrowleft$, i.e., at each node the storage is either completely reset or passed to exactly one child.

**Spinal computation trees** Let us define an alternative semantics for linear PTAR, needed later when we prove decidability of their non-emptiness problem. The idea is to recursively structure the computations of such a linear PTAR $\mathscr{A}$ as follows: during the computation on an input tree $\xi$, we always apply the rewrite relation $\Rightarrow_{\mathscr{A}}$ to the node $w \in \mathrm{pos}(\xi)$ that has been passed the storage from its parent node, if there is any such node. When there is no longer such a node, we apply this process recursively to the remaining nodes labeled by states.

**Example 4.** For an example, consider the linear 1-PTAR $\mathscr{A} = (Q, \Sigma, q, \Delta, C)$, where $Q = \{q, p\}$, $\Sigma = \{\sigma^{(2)}, \alpha^{0}\}$, $C = \mathbb{N}$, and $\Delta$ contains the transitions

$$q \to \sigma\big(q(\circlearrowleft), q(1)\big)\,, \qquad q \to \sigma\big(q(2), p(\circlearrowleft)\big)\,, \qquad q \to \alpha\,, \qquad p \to \sigma(q(\circlearrowleft), q(4)\,,$$

denoted by $\tau_1$, $\tau_2$, $\tau_3$, and $\tau_4$, respectively. Assume the following computation of $\mathscr{A}$.



Since in the computation's first step, the storage has been passed to the second child, labeled $(q,1)$, we rewrite this position in the next step, and so on, until the leaf node $\alpha$ is reached. The path along which this process takes place is shaded in red. Afterwards, the process can be applied recursively to the states which did not receive the storage of their parent, resulting in the paths shaded in other colors.

The gist of this section is that the shaded parts of this computation can also be arranged into a tree of subcomputations of the form

$$\begin{tikzcd}
 & & (q,0) \Rightarrow^* \sigma((q,0),\sigma(\alpha,(p,0))) \arrow[dl, no head] \arrow[dr, no head] & & \\
 & (q,0) \Rightarrow^* \sigma((q,0),\sigma((q,0),\alpha)) \arrow[dl, no head] \arrow[dr, no head] & & (q,0) \Rightarrow^* \sigma((q,0),\alpha) \arrow[d, no head] & , \\
(q,0) \Rightarrow^* \alpha & & (q,0) \Rightarrow^* \alpha & (q,0) \Rightarrow^* \alpha &
\end{tikzcd}$$

called a *spinal computation tree.* We will prove that if the tree language of a PTAR is not empty, then there is a spinal computation tree of bounded height, leading to a decision procedure. $\triangleleft$

To formally define the notion of spinal computation trees, we have to restrict the derivation relation so that only children that received the storage from their parents can be rewritten. We do so by constructing a new automaton which only has transitions for such positions. For this, assume a linear $m$-PTAR $\mathscr{A} = (Q,\Sigma,q_0,\Delta,C)$. Let $Q' = Q \cup \hat{Q}$, where $\hat{Q} = \{\hat{q} \mid q \in Q\}$. We construct the linear $m$-PTAR $\mathscr{A}' = (Q',\Sigma,\hat{q}_0,\Delta',C)$ , where $\Delta'$ is defined as follows.

- For every transition of the form $q \to \alpha$ in $\Delta$, the set $\Delta'$ contains the transition $\hat{q} \to \alpha$.
- For every transition $q \to \sigma(q_1(d_1),\dots,q_n(d_n))$ in $\Delta$, where $d_i = \circlearrowleft$ for each $i \in [n]$, the set $\Delta'$ contains the transition
$$\hat{q} \to \sigma(q_1(d_1),\dots,q_n(d_n))\,.$$
- Finally, consider a transition $q \to \sigma(q_1(d_1),\dots,q_n(d_n))$ in $\Delta$ such that $d_i \neq \circlearrowleft$ for some $i \in [n]$. Then the transition
$$\hat{q} \to \sigma\big(q_1(d_1),\dots,q_{i-1}(d_{i-1}),\hat{q}_i(d_i),q_{i+1}(d_{i+1}),\dots,q_n(d_n)\big)$$
is in $\Delta'$.

Note that there are only transitions for states from $\hat{Q}$ in $\Delta'$, the computation cannot continue on states from $Q$.

Consider a computation
$$\zeta_0 \Rightarrow^{\tau_1} \zeta_1 \Rightarrow^{\tau_2} \cdots \Rightarrow^{\tau_n} \zeta_n$$
of $\mathscr{A}'$, such that $n > 0$, $\zeta_0 = (\hat{q},\mathbf{0})$ for some $q \in Q$, and $\zeta_n \in T_\Sigma(Q \times \mathbb{N}^m)$. We call such a computation a *spine computation* of $\zeta_n$ from $q$. In fact, it is easy to see from the definition of $\mathscr{A}'$ that for every occurrence in $\zeta_n$ of a tuple $(q,c)$ with $q \in Q$ and $c \in \mathbb{N}^m$, it is the case that $c = \mathbf{0}$.

The set of all spine computations from $q$ will be denoted by $S_q$, and given such a spine computation $s \in S_q$, the generated tree $\zeta_n$ will be denoted by $\mathrm{tree}(s)$. Moreover, assume that $\{w_1,\cdots,w_\ell\} \subseteq \mathrm{pos}(\mathrm{tree}(s))$, for some $\ell \in \mathbb{N}$, is the set of positions in $\mathrm{tree}(s)$ with labels from $Q \times \mathbb{N}^m$, and assume that $w_1, \dots, w_\ell$ are in left-to-right order, i.e. $w_1 <_{\mathrm{lex}} \cdots <_{\mathrm{lex}} w_\ell$. Then we will write $\mathrm{statepos}(s)$ for the sequence $w_1 \cdots w_\ell$. Additionally, if for every $i \in [\ell]$, we have $\mathrm{tree}(s)(w_i) = (q_i,\mathbf{0})$, then we will denote the sequence $q_1 \cdots q_\ell$ by $\mathrm{stateseq}(s)$. For instance, when we write $s$ for the red-shaded subcomputation from Example 4, we would have

$$\mathrm{tree}(s) = \sigma((q,0), \sigma(\alpha,(p,0))), \qquad \mathrm{statepos}(s) = 1\ \ 22, \qquad \text{and} \qquad \mathrm{stateseq}(s) = q\,p.$$

Now, for every $q \in Q$, the set of *spinal computation trees $D_q$* is defined to be the smallest set such that the following property holds: for every spine computation $s \in S_q$ with $\mathrm{stateseq}(s) = q_1 \cdots q_\ell$ where $\ell \in \mathbb{N}$, and for every $d_i \in D_{q_i}$, where $i \in [\ell]$, the tree $s(d_1,\ldots,d_\ell)$ is an element of $D_q$.[2]

Given a spinal computation tree $d = s(d_1,\ldots,d_\ell) \in D_q$ with $\mathrm{statepos}(s) = w_1 \cdots w_\ell$, we finally define the computed tree $\mathrm{tree}(d) \in T_\Sigma$ by

$$\mathrm{tree}(d) = \mathrm{tree}(s)[\mathrm{tree}(d_1)]_{w_1} \cdots [\mathrm{tree}(d_\ell)]_{w_\ell}\,.$$

This recursive definition is well-behaved because we chose $D_q$ to be the smallest set of trees fulfilling the given property.

The following lemma relates the rewrite semantics of PTAR to the notion of spinal computation trees.

**Lemma 4.** *Let $\mathscr{A} = (Q,\Sigma,q_0,\Delta,C)$ be a linear PTAR, let $q \in Q$, and let $\xi \in T_\Sigma$. Then $(q,\mathbf{0}) \Rightarrow^*_{\mathscr{A}} \xi$ if and only if there is some $d \in D_q$ with* $\mathrm{tree}(d) = \xi$. *In particular, $\xi \in \mathscr{L}(\mathscr{A})$ if and only if there is some $d \in D_{q_0}$ with* $\mathrm{tree}(d) = \xi$.

The direction "if" of the lemma can be shown by recursively "composing" the spine computations in $d$. For the direction "only if", one has to reorder the computation of $\xi$ such that it begins with the computation steps along the spine where no reset operation is performed. Then these steps correspond to a spine computation $s$. As all computations besides the spine start in a configuration $(q,\mathbf{0})$ for some state $q \in Q$, this process can be repeated recursively to obtain a spinal computation tree $d$.

**Lemma 5.** *For every linear PTAR $\mathscr{A}$ with state set Q, if $\mathscr{L}(\mathscr{A}) \neq \emptyset$, then there is some spinal computation tree $d \in D_{q_0}$ such that* $\mathrm{ht}(d) \leq |Q|$.

*Proof.* By Lemma 4, we know that $\mathscr{L}(\mathscr{A}) \neq \emptyset$ implies the existence of some $d \in D_{q_0}$. Assume that there is some path $\rho_1 \ldots \rho_n$ in $d$ such that $n > |Q|$. But then there are two distinct indices $i$ and $j \in [n]$, say $i < j$, such that $d(\rho_i) \in S_q$ and $d(\rho_j) \in S_q$ for some $q \in Q$. Construct

$$d' = d\big[d_{|\rho_j}\big]_{\rho_i}\,.$$

It is easy to see that $d'$ is also a valid spinal computation tree in $D_{q_0}$, by inspection of the property used in their definition. Moreover, $\mathrm{size}(d') < \mathrm{size}(d)$, so by iterating this construction a finite number of times, we obtain a tree $d'' \in D_{q_0}$ such that $\mathrm{ht}(d'') \leq |Q|$. □

Now we turn to an example for a tree language that is recognizable by a linear PTAR and still quite powerful: Although each counter configuration is passed to exactly one subtree, this PTAR ensures that an arithmetical constraint holds on each c-path in the trees it accepts.

**Example 5.** Let $\Sigma = \{a^{(2)}, b^{(2)}, c^{(1)}, d^{(1)}, \#^{(0)}\}$. Given a word $w \in \Sigma^*$, we denote by $\mathrm{pref}(w)$ the set of all nonempty prefixes of $w$, i.e., $\mathrm{pref}(w) = \{u \in \Sigma^+ \mid u \sqsubseteq w\}$. Now consider the tree language $L_{\mathrm{lin}}$ consisting of trees $\xi$ of the form

$$a(u_1, a(u_2, \ldots a(u_n, b(u_{n+1}, \ldots b(u_{2n}, \#)))))$$

[2]Note that this is the point mentioned in the preliminaries, because of which we must allow trees with labels from an infinite ranked set: the set of labels used for $D_q$ is the set of spine computations $\bigcup_{q \in Q} S_q$.

for some $n \geq 1$ and with $u_i \in \{c^m d^m \# \mid m \geq 1\}$ for each $i \in [2n]$. Thus, for each $\xi \in L_{\text{lin}}$ there is exactly one c-path $\pi \in \text{paths}(\xi)$ with $\xi(\pi) = a^n b^n \#$ for some $n \geq 1$ and for each remaining c-path $\pi' \in \text{paths}(\xi)$ we have $\xi(\pi') = wu$ with $w \in \text{pref}(a^n b^n)$ and $u \in \{c^m d^m \# \mid m \geq 1\}$.

The tree language $L_{\text{lin}}$ can be recognized by the following linear 2-PTAR: $\mathscr{A} = (Q, \Sigma, q_0, \Delta, C)$ where $Q = \{q_a, q_b, q_c, q_d\}$, $C = \{(i,i) \mid i \geq 1\}$, and $\Delta$ contains the following transitions:

- $q_a \to a(q_c(\circlearrowleft), q_a(1,0))$, $q_a \to b(q_c(\circlearrowleft), q_b(0,1))$
- $q_b \to b(q_c(\circlearrowleft), q_b(0,1))$, $q_b \to \#$, and
- $q_c \to c(q_c(1,0))$, $q_c \to d(q_d(0,1))$,
- $q_d \to d(q_d(0,1))$, $q_d \to \#$

Thus, in the states $q_a$ and $q_b$, $\mathscr{A}$ counts number of $a$s and $b$s, respectively. When switching into state $q_c$, the counter configuration is reset and from now on it counts the number of $c$s and $d$s. $\lhd$

## 5 Expressiveness

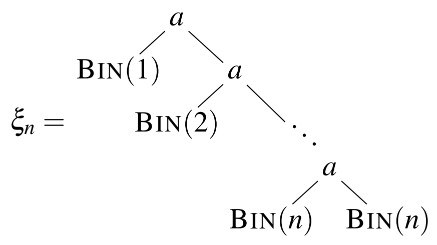
In this section we want to examine how the different formalisms we have introduced relate to each other in terms of their expressiveness.

### 5.1 GPTA and PTA

First, we want to compare the classes of tree languages recognizable by PTA and GPTA. As the different counting mechanisms of the two models already intuitively suggest, the two classes are incomparable. For the formal proof we use tree languages which require counting on paths, or global counting, respectively. We start by showing that the tree language $L_{ab}$ from Example 3 can not be recognized by a global Parikh tree automaton by using the exchange lemma we obtained for GPTA (Lemma 3).

**Lemma 6.** *The tree language $L_{ab}$ is not GPTA-recognizable.*

*Proof.* Assume that there is some GPTA $\mathscr{A}$ with $\mathscr{L}(\mathscr{A}) = L_{ab}$ and let $p, l \in \mathbb{N}$ as in the proof of Lemma 3. Now consider, for $n \geq 1$, the tree



where BIN($n$) is the complete binary tree over $b$ of height $n$ inductively defined by $\text{BIN}(1) = b(\#, \#)$ and $\text{BIN}(i) = b(\text{BIN}(i-1), \text{BIN}(i-1))$ for each $i > 1$. Clearly, $\xi_n \in L_{ab}$ for each $n \in \mathbb{N}$.

Now choose $n$ big enough such that there are at least $l$ independent subtrees of height at least $p$ in $\xi_n$ and, thus, the requirements of Lemma 3 are fulfilled. However, it is not hard to see that we will not find a context $\zeta_2$ as in Lemma 3 in $\xi_n$ such that item (2.) and (3.) of the lemma are satisfied: as $\zeta_2$ needs to occur in two independent subtrees, it can only consist of $b$s. However, cutting out $b$s in one subtree and inserting them in another one necessarily leads to c-paths that are not of the form $a^k b^k \#$ anymore and, thus, the resulting tree $\xi'$ is not in $L_{ab}$. This is a contradiction. $\square$

We can use the exchange lemma in a very similar way to show that $L_{\text{lin}}$, which is recognizable by a linear PTAR, is not GPTA-recognizable either.

**Corollary 1.** *The tree language $L_{\text{lin}}$ is not GPTA-recognizable.*

For the other direction, we consider a tree language where the number of symbol occurrences on two different paths are compared. This global counting behavior cannot be implemented by non-global PTA.

**Example 6.** Let $\Sigma = \{\sigma^{(2)}, \gamma^{(1)}, \#^{(0)}\}$ and consider the tree language

$$L_{\gamma\gamma} = \{\sigma(\gamma^n\#, \gamma^n\#) \mid n \in \mathbb{N}\}$$

which can be recognized by the following 2-GPTA $\mathscr{A}$: We let $\mathscr{A} = (Q, \Sigma, D, q_0, \Delta, C)$ where $Q = \{q_0, q_1, q_2\}$, $D = \{(0,0), (1,0), (0,1)\}$, $C = \{(i,i) \mid i \in \mathbb{N}\}$, and $\Delta$ consists of the transitions

- $q_0 \to \langle \sigma, (0,0) \rangle (q_1, q_2)$,
- $q_1 \to \langle \gamma, (1,0) \rangle (q_1)$ and $q_2 \to \langle \gamma, (0,1) \rangle (q_2)$, as well as
- $q_1 \to \langle \#, (0,0) \rangle$ and $q_2 \to \langle \#, (0,0) \rangle$. $\triangleleft$

**Lemma 7.** *The language $L_{\gamma\gamma}$ is not PTA-recognizable.*

*Proof.* Assume towards a contradiction there is some $m \in \mathbb{N}$ and an $m$-PTA $\mathscr{A}$ with $L(\mathscr{A}) = L_{\gamma\gamma}$. As all trees in $L_{\gamma\gamma}$ are of the shape $\sigma(\gamma^n\#, \gamma^n\#)$ for some $n \in \mathbb{N}$, each computation of $\mathscr{A}$ on some $\xi = \sigma(\xi_1, \xi_1) \in L_{\gamma\gamma}$ has to be of the form

$$(q_0, \mathbf{0}) \Rightarrow \sigma((q_1, s_1), (q_2, s_2)) \Rightarrow^*_{\mathscr{A}} \xi$$

for some $(q_1, s_1), (q_2, s_2) \in \text{ID}$. As $\Delta$ is finite, there are only finitely many configurations $(q, s) \in \text{ID}$ reachable from $(q_0, \mathbf{0})$ in one step by reading a $\sigma$ and occurring in a successful computation; we denote the set of all those configurations by $\text{ID}_1$. However, as $L_{\gamma\gamma}$ is infinite, there has to be a $(q, s) \in \text{ID}_1$ with $(q, s) \Rightarrow^*_{\mathscr{A}} \gamma^{n_1}\#$, $(q, s) \Rightarrow^*_{\mathscr{A}} \gamma^{n_2}\#$, and $n_1 \neq n_2$. Suppose without loss of generality that $(q, s)$ is reached in the left subtree of $\sigma$, i.e. $(q_0, \mathbf{0}) \Rightarrow_{\mathscr{A}} \sigma((q, s), (q', s'))$ for some $(q', s') \in \text{ID}$.

By the assumption on $(q, s)$, there exists a computation

$$(q_0, \mathbf{0}) \Rightarrow \sigma((q, s), (q', s')) \Rightarrow^*_{\mathscr{A}} \sigma(\xi_1, \xi_1)$$

and we can assume that $\xi_1 = \gamma^{n_1}\#$. But by the above also

$$(q_0, \mathbf{0}) \Rightarrow \sigma((q, s), (q', s')) \Rightarrow^*_{\mathscr{A}} \sigma(\gamma^{n_2}\#, \xi_1)$$

and $\sigma(\gamma^{n_2}\#, \xi_1) \notin L_{\gamma\gamma}$, which is a contradiction. $\square$

From Lemma 6 and 7 it immediately follows that the tree languages recognizable by GPTA and PTA are incomparable.

**Theorem 1.** *The classes of tree languages recognizable by GPTA and PTA are incomparable.*

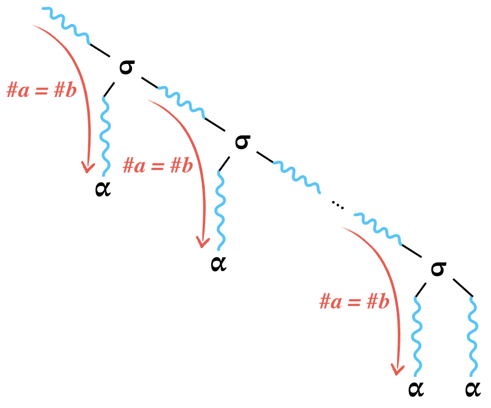
## 5.2 PTA, PTAR, and Linear PTAR

In contrast to the string case, in the tree case a reset cannot be simulated simply by guessing the last reset position: because of branching, a counter configuration could be processed further in one subtree, while a reset takes place in the second subtree. This observation is illustrated by the following example.

**Example 7.** We consider the ranked alphabet $\Sigma = \{\sigma^{(2)}, a^{(1)}, b^{(1)}, \alpha^{(0)}\}$ as well as the 2-PTAR $\mathscr{A} = (\{q_0, q_1\}, \Sigma, q_0, \Delta, C)$ where $C = \{(i,i) \mid i \in \mathbb{N}\}$ and $\Delta$ contains the following transitions:

- $q_j \to a(q_j(1,0))$ and $q_j \to b(q_j(0,1))$ for $j \in \{0,1\}$,
- $q_0 \to \sigma(q_1(0,0), q_0(\circlearrowleft))$, $q_0 \to \sigma(q_1(0,0), q_1(0,0))$, and
- $q_1 \to \alpha$

It is easy to observe that for each tree $\xi \in \mathscr{L}(\mathscr{A})$ it holds that if the context $\sigma(x_1, w_1(\sigma(w_2\alpha, x_2)))$ occurs in $\xi$ for some $w_1, w_2 \in \{a,b\}^*$, then the number of $a$s in $w_1w_2$ equals the number of $b$s in $w_1w_2$. Moreover, each $\sigma$ only occurs on the rightmost c-path in $\xi$. Thus, each $\xi \in \mathscr{L}(\mathscr{A})$ is of the form



where the red arrows indicate the paths in $\xi$ on which the number constraint on $a$s and $b$s is tested, respectively. As there might be arbitrary many such tests that are not calculated in completely independent subtrees, it is crucial to reset the counter configuration in between. ◁

Therefore, we strongly expect PTAR to be more expressive than PTA. However, our proof methods for PTA were not sufficient to formally prove this statement.

*Conjecture.* PTA are strictly less expressive than PTAR.

Finally, we observe that also the property of a PTAR to be linear restricts its expressive power.

**Lemma 8.** *Linear PTAR are strictly less expressive than PTAR.*

*Proof.* Let $\Sigma = \{\sigma^{(2)}, \gamma^{(1)}, \#^{(0)}\}$ and consider the tree language $L_3 = \{\gamma^n(\sigma(\gamma^n\#, \gamma^n\#)) \mid n \in \mathbb{N}\}$. This tree language can be recognized by a 2-PTA $\mathscr{A} = (\{q_0, q_1\}, \Sigma, q_0, \Delta, C)$ where $C = \{(i,i) \mid i \in \mathbb{N}\}$ and $\Delta$ consists of the transitions $q_0 \to \gamma(q_0(1,0))$, $q_0 \to \sigma(q_1(0,0), q_1(0,0))$, $q_1 \to \gamma(q_1(0,1))$, and $q_1 \to \#$.

On the other hand, it is not hard to see that $L_3$ cannot be recognized by any linear PTAR $\mathscr{A}'$: by definition, each transition recognizing $\sigma$ in $\mathscr{A}'$ has to be of the form $q \to \sigma(q_1(d), q_2(\circlearrowleft))$ or $q \to \sigma(q_1(\circlearrowleft), q_2(d))$. Then the argumentation is very similar to the proof of Lemma 7: we will find a state $p$ such that (1) $(p, \mathbf{0})$ occurs in a successful computation of $\mathscr{A}'$ and (2) there are $n_1, n_2 \in \mathbb{N}$ with $n_1 \neq n_2$, $(p, \mathbf{0}) \Rightarrow^*_{\mathscr{A}'} \gamma^{n_1}\#$, and $(p, \mathbf{0}) \Rightarrow^*_{\mathscr{A}'} \gamma^{n_2}\#$. Thus, $\mathscr{A}'$ cannot recognize $L_3$. □

## 6 Decidability

Now we investigate the question of decidability for two basic problems of PTAR – the non-emptiness problem and the membership problem.[3] The former is undecidable in general: as soon as we consider PTA of at least dimension 3, we can simulate calculations of two-counter machines [17, 12] in a similar way as it was done in [16, Lemma 3.4] for *and-branching two-counter machines without zero-test* (ACM).

A *two-counter machine (2CM)* is a tuple $M = (Q, q_0, Q_f, T)$ where $Q$ is a finite set of states, $q_0 \in Q$ is the initial state, $Q_f \subseteq Q$ is a set of final states and $T$ is a finite set of transitions of the following two forms:

$$(q, f, q') \qquad \text{(instruction)}$$

$$(q, p, q') \qquad \text{(zero-test)}$$

where $q, q' \in Q$, $p \in \{0(1), 0(2)\}$, and $f \in \{\text{inc}(1), \text{inc}(2), \text{dec}(1), \text{dec}(2)\}$.

Define for convenience of notation $\neg 1 = 2$ and $\neg 2 = 1$. For each $\tau \in T$ we let $\Rightarrow^{\tau}$ be the binary relation on $Q \times \mathbb{N} \times \mathbb{N}$ such that for each $(q, k_1, k_2), (q', k'_1, k'_2) \in (Q \times \mathbb{N} \times \mathbb{N})$ we have $(q, k_1, k_2) \Rightarrow^{\tau} (q', k'_1, k'_2)$ if either

- $\tau = (q, \text{inc}(i), q')$ for some $i \in \{1,2\}$, $k'_i = k_i + 1$, and $k'_{\neg i} = k_{\neg i}$, or
- $\tau = (q, \text{dec}(i), q')$ for some $i \in \{1,2\}$, $k_i > 0$, $k'_i = k_i - 1$, and $k'_{\neg i} = k_{\neg i}$, or
- $\tau = (q, 0(i), q')$, $k_i = 0$, $k'_1 = k_1$, and $k_2 = k'_2$.

We let $\Rightarrow_M = \bigcup_{\tau \in T} \Rightarrow^{\tau}$. We say that $M$ *accepts* if $(q_0, 0, 0) \Rightarrow_M^* (q_f, 0, 0)$ for some $q_f \in Q_f$. By the classical result that, given a 2CM $M$ and $k_1, k_2 \in \mathbb{N}$, it is undecidable whether $(q_0, k_1, k_2) \Rightarrow_M^* (q_f, 0, 0)$ [17], it is straightforward to obtain undecidability of acceptance of 2CM.

**Lemma 9** ([17]). *Let M be a 2CM. It is undecidable whether $(q_0, 0, 0) \Rightarrow_M^* (q_f, 0, 0)$ for some $q_f \in Q_f$.*

**Theorem 2.** *For each $m \geq 3$ and m-PTA $\mathscr{A}$ it is undecidable whether $\mathscr{L}(\mathscr{A}) \neq \emptyset$.*

*Proof.* To prove the statement we reduce the acceptance problem of 2CM to the emptiness problem of PTA similar to the proof of [16, Lemma 3.4]. The idea is to simulate zero-tests with branching: while in the right successor the calculation continues as if the zero-test had been successful, in the left subtree it is checked whether the zero-test is indeed successful. In contrast to 2CM and ACM, PTAs cannot decrement their counters. Therefore, we need 3 counters to represent the counter values of 2CM – the counter configuration $(s_1, s_2, l)$ of a PTA stands for the value $(s_1 - l, s_2 - l)$ of the 2CM; each $(j, j, j)$ represents $(0,0)$. In addition, for each decrement of counter $i$ it must be tested that $l$ is smaller than $s_i$, this also happens via branching.

Given a 2CM $M = (Q, q_0, Q_f, T)$, we construct the 3-PTA $\mathscr{A}$ as follows: Let $\Sigma = \{\sigma^{(2)}, \gamma^{(1)}, \alpha^{(0)}\}$ be a ranked alphabet and $\mathscr{A} = (Q', \Sigma, q_0, \Delta, C)$ where $Q' = Q \cup \{=_1, =_2, <_1, <_2\}$, $C = \{(i,i,i) \mid i \in \mathbb{N}\}$, and $\Delta$ consists of the following transitions:

- for each transition of the form $(q, \text{inc}(i), q')$ in $T$, the transition $q \to \gamma(q'(d))$ is in $\Delta$ where $d = (2,1,1)$ if $i = 1$ and $d = (1,2,1)$ if $i = 2$,
- for each transition of the form $(q, \text{dec}(i), q')$ in $T$, the transition

$$q \to \sigma(<_i(d), q'(d))$$

is in $\Delta$ where $d = (0,1,1)$ if $i = 1$ and $d = (1,0,1)$ if $i = 2$,

[3] Note that the universality problem is undecidable already for Parikh string automata [15, Prop. 7].

- for each transition of the form $(q, 0(i), q')$ in $T$, the transition
$$q \to \sigma(=_i (0,0,0), q'(0,0,0))$$
is in $\Delta$,
- for each $q_f \in Q_f$ the transition $q_f \to \alpha$ is in $\Delta$,
- for each $d \in \{(0,1,0),(0,0,1),(1,0,1)\}$ the transition $<_1 \to \gamma(<_1 (d))$ is in $\Delta$ and for each $d' \in \{(1,0,0),(0,0,1),(0,1,1)\}$ the transition $<_2 \to \gamma(<_2 (d'))$ is in $\Delta$,
- for each $d \in \{(1,0,1),(0,1,0)\}$ the transition $=_1 \to \gamma(=_1 (d))$ is in $\Delta$ and for each $d' \in \{(0,1,1),(1,0,0)\}$ the transition $=_2 \to \gamma(=_2 (d'))$ is in $\Delta$, and
- the transitions $<_i \to \alpha$ and $=_i \to \alpha$ are in $\Delta$ for each $i \in \{1,2\}$.

We can show that $\mathscr{L}(\mathscr{A}) \neq \emptyset$ if and only if $(q_0,0,0) \Rightarrow_M^* (q_f,0,0)$ for some $q_f \in Q_f$ by induction on the length of the respective computations. For this, we note that the mapping $\varphi\colon T \to \Delta$ given by the above construction is an injection. Moreover, the following three observations are helpful:

**Observation 1.** *Let $s_1, s_2, l \in \mathbb{N}$. Then $(<_i, (s_1,s_2,l)) \Rightarrow_{\mathscr{A}}^* \gamma^n(<_i, (j,j,j)) \Rightarrow_{\mathscr{A}} \gamma^n(\alpha)$ for some $j \in \mathbb{N}$ if and only of either $i = 1$ and $l \le s_1$ or $i = 2$ and $l \le s_2$.*

**Observation 2.** *Let $s_1, s_2, l \in \mathbb{N}$. Then $(=_i, (s_1,s_2,l)) \Rightarrow_{\mathscr{A}}^* \gamma^n(=_i, (j,j,j)) \Rightarrow_{\mathscr{A}} \gamma^n(\alpha)$ for some $j \in \mathbb{N}$ if and only of either $i = 1$ and $s_1 = l$ or $i = 2$ and $s_2 = l$.*

**Observation 3.** *Let $q_1, q_2 \in Q$, let $s_1, s_2, l, s_1', s_2', l' \in \mathbb{N}$, and let $\zeta \in C_\Sigma(X_1)$. If $(q_1, (s_1,s_2,l)) \Rightarrow_{\mathscr{A}}^* \zeta[(q_2, (s_1', s_2', l'))]$, then also $(q_1, (s_1+1, s_2+1, l+1)) \Rightarrow_{\mathscr{A}}^* \zeta[(q_2, (s_1'+1, s_2'+1, l'+1))]$.*

By using Lemma 9, we can conclude that non-emptiness of $m$-PTA for $m \ge 3$ is undecidable. □

Now we come to a case of PTAR for which the situation is different: we can show that for linear PTAR non-emptiness is decidable. To do so, we use the fact that for every non-empty linear PTAR there must be a tree with less than $|Q|+1$ non-reset paths (Lemma 5) and, thus, reduce the problem to non-emptiness of Parikh string automata, which is decidable (Lemma 2).

**Definition 1.** Let $\mathscr{A} = (Q, \Sigma, q_0, \Delta, C)$ be a linear PTAR, $U \subseteq Q$, and $q \in Q$. The $(U,q)$-*linearization automaton of* $\mathscr{A}$ is the PA $\mathscr{A}' = (Q, \Sigma, q, \Delta', F, C)$ where
$$F = \{p \mid p \to \alpha \in \Delta, \alpha \in \Sigma^{(0)}\} \cup \{p \mid p \to \sigma(q_1(d_1), \ldots, q_n(d_n)) \in \Delta, d_1, \ldots, d_n = \circlearrowleft, q_1, \ldots, q_n \in U\}$$
and $\Delta'$ contains the transition $(p, \sigma, d, p')$ if and only if there is a transition $p \to \sigma(q_1(d_1), \ldots, q_n(d_n)) \in \Delta$, $i \in [n]$, such that $q_i = p'$ and $d_i = d \neq \circlearrowleft$, and $q_1, \ldots, q_{i-1}, q_{i+1}, \ldots, q_n \in U$.

**Theorem 3.** *Given a linear PTAR $\mathscr{A}$, it is decidable whether $\mathscr{L}(\mathscr{A}) \neq \emptyset$.*

*Proof.* Consider Algorithm 1. We claim that this algorithm is a decision procedure for the non-emptiness problem of linear PTAR.

Observe that, for every $U \subseteq Q$ and every $q \in Q$, when $\mathscr{A}'$ is the $(U,q)$-linearization automaton of $\mathscr{A}$, we have
$$\mathscr{L}(\mathscr{A}') \neq \emptyset \qquad \text{iff} \qquad \exists s \in S_q\colon \text{stateseq}(s) \in U^*. \tag{$\star$}$$

By this property, we can conclude that the following loop invariant holds for the outer loop of the algorithm: for every $j \in \mathbb{N}$ and $q \in Q$, we have
$$q \in U_j \qquad \text{iff} \qquad \exists d \in D_q\colon \text{ht}(d) \le j.$$

**Algorithm 1** Decision procedure for non-emptiness of linear PTAR

```
Input: linear PTAR $\mathscr{A} = (Q, \Sigma, q_0, \Delta, C)$
Output: "$\mathscr{L}(\mathscr{A}) \neq \emptyset$" if $\mathscr{L}(\mathscr{A}) \neq \emptyset$, otherwise "$\mathscr{L}(\mathscr{A}) = \emptyset$".
  $U_0 \leftarrow \emptyset$, $i \leftarrow 0$
  repeat
      $U_{i+1} \leftarrow U_i$
      for $q \in Q$ do
          Construct the $(U_i, q)$-linearization automaton $\mathscr{A}'$ of $\mathscr{A}$.
          if $\mathscr{L}(\mathscr{A}') \neq \emptyset$ then
              $U_{i+1} \leftarrow U_{i+1} \cup \{q\}$
          end if
      end for
      $i \leftarrow i + 1$
  until $U_i = U_{i-1}$
  if $q_0 \in U_i$ then Output "$\mathscr{L}(\mathscr{A}) \neq \emptyset$" else Output "$\mathscr{L}(\mathscr{A}) = \emptyset$" end if
```

The proof of the loop invariant is by induction on $j$. The base case $j = 0$ is vacuously true, so assume the property is proven for some $j \in \mathbb{N}$. For the direction "only if", assume some $q \in U_{j+1}$. The case $q \in U_j$ is already covered by the induction hypothesis, so it remains to consider $q \in U_{j+1} \setminus U_j$. Then the language of the $(U_j, q)$-linearization automaton of $\mathscr{A}$ is nonempty, hence there is some $s \in S_q$ with $\mathrm{stateseq}(s) \in U_j^*$ by $(\star)$. Moreover, by the induction hypothesis, there are, for all states $q_i$ in $\mathrm{stateseq}(s)$, computation trees $d_i \in D_{q_i}$ with $\mathrm{ht}(d_i) \leq j$. So the computation tree $s(d_1, \ldots, d_n) \in D_q$ has height at most $j + 1$.

For the direction "if", assume that there is some $d \in D_q$ such that $\mathrm{ht}(d) \leq j + 1$. Again, if $\mathrm{ht}(d) < j + 1$, we can apply the induction hypothesis and are done immediately. So consider the case $\mathrm{ht}(d) = j + 1$. Then $d = s(d_1, \ldots, s_\ell)$ for some $s \in S_q$, $\ell \in \mathbb{N}$ and $d_i \in D_{q_i}$ for all $i \in [\ell]$. In particular $\mathrm{ht}(d_i) \leq j$, so $q_i \in U_j$ by the induction hypothesis. Moreover, by $(\star)$, the language of the $(U_j, q)$-linearization automaton is nonempty, so we obtain that $q \in U_{j+1}$.

To show correctness of the algorithm, assume that the algorithm outputs "$\mathscr{L}(\mathscr{A}) \neq \emptyset$". Then we have $q_0 \in U_k$, where $k$ is the value of the counter $i$ after the outer loop terminates. By the loop invariant, there is some $d \in D_{q_0}$, and by Lemma 4, $\mathscr{L}(\mathscr{A}) \neq \emptyset$.

For the proof of completeness of the algorithm, assume that $\mathscr{L}(\mathscr{A}) \neq \emptyset$. By Lemma 5, there is some $d \in D_{q_0}$ with $\mathrm{ht}(d) \leq |Q|$, and by the loop invariant, this means that $q_0 \in U_j$ for some $j \leq |Q|$. Thus, also $q_0 \in U_k$, where $k$ is the value of the counter $i$ after the outer loop terminates. So the algorithm outputs "$\mathscr{L}(\mathscr{A}) \neq \emptyset$". □

Finally, we obtain that membership is decidable for arbitrary PTAR.

**Theorem 4.** *Given an m-PTAR $\mathscr{A}$ over $\Sigma$ and a tree $\xi \in T_\Sigma$, it is decidable whether $\xi \in \mathscr{L}(\mathscr{A})$.*

*Proof.* In order to check whether $\xi \in \mathscr{L}(\mathscr{A})$, the naive approach is sufficient: Clearly, $\xi \in \mathscr{L}(\mathscr{A})$ if and only if $\mathrm{comp}_{\mathscr{A}}(\xi) \neq \emptyset$. As each $t \in \mathrm{comp}_{\mathscr{A}}(\xi)$ is of length $|\xi|$, we can simply guess a sequence of transitions $\tau_1 \ldots \tau_{|\xi|}$ and check whether its application in lexicographic order results in a valid computation. This mainly involves to ensure a valid state behavior and to test for each leaf whether the reached counter configuration is an element of $C$. The latter is decidable due to Lemma 1. □

## 7 Conclusion

In this work, we introduced non-global PTA and compared its expressive power with that of GPTA. To do so, we generalized an exchange lemma known from Parikh word automata to GPTA. Furthermore, we investigated the question of decidability of non-emptiness and membership for PTA and linear PTAR.

**Future work** Our investigations in this paper were only a first step and raise many more questions that can be addressed in future work. In particular, we think it worthwhile to further investigate the following questions:

- Is it possible to formulate an exchange lemma for (linear) PTA(R)? Since the successful computations of all subtrees of a node depend on the current counter configuration, our attempts to reorder parts of a computation have not been successful so far.
- Are PTAR strictly more expressive than PTA?

Finally, we did not investigate closure properties for the different models introduced in this work as well as complexities of their non-emptiness and membership problems. We think that a further study could contribute to a more complete picture.

**Acknowledgements** We want to thank the reviewers for their insightful and detailed comments, which helped us to improve the paper. In particular, one of the reviewers had a really nice idea how to strengthen the undecidability result from 4-PTA to 3-PTA.

## References

[1] Michaël Cadilhac, Alain Finkel & Pierre McKenzie (2011): *On the Expressiveness of Parikh Automata and Related Models*. In Rudolf Freund, Markus Holzer, Carlo Mereghetti, Friedrich Otto & Beatrice Palano, editors: *Third Workshop on Non-Classical Models for Automata and Applications (NCMA 2011)*, *books@ocg.at* 282, Austrian Computer Society, pp. 103–119.

[2] Michaël Cadilhac, Arka Ghosh, Guillermo A. Pérez & Ritam Raha (2023): *Parikh One-Counter Automata*. In Jérôme Leroux, Sylvain Lombardy & David Peleg, editors: *48th International Symposium on Mathematical Foundations of Computer Science (MFCS 2023)*, *LIPIcs* 272, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, pp. 30:1–30:15, doi:10.4230/LIPIcs.MFCS.2023.30.

[3] Jean-Baptiste Courtois & Sylvain Schmitz (2014): *Alternating Vector Addition Systems with States*. In Erzsébet Csuhaj-Varjú, Martin Dietzfelbinger & Zoltán Ésik, editors: *Mathematical Foundations of Computer Science 2014*, *LNCS* 8634, Springer, pp. 220–231, doi:10.1007/978-3-662-44522-8_19.

[4] Akio Fujiyoshi & Takumi Kasai (2000): *Spinal-Formed Context-Free Tree Grammars*. *Theory Comput. Syst.* 33(1), pp. 59–83, doi:10.1007/S002249910004.

[5] Seymour Ginsburg & Edwin Spanier (1966): *Semigroups, Presburger formulas, and languages*. *Pacific Journal of Mathematics* 16(2), pp. 285–296, doi:10.2140/pjm.1966.16.285.

[6] Mario Grobler, Leif Sabellek & Sebastian Siebertz (2024): *Remarks on Parikh-Recognizable Omega-languages*. In Aniello Murano & Alexandra Silva, editors: *32nd EACSL Annual Conference on Computer Science Logic (CSL 2024)*, *LIPIcs* 288, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, pp. 31:1–31:21, doi:10.4230/LIPIcs.CSL.2024.31.

[7] I. Guessarian (1981): *On pushdown tree automata*. In G. Goos, J. Hartmanis, W. Brauer, P. Brinch Hansen, D. Gries, C. Moler, G. Seegmüller, J. Stoer, N. Wirth, Egidio Astesiano & Corrado Böhm, editors: *CAAP '81*, 112, Springer, pp. 211–223, doi:10.1007/3-540-10828-9_64.

[8] Shibashis Guha, Ismaël Jecker, Karoliina Lehtinen & Martin Zimmermann (2022): *Parikh Automata over Infinite Words*. In Anuj Dawar & Venkatesan Guruswami, editors: *42nd IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2022)*, *LIPIcs* 250, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 40:1–40:20, doi:10.4230/LIPICS.FSTTCS.2022.40.

[9] Christoph Haase & Simon Halfon (2014): *Integer Vector Addition Systems with States*. In Joël Ouaknine, Igor Potapov & James Worrell, editors: *Reachability Problems*, Springer, pp. 112–124, doi:10.1007/978-3-319-11439-2_9.

[10] Luisa Herrmann (2021): *Linear weighted tree automata with storage and inverse linear tree homomorphisms*. *Information and Computation* 281, p. 104816, doi:10.1016/j.ic.2021.104816.

[11] Luisa Herrmann, Vincent Peth & Sebastian Rudolph (2024): *Decidable (Ac)counting with Parikh and Muller: Adding Presburger Arithmetic to Monadic Second-Order Logic over Tree-Interpretable Structures*. In Aniello Murano & Alexandra Silva, editors: *32nd EACSL Annual Conference on Computer Science Logic (CSL 2024)*, *LIPIcs* 288, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 33:1–33:19, doi:10.4230/LIPICS.CSL.2024.33.

[12] John Hopcroft, Rajeev Motwani & Jeffrey Ullmann (2000): *Introduction to Automata Theory, Languages, and Computation*, 2 edition. Addison-Wesley.

[13] Felix Klaedtke (2004): *Automata-based decision procedures for weak arithmetics*. Ph.D. thesis, University of Freiburg. Available at `http://freidok.ub.uni-freiburg.de/volltexte/1439/index.html`.

[14] Felix Klaedtke & Harald Rueß (2002): *Parikh automata and monadic second-order logics with linear cardinality constraints*. Technical Report 177, Albert-Ludwigs-Universität Freiburg. (revised version).

[15] Felix Klaedtke & Harald Rueß (2003): *Monadic Second-Order Logics with Cardinalities*. In Jos C. M. Baeten, Jan Karel Lenstra, Joachim Parrow & Gerhard J. Woeginger, editors: *Automata, Languages and Programming, 30th International Colloquium (ICALP 2003)*, *LNCS* 2719, Springer, pp. 681–696, doi:10.1007/3-540-45061-0_54.

[16] Patrick Lincoln, John Mitchell, Andre Scedrov & Natarajan Shankar (1992): *Decision problems for propositional linear logic*. *Annals of Pure and Applied Logic* 56(1), pp. 239–311, doi:10.1016/0168-0072(92)90075-B.

[17] Marvin L. Minsky (1967): *Computation: finite and infinite machines*. Prentice-Hall, Inc.

[18] J. W. Thatcher & J. B. Wright (1968): *Generalized finite automata theory with an application to a decision problem of second-order logic*. *Mathematical systems theory* 2, pp. 57–81, doi:10.1007/BF01691346.

# Various Types of Comet Languages and their Application in External Contextual Grammars

Marvin Ködding Bianca Truthe
Institut für Informatik, Universität Giessen
Arndtstr. 2, 35392 Giessen, Germany
{marvin.koedding,bianca.truthe}@informatik.uni-giessen.de

In this paper, we continue the research on the power of contextual grammars with selection languages from subfamilies of the family of regular languages. We investigate various comet-like types of languages and compare such language families to some other subregular families of languages (finite, monoidal, nilpotent, combinational, (symmetric) definite, ordered, non-counting, power-separating, suffix-closed, commutative, circular, or union-free languages). Further, we compare the language families defined by these types for the selection with each other and with the families of the hierarchy obtained for external contextual grammars. In this way, we extend the existing hierarchy by new language families.

Keywords: Comet languages, contextual grammars, subregular selection languages, computational capacity.

## 1 Introduction

Contextual grammars were first introduced by Solomon Marcus in [16] as a formal model that might be used for the generation of natural languages. The derivation steps consist in adding contexts to given well-formed sentences, starting from an initial finite basis. Formally, a context is given by a pair $(u,v)$ of words and the external adding to a word $x$ gives the word $uxv$. In order to control the derivation process, contextual grammars with selection in a certain family of languages were defined. In such contextual grammars, a context $(u,v)$ may be added only around a word $x$ if this word $x$ belongs to a language which is associated with the context. Language families were defined where all selection languages in a contextual grammar belong to some language family $F$.

The study of external contextual grammars with selection in special regular sets was started by Jürgen Dassow in [6]. The research was continued by Jürgen Dassow, Florin Manea, and Bianca Truthe (see [8]) where further subregular families of selection languages were considered.

In the present paper, we extend the hierarchy of subregular language families by families of comet-like languages. Furthermore, we investigate the generative capacity of external contextual grammars with selection in such subregular language families.

## 2 Preliminaries

Throughout the paper, we assume that the reader is familiar with the basic concepts of the theory of automata and formal languages. For details, we refer to [22]. Here we only recall some notation, definitions, and previous results which we need for the present research.

An alphabet is a non-empty finite set of symbols. For an alphabet $V$, we denote by $V^*$ and $V^+$ the set of all words and the set of all non-empty words over $V$, respectively. The empty word is denoted by $\lambda$. For a word $w$ and a letter $a$, we denote the length of $w$ by $|w|$ and the number of occurrences of the letter $a$ in the word $w$ by $|w|_a$. For a set $A$, we denote its cardinality by $|A|$. The reversal of a word $w$ is denoted by $w^R$: if $w = x_1x_2\dots x_n$ for letters $x_1,\dots,x_n$, then $w^R = x_nx_{n-1}\dots x_1$. By $L^R$, we denote the language of all reversals of the words in $L$: $L^R = \{\, w^R \mid w \in L \,\}$.

A deterministic finite automaton is a quintuple

$$\mathcal{A} = (V, Z, z_0, F, \delta)$$

where $V$ is a finite set of input symbols, $Z$ is a finite set of states, $z_0 \in Z$ is the initial state, $F \subseteq Z$ is a set of accepting states, and $\delta$ is a transition function $\delta : Z \times V \to Z$. The language accepted by such an automaton is the set of all input words over the alphabet $V$ which lead letterwise by the transition function from the initial state to an accepting state.

A regular expression over some alphabet $V$ is defined inductively as follows:

1. $\emptyset$ is a regular expression;
2. every element $x \in V$ is a regular expression;
3. if $R$ and $S$ are regular expressions, so are the concatenation $R \cdot S$, the union $R \cup S$, and the Kleene closure $R^*$;
4. for every regular expression, there is a natural number $n$ such that the regular expression is obtained from the atomic elements $\emptyset$ and $x \in V$ by $n$ operations concatenation, union, or Kleene closure.

The language $L(R)$ which is described by a regular expression $R$ is also inductively defined:

1. $L(\emptyset) = \emptyset$;
2. for every element $x \in V$, we have $L(x) = \{x\}$;
3. if $R$ and $S$ are regular expressions, then

$$L(R \cdot S) = L(R) \cdot L(S), \quad L(R \cup S) = L(R) \cup L(S), \quad L(R^*) = (L(R))^*.$$

A general regular expression admits as operations (in the third item of the definition above) also intersection (where $L(R \cap S) = L(R) \cap L(S)$) and complementation (where $L(\overline{R}) = \overline{L(R)}$).

All the languages accepted by a finite automaton or described by some regular expression are called regular and form a family denoted by *REG*. Any subfamily of this set is called a subregular language family.

## 2.1 Some subregular language families

We consider the following restrictions for regular languages. In the following list of properties, we give already the abbreviation which denotes the family of all languages with the respective property. Let $L$ be a regular language over an alphabet $V$. With respect to the alphabet $V$, the language $L$ is said to be

- *monoidal* (*MON*) if and only if $L = V^*$,
- *nilpotent* (*NIL*) if and only if it is finite or its complement $V^* \setminus L$ is finite,
- *combinational* (*COMB*) if and only if it has the form $L = V^*X$ for some subset $X \subseteq V$,
- *definite* (*DEF*) if and only if it can be represented in the form $L = A \cup V^*B$ where $A$ and $B$ are finite subsets of $V^*$,

- *symmetric definite* (*SYDEF*) if and only if $L = EV^*H$ for some regular languages $E$ and $H$,
- *suffix-closed* (*SUF*) (or *fully initial* or *multiple-entry* language) if and only if, for any two words over $V$, say $x \in V^*$ and $y \in V^*$, the relation $xy \in L$ implies the relation $y \in L$,
- *ordered* (*ORD*) if and only if the language is accepted by some deterministic finite automaton

$$\mathcal{A} = (V, Z, z_0, F, \delta)$$

  with an input alphabet $V$, a finite set $Z$ of states, a start state $z_0 \in Z$, a set $F \subseteq Z$ of accepting states and a transition mapping $\delta$ where $(Z, \preceq)$ is a totally ordered set and, for any input symbol $a \in V$, the relation $z \preceq z'$ implies $\delta(z,a) \preceq \delta(z',a)$,
- *commutative* (*COMM*) if and only if it contains with each word also all permutations of this word,
- *circular* (*CIRC*) if and only if it contains with each word also all circular shifts of this word,
- *non-counting* (*NC*) if and only if there is a natural number $k \geq 1$ such that, for any three words $x \in V^*$, $y \in V^*$, and $z \in V^*$, it holds $xy^kz \in L$ if and only if $xy^{k+1}z \in L$,
- *star-free* (*SF*) if and only if $L$ can be described by a regular expression which is built by concatenation, union, and complementation,
- *power-separating* (*PS*) if and only if, there is a natural number $m \geq 1$ such that for any word $x \in V^*$, either $J_x^m \cap L = \emptyset$ or $J_x^m \subseteq L$ where $J_x^m = \{\, x^n \mid n \geq m \,\}$,
- *union-free* (*UF*) if and only if $L$ can be described by a regular expression which is only built by concatenation and Kleene closure,
- *star* (*STAR*) if and only if $L = H^*$ for some regular language $H \subseteq V^*$,
- *left-sided comet* (*LCOM*) if and only if $L = EG^*$ for some regular language $E$ and a regular language $G \notin \{\emptyset, \{\lambda\}\}$,
- *right-sided comet* (*RCOM*) if and only if $L = G^*H$ for some regular language $H$ and a regular language $G \notin \{\emptyset, \{\lambda\}\}$,
- *two-sided comet* (*2COM*) if and only if $L = EG^*H$ for two regular languages $E$ and $H$ and a regular language $G \notin \{\emptyset, \{\lambda\}\}$.

We remark that monoidal, nilpotent, combinational, (symmetric) definite, ordered, non-counting, star-free, union-free, star, and (left-, right-, or two-sided) comet languages are regular, whereas non-regular languages of the other types mentioned above exist. Here, we consider among the suffix-closed, commutative, circular, and power-separating languages only those which are also regular. By *FIN*, we denote the family of languages with finitely many words. In [17], it was shown that the families of the non-counting languages and the star-free languages are equivalent ($NC = SF$).

Some properties of the languages of the classes mentioned above can be found in [23] (monoids), [10] (nilpotent languages), [13] (combinational and commutative languages), [21] (definite languages), [20] (symmetric definite languages), [11] and [5] (suffix-closed languages), [24] (ordered languages), [15] (circular languages), [17] (non-counting and star free languages), [25] (power-separating languages), [2] (union-free languages), [3] (star languages), [4] (comet languages).

## 2.2 Contextual grammars

Let $\mathcal{F}$ be a family of languages. A contextual grammar with selection in $\mathcal{F}$ is a triple $G = (V, \mathcal{S}, A)$ where

- $V$ is an alphabet,
- $\mathcal{S}$ is a finite set of selection pairs $(S, C)$ with a selection language $S$ over some subset $U$ of the alphabet $V$ which belongs to the family $\mathcal{F}$ with respect to the alphabet $U$ and a finite set $C \subset V^* \times V^*$ of contexts where, for each context $(u, v) \in C$, at least one side is not empty: $uv \neq \lambda$,
- $A$ is a finite subset of $V^*$ (its elements are called axioms).

We write a selection pair $(S, C)$ also as $S \to C$. In the case that $C$ is a singleton set $C = \{(u, v)\}$, we also write $S \to (u, v)$. For a contextual grammar $G = (V, \{ (S_1, C_1), (S_2, C_2), \ldots, (S_n, C_n) \}, A)$, we set

$$\ell_A(G) = \max \{ |w| \mid w \in A \}, \quad \ell_C(G) = \max \{ |uv| \mid (u, v) \in C_i, 1 \leq i \leq n \}, \quad \ell(G) = \ell_A(G) + \ell_C(G) + 1.$$

We now define the derivation modes for contextual grammars with selection.

Let $G = (V, \mathcal{S}, A)$ be a contextual grammar with selection. A direct external derivation step in $G$ is defined as follows: a word $x$ derives a word $y$ (written as $x \Longrightarrow y$) if and only if there is a pair $(S, C) \in \mathcal{S}$ such that $x \in S$ and $y = uxv$ for some pair $(u, v) \in C$. Intuitively, one can only wrap a context $(u, v) \in C$ around a word $x$ if $x$ belongs to the corresponding selection language $S$.

By $\Longrightarrow^*$ we denote the reflexive and transitive closure of the relation $\Longrightarrow$. The language generated by $G$ is $L = \{ z \mid x \Longrightarrow^* z \text{ for some } x \in A \}$.

**Example 1** Consider the contextual grammar $G = (\{a, b, c\}, \{(S_1, C_1), (S_2, C_2)\}, \{\lambda\})$ with

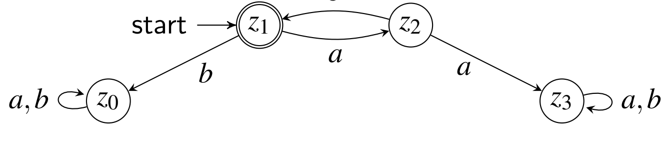
$$S_1 = \{a, b\}^*, \quad C_1 = \{(\lambda, a), (\lambda, b)\}, \qquad S_2 = \{ab\}^*, \quad C_2 = \{(c, c)\}.$$

Starting from the axiom $\lambda$, every word of the language $S_1$ is generated by applying the first selection. Starting from any word of $S_2 \subset S_1$, every word of the language $\{c\}S_2\{c\}$ is generated by applying the second selection. Other words are not generated.

Thus, the language generated is

$$L(G) = \{a, b\}^* \cup \{ c(ab)^n c \mid n \geq 0 \}.$$

Both selection languages are ordered: The language $S_1$ is accepted by a finite automaton with exactly one state. Hence, it is ordered. The language $S_2$ is accepted by the following deterministic finite automaton $A = (\{z_0, z_1, z_2, z_3\}, \{a, b\}, \delta, z_1, \{z_1\})$ where the transition function is illustrated in the following picture and given in the table next to it, from which it can be seen that the automaton is ordered:



| | $z_0$ | $z_1$ | $z_2$ | $z_3$ |
|---|---|---|---|---|
| $a$ | $z_0$ | $z_2$ | $z_3$ | $z_3$ |
| $b$ | $z_0$ | $z_0$ | $z_1$ | $z_3$ |

◊

By $\mathcal{EC}(\mathcal{F})$, we denote the family of all languages generated externally by contextual grammars with selection in $\mathcal{F}$. When a contextual grammar works in the external mode, we call it an external contextual grammar.

The language generated by the external contextual grammar in Example 1 belongs, for instance, to the family $\mathcal{EC}(ORD)$ because all selection languages ($S_1$ and $S_2$) are ordered.

## 3 Results on families of comet languages

We first present some observations about star languages and two-sided comet languages, we give normal forms for two-sided comets, and we insert the subregular families investigated here into the existing hierarchy.

From the structure of two-sided comet languages (languages $L$ of the form $EG^*H$ where $G$ is neither the empty set nor the set with the empty word only), we see that every such language is infinite if none of the sets $E$, $G$, and $H$ is the empty set. If one of the sets $E$ or $H$ is empty, then the whole language $L$ is also empty.

**Lemma 2** *For each language $L \in$ 2COM, it holds that $L$ is either infinite or empty.*

A similar observation can be made for star languages.

**Lemma 3** *For each language $L \in$ STAR, it holds that $L$ either is infinite or consists of the empty word $\lambda$.*

### 3.1 Normal forms

We first show some observations before we conclude a normal form for languages from the class *2COM*. This normal form is later used when we prove that *2COM*-languages as selection languages are as powerful as arbitrary regular languages.

**Lemma 4** *Each two-sided comet language $L = EG^*H$ can be represented as a finite union*

$$L = \bigcup_{i=1}^{n} E_i G^* H$$

*for some number $n \geq 1$ and with union-free languages $E_i$ for all $1 \leq i \leq n$.*

*Proof.* Let $L = EG^*H$ be a two-sided comet language. Every regular language is the union of finitely many union-free languages [18]. Let $n \geq 1$ be a natural number and $E_i$ be a union-free language for any $i$ with $1 \leq i \leq n$ such that $E = E_1 \cup E_2 \cup \cdots \cup E_n$. Then, it follows $L = E_1G^*H \cup E_2G^*H \cup \cdots \cup E_nG^*H$. □

In order to show later that we can transform any *2COM*-language into the mentioned normal form, we now present how an infinite union-free language can be represented by a special *2COM*-form.

**Lemma 5** *For an infinite union-free language $L$, there exist sets $L_l$, $L_i$, and $L_r$ such that $L = L_l L_i^* L_r$ where $L_l$ is finite and $L_i \notin \{\emptyset, \{\lambda\}\}$.*

*Proof.* We prove the assertion inductively via the number of construction steps required to create a regular expression $\mathcal{R}$ such that $L = L(\mathcal{R})$ holds. In construction step 0, only finite languages are created. Therefore, the base case is $n = 1$.

*Base case* $n = 1$: Since $L$ is infinite, we have $\mathcal{R} = \{x\}^*$ for a letter $x \in V$. A desired representation for the language $L$ is then $\{\lambda\}\{x\}^*\{\lambda\}$.

*Induction step* $n \to n+1$: Assume the induction hypothesis: For every regular expression $\mathcal{R}$ without the union operator which describes an infinite language and which is at construction level of at most $n$, the language $L(\mathcal{R})$ can be represented as $L(\mathcal{R}) = L_l L_i^* L_r$ with $|L_l| < \infty$ and $L_i \notin \{\emptyset, \{\lambda\}\}$. Now, let $\mathcal{R}$ be a regular expression of construction level $n+1$ which describes an infinite language and which does not contain the union operator. Then, there are two possibilities how $\mathcal{R}$ is built: by concatenation of two regular expressions where for at least one of the described languages the induction hypothesis holds or by Kleene closure of a regular expression which neither describes the empty set nor the language $\{\lambda\}$ (otherwise, $L(\mathcal{R})$ would be finite).

*Case 1*: Let $\mathcal{R} = \mathcal{S}\mathcal{T}$. Then, the equation $L(\mathcal{R}) = L(\mathcal{S})L(\mathcal{T})$ holds. If $L(\mathcal{S})$ is infinite, we get, according to the induction hypothesis, $L(\mathcal{S}) = S_l S_i^* S_r$ for suitable sets $S_l$, $S_i$, and $S_r$. With $R_l = S_l$, $R_i = S_i$, and $R_r = S_r L(\mathcal{T})$, we obtain $L(\mathcal{R}) = R_l R_i^* R_r$ with $|R_l| < \infty$ and $R_i \notin \{\emptyset, \{\lambda\}\}$. If $L(\mathcal{S})$ is finite, then $L(\mathcal{T})$ is infinite (because we consider only such $\mathcal{R}$ where $L(\mathcal{R})$ is infinite) and we get, according to the induction hypothesis, that $L(\mathcal{T}) = T_l T_i^* T_r$ for suitable sets $T_l$, $T_i$, and $T_r$. With $R_l = L(\mathcal{S})T_l$, $R_i = T_i$, and $R_r = T_r$, we obtain a desired representation $L(\mathcal{R}) = R_l R_i^* R_r$ with $|R_l| < \infty$ and $R_i \notin \{\emptyset, \{\lambda\}\}$.

*Case 2*: Let $\mathcal{R} = \mathcal{S}^*$. Then, the equation $L(\mathcal{R}) = (L(\mathcal{S}))^*$ holds. Thus, with $R_l = \{\lambda\}$, $R_i = L(\mathcal{S})$, and $R_r = \{\lambda\}$, we obtain that $L(\mathcal{R}) = R_l R_i^* R_r$ with $|R_l| < \infty$ and $R_i \notin \{\emptyset, \{\lambda\}\}$.

Hence, every infinite union-free language can be expressed in the claimed form. □

We proved with Lemma 4 that any *2COM*-language can be given as a union of finitely many *2COM*-languages where the first comet tail is always union-free. Together, we obtain that any *2COM*-language has a representation in the *2COM*-form where the first comet tail is a finite set.

**Lemma 6** *For each two-sided comet language $L = EG^*H$ with $E \in UF$, there exist a finite language $E'$, a language $G' \notin \{\emptyset, \{\lambda\}\}$, and a regular language $H'$ such that $L = E'(G')^*H'$.*

*Proof.* We have shown in Lemma 2 that each two-sided comet language $L$ is either empty or infinite. For the first case, the assertion holds with $E' = \emptyset$ and any regular languages $G' \notin \{\emptyset, \{\lambda\}\}$ and $H'$.

Now, let $L = EG^*H$ be an infinite *2COM*-language with $E \in UF$. If $E$ is finite, then we already have a desired form with $E' = E$, $G' = G$, and $H' = H$.

So, let $E$ be infinite. By Lemma 5, we know that there are languages $E_l$, $E_i$, and $E_r$ such that $E_l$ is a finite set, $E_i \notin \{\emptyset, \{\lambda\}\}$, and $E = E_l E_i^* E_r$. If we set $E' = E_l$, $G' = E_i$, and $H' = E_r G^* H$, then we obtain a desired form because $L = E'(G')^*H'$ where $E'$ is finite, $G' \notin \{\emptyset, \{\lambda\}\}$, and $H'$ is a regular language. □

Now we connect the previous lemmas and conclude that, for every two-sided comet language, there is such a representation where the first comet tail of the language is finite.

**Theorem 7 (Normal form for *2COM*-languages)** *For each two-sided comet language, there exists a representation $L = EG^*H$ such that $E$ is a finite language and $G \notin \{\emptyset, \{\lambda\}\}$.*

*Proof.* According to Lemma 4, any two-sided comet language $L = E'(G')^*H'$ can be represented as a union of finitely many languages $E_i'(G')^*H'$ such that all languages $E_i'$ are union-free. According to Lemma 6, every such language $E_i'(G')^*H'$ can in turn be represented as a *2COM*-language $E_i G^* H$ where the first tail $E_i$ is finite. The union $E$ of all these finite languages $E_i$ is also finite. Hence, we obtain

$$L = E'(G')^*H' = \bigcup_{i=1}^{n} E_i'(G')^*H' = \bigcup_{i=1}^{n} E_i G^* H = \left(\bigcup_{i=1}^{n} E_i\right) G^* H = EG^*H$$

where $E$ is finite and $G \notin \{\emptyset, \{\lambda\}\}$. □

We refer to this representation as a left-sided normal form. A right-sided normal form (where the last comet tail is a finite set) can be derived in a similar way.

## 3.2 Hierarchy of subregular language classes

In this section, we investigate inclusion relations between various subregular languages classes. Figure 1 shows the results.

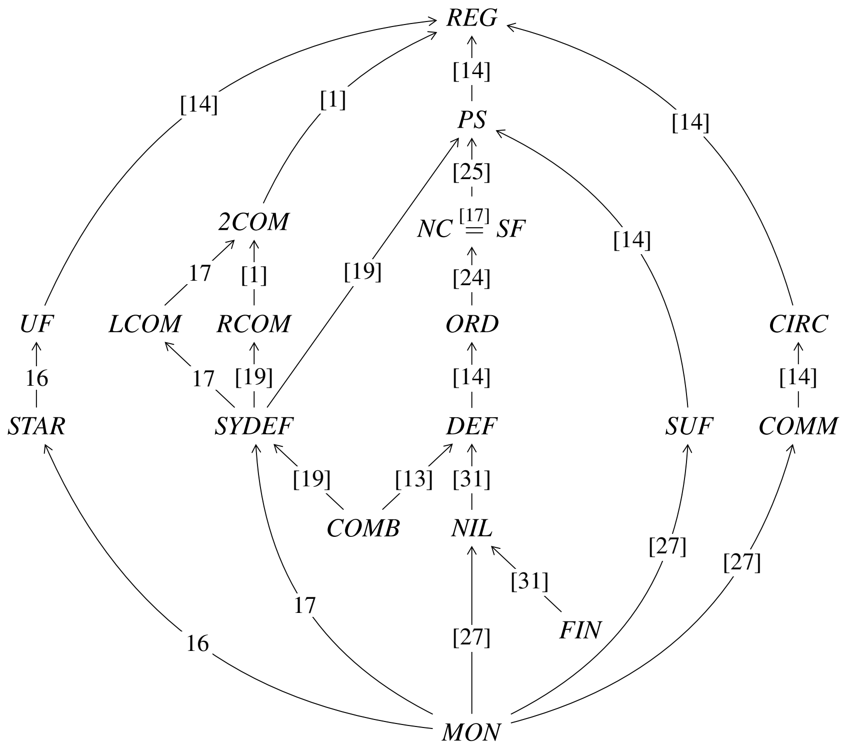


Figure 1: Resulting hierarchy of subregular language families

An arrow from a node $X$ to a node $Y$ stands for the proper inclusion $X \subset Y$. If two families are not connected by a directed path, then they are incomparable. An edge label refers to the paper where the proper inclusion has been shown (in some cases, it might be that it is not the first paper where the respective inclusion has been mentioned, since it is so obvious that it was not emphasized in a publication) or the lemma of this paper where the proper inclusion will be shown.

In the literature, it is often said that two languages are equivalent if they are equal or differ at most in the empty word. Similarly, two families can be regarded to be equivalent if they differ only in the languages $\emptyset$ or $\{\lambda\}$. Therefore, the set *STAR* of all star languages is sometimes regarded as a proper subset of the set *COM* of all (left-, right-, or two-sided) comet languages although $\{\lambda\}$ belongs to the family *STAR* but not to *LCOM*, *RCOM* or *2COM*. We regard *STAR* and $\mathit{STAR} \setminus \{\{\lambda\}\}$ as different. Then, the family *STAR* is incomparable to *LCOM*, *RCOM*, and *2COM*, as we will later show.

For space reasons, we give the following observation without a proof.

**Lemma 8** *Whenever a language $L$ is a right-sided comet then its reversal $L^R$ is a left-sided comet language and vice versa.*

**Corollary 9** *We have $\mathit{LCOM} = \{\, L^R \mid L \in \mathit{RCOM} \,\}$ and $\mathit{RCOM} = \{\, L^R \mid L \in \mathit{LCOM} \,\}$.*

We now present some languages which will serve later as witness languages for proper inclusions or incomparabilities.

**Lemma 10** *The language $L = \{\lambda\}$ is in $STAR \setminus 2COM$.*

*Proof.* The language $L$ is a star language since $L = H^*$ with $H = \{\lambda\}$. According to Lemma 2, a two-sided comet language is either infinite or the empty language. Hence, $L$ is not a two-sided comet. □

**Lemma 11** *Let $L = \{\, a^{2n} \mid n \geq 0 \,\}$. Then, it holds $L \in (STAR \cap LCOM \cap RCOM) \setminus PS$.*

*Proof.* Let $G = \{aa\}$ and $E = H = \{\lambda\}$. The language $L$ can be expressed as $L = G^* = EG^* = G^*H$. Therefore, $L \in STAR \cap LCOM \cap RCOM$.

Assume that $L \in PS$. Then, there is a natural number $m \geq 1$ such that, for any word $x \in \{a\}^*$, either $J_x^m \cap L = \emptyset$ or $J_x^m \subseteq L$ where $J_x^m = \{\, x^n \mid n \geq m \,\}$. For any natural number $m \geq 1$, we have with the word $x = a$ the set $J_a^m = \{\, a^n \mid n \geq m \,\}$. Since $a^{2m} \in J_a^m \cap L$, the intersection is not empty. But, since $a^{2m+1} \in J_a^m \setminus L$, it neither holds $J_a^m \subseteq L$. Hence, the language $L$ is not power-separating. □

**Lemma 12** *Let $L = \{ab\}^*$. Then, it holds $L \in (STAR \cap LCOM \cap RCOM) \setminus CIRC$.*

*Proof.* Let $G = \{ab\}$ and $E = H = \{\lambda\}$. The language $L$ can be expressed as $L = G^* = EG^* = G^*H$. Therefore, $L \in STAR \cap LCOM \cap RCOM$.

Assume that the language $L$ is circular. Then, the word $ba$ would belong to it because $ab \in L$ but it does not. Hence, $L \notin CIRC$. □

**Lemma 13 ([19])** *Let $V = \{a, b\}$ be an alphabet, $H = \{ba\}\{b\}^*(\{aa\}\{b\}^*)^*$ a regular language over $V$, and $L = V^*H$. Then, $L \in SYDEF \setminus SF$.*

*Proof.* The language $L$ can be represented as $\{\lambda\}V^*H$. So, the language is symmetric definite. As shown in [19], the language is not star-free. □

**Lemma 14** *Let $L_1 = \{\, a^n b \mid n \geq 0 \,\}$ and $L_2 = L_1^R$. Then, $L_1 \in RCOM \setminus LCOM$ and $L_2 \in LCOM \setminus RCOM$.*

*Proof.* The language $L_1$ can be expressed as $\{a\}^*\{b\}$, hence, in the form $L_1 = G^*H$ with $G = \{a\}$ and $H = \{b\}$. Thus, $L_1 \in RCOM$.

Assume that $L_1 \in LCOM$. Then, two languages $E$ and $I$ would exist such that $L_1 = EI^*$. Since $b$ is a suffix of every word in $L_1$, the letter $b$ is also a suffix of a word in $I$. But then $L_1$ would also contain a word with more than one $b$ which is a contradiction. Hence, $L_1 \notin LCOM$.

By Corollary 9, it follows that $L_2 \in LCOM \setminus RCOM$. □

**Lemma 15** *The language $L = \{\lambda, a\}$ belongs to the set $(FIN \cap SUF \cap COMM) \setminus (STAR \cup 2COM)$.*

*Proof.* All suffixes of all words of the language $L$ belong to $L$. Thus, $L$ is suffix-closed. Furthermore, the language is finite but not empty and commutative. According to Lemma 2, each two-sided comet language is either empty or infinite. Hence, $L$ is not a two-sided comet language. According to Lemma 3, each star language is either infinite or contains only the empty word. Hence, $L$ is not a star language either. □

We now prove some proper inclusions.

**Lemma 16** *We have the proper inclusions $MON \subset STAR \subset UF$.*

*Proof.* We first prove the relation $MON \subset STAR$: Any monoidal language can be expressed as $L = V^*$ for some alphabet $V$. Since $V$ is a regular language, $L$ is a star language. A witness language for the properness is the language $L = \{\, a^{2n} \mid n \geq 0 \,\}$ as shown in Lemma 11.

We now prove the relation $STAR \subset UF$: Every language $H^*$ for some regular language $H$ is union-free according to [18]. A witness language for the properness is $L = \{a\}$ which is union-free but, according to Lemma 3, not a star language since it is neither infinite nor equal to $\{\lambda\}$. □

**Lemma 17** *We have the proper inclusions $MON \subset SYDEF \subset \mathcal{C} \subset 2COM$ for $\mathcal{C} \in \{LCOM, RCOM\}$.*

*Proof.*

1. $MON \subset SYDEF$: Any monoidal language can be expressed as $L = V^*$ for some alphabet $V$ and, with $E = H = \{\lambda\}$ also in the form $EV^*H$. Hence, the language $L$ is symmetric definite. A witness language for the properness is $\{a,b\}^*\{ba\}\{b\}^*(\{aa\}\{b\}^*)^*$ from Lemma 13 (and originally [19]).
2. $SYDEF \subset RCOM$: This relation was proved in [19].
3. $SYDEF \subset LCOM$: The family $SYDEF$ is closed under reversal. For any symmetric definite language $L$, its reversal $L^R$ also belongs to the family $SYDEF$ and, by [19], is also a right-sided comet language. By Lemma 8, the reversal of the language $L^R$, hence $L$ itself, is a left-sided comet language. A witness language for the properness is the language $L = \{\, a^{2n} \mid n \geq 0 \,\}$ according to Lemma 11 where it is shown that $L \in LCOM \setminus PS$ and according to [19] where the inclusion $SYDEF \subset PS$ is proved.
4. $RCOM \subset 2COM$: This relation was proved in [19].
5. $LCOM \subset 2COM$: Any left-sided comet language $L = EG^*$ is also a two-sided comet $EG^*H$ with $H = \{\lambda\}$. In Lemma 14, it was shown that the language $L = \{\, a^n b \mid n \geq 0 \,\}$ is a right-sided comet language but not a left-sided comet. By [19], it is a two-sided comet language. □

We now prove the incomparability relations mentioned in Figure 1 which have not been proved earlier. These are the relations regarding the families $STAR$, $SYDEF$, $LCOM$, $RCOM$, and $2COM$.

**Lemma 18** *Each of the families STAR and UF is incomparable to each of the families COMB, SYDEF, RCOM, LCOM, and 2COM.*

*Proof.* Due to inclusion relations, it suffices to show that there are a language $L_1 \in STAR \setminus 2COM$ and a language $L_2 \in COMB \setminus UF$. From Lemma 10, we get $L_1 = \{\lambda\}$. From [14], we take $L_2 = \{a,b,c\}^*\{a,b\}$. □

**Lemma 19** *The language family STAR is incomparable to each of the families FIN, NIL, DEF, ORD, NC, SF, PS, and SUF.*

*Proof.* Due to inclusion relations, it suffices to show that there are a language $L_1 \in STAR \setminus PS$, a language $L_2 \in FIN \setminus STAR$, and a language $L_3 \in SUF \setminus STAR$. As $L_1$, we obtain from Lemma 11 the language $L_1 = \{\, a^{2n} \mid n \geq 0 \,\}$. From Lemma 15, we take $L_2 = L_3 = \{\lambda, a\}$. □

**Lemma 20** *The language family STAR is incomparable to the families CIRC and COMM.*

*Proof.* Due to inclusion relations, it suffices to show that there are a language $L_1 \in \mathit{STAR} \setminus \mathit{CIRC}$ and a language $L_2 \in \mathit{COMM} \setminus \mathit{STAR}$. From Lemma 12, we have $L_1 = \{ab\}^*$. From Lemma 15, we take again the language $L_2 = \{\lambda, a\}$. □

**Lemma 21** *The language families LCOM and RCOM are incomparable to each other.*

*Proof.* With the witness languages $L_1 = \{\, a^n b \mid n \geq 0 \,\} \in \mathit{RCOM} \setminus \mathit{LCOM}$ and $L_2 = L_1^R \in \mathit{LCOM} \setminus \mathit{RCOM}$, the statement follows from Lemma 14. □

**Lemma 22** *The language families SYDEF, LCOM, RCOM, and 2COM are incomparable to each of the families FIN, NIL, DEF, ORD, NC, and SF.*

*Proof.* Due to inclusion relations, it suffices to show that there are a language $L_1 \in \mathit{SYDEF} \setminus \mathit{SF}$ and a language $L_2 \in \mathit{FIN} \setminus \mathit{2COM}$. From Lemma 13 (and previously [19]), for the first language, we obtain the language $L_1 = \{a,b\}^*\{ba\}\{b\}^*(\{aa\}\{b\}^*)^*$. From Lemma 15, we take the language $L_2 = \{\lambda, a\}$. □

**Lemma 23** *The language families LCOM, RCOM, and 2COM are incomparable to the family PS.*

*Proof.* Due to inclusion relations, it suffices to show that there are a language $L_1 \in \mathit{LCOM} \setminus \mathit{PS}$ and a language $L_2 \in \mathit{PS} \setminus \mathit{2COM}$. The property of (non) power-separating is not influenced by the reversal operation. If there is a language $L_1 \in \mathit{LCOM} \setminus \mathit{PS}$, then there is also a language in the set $\mathit{RCOM} \setminus \mathit{PS}$, namely $L_1^R$. From Lemma 11, we have $L_1 = \{\, a^{2n} \mid n \geq 0 \,\} \in (\mathit{LCOM} \cap \mathit{RCOM}) \setminus \mathit{PS}$. As language $L_2$, we take again the language $L_2 = \{\lambda, a\}$ from Lemma 15. □

**Lemma 24** *The language families SYDEF, LCOM, RCOM, and 2COM are incomparable to each of the families SUF, CIRC and COMM.*

*Proof.* Due to inclusion relations, it suffices to show that there are a language $L_1 \in \mathit{SYDEF} \setminus \mathit{SUF}$, a language $L_2 \in \mathit{SYDEF} \setminus \mathit{CIRC}$, a language $L_3 \in \mathit{SUF} \setminus \mathit{2COM}$, and a language $L_4 \in \mathit{COMM} \setminus \mathit{2COM}$. In [14], it was shown that the families *COMB* and *SUF* are disjoint. Since $\mathit{COMB} \subseteq \mathit{SYDEF}$, we can take any combinational language as $L_1$, for instance, $L_1 = \{a,b\}^*\{b\}$. The same language serves as $L_2$ because it is not circular. From Lemma 15, we take again the language $\{\lambda, a\}$ as $L_3$ and $L_4$. □

From all these relations, the hierachy presented in Figure 1 follows.

**Theorem 25 (Resulting hierarchy)** *The inclusion relations presented in Figure 1 hold. An arrow from an entry $X$ to an entry $Y$ depicts the proper inclusion $X \subset Y$; if two families are not connected by a directed path, then they are incomparable.*

*Proof.* An edge label refers to the paper or lemma in the present paper where the proper inclusion is shown. The incomparability results are proved in Lemmas 18 to 24. □

## 4 Results on subregular control in external contextual grammars

In this section, we include the families of languages generated by external contextual grammars with selection languages from the subregular families under investigation into the existing hierarchy with respect to external contextual grammars.

If, in a contextual grammar, all selection languages belong to some language family $X$, then they belong also to every super set $Y$ of $X$. Therefore, each language in $\mathcal{EC}(X)$ is also generated by a contextual grammar with selection languages from $Y$ and we have the following monotonicity.

**Lemma 26** *For any two language classes X and Y with $X \subseteq Y$, we have the inclusion $\mathcal{EC}(X) \subseteq \mathcal{EC}(Y)$.*

Figure 2 shows a hierarchy of some language families which are generated by external contextual grammars where the selection languages belong to subregular classes investigated before. The hierarchy contains results which were already known (marked by a reference to the literature) and results which will be proved in this section (marked by a number which refers to the respective lemma).

$$\begin{tikzcd}
 & & \mathcal{EC}(REG) \overset{[8]}{=} \mathcal{EC}(UF) \overset{35}{=} \mathcal{EC}(LCOM) \overset{35}{=} \mathcal{EC}(RCOM) \overset{35}{=} \mathcal{EC}(2COM) & & \\
 & & \mathcal{EC}(PS) \arrow[u, "{[28]}"] & & \\
 & & \mathcal{EC}(NC) \arrow[u, "{[28]}"] & \mathcal{EC}(CIRC) \arrow[uul, "{[8]}"'] & \\
 & \mathcal{EC}(SYDEF) \arrow[uur, "40"] & \mathcal{EC}(ORD) \arrow[u, "{[9]}"] & \mathcal{EC}(COMM) \arrow[u, "{[8]}"] & \\
\mathcal{EC}(SUF) \arrow[uuurr, bend left, "{[28]}"] & & \mathcal{EC}(DEF) \arrow[u, "{[26]}"] \arrow[ul, "39"] & & \mathcal{EC}(STAR) \arrow[uuuull, bend right, "38"'] \\
 & & \mathcal{EC}(COMB) \arrow[u, "{[28]}"] & \mathcal{EC}(NIL) \arrow[ul, "{[6]}"] \arrow[uu, "{[6]}"] & \\
 & & \mathcal{EC}(MON) \arrow[u, "{[7]}"] \arrow[ur, "{[6]}"] \arrow[uull, bend left, "{[6]}"] \arrow[uurr, bend right, "36"'] & & \\
 & & \mathcal{EC}(FIN) \arrow[u, "{[6]}"] & &
\end{tikzcd}$$

Figure 2: Resulting hierarchy of language families by external contextual grammars with special selection languages

An arrow from a node $X$ to a node $Y$ stands for the proper inclusion $X \subset Y$. If two families are not connected by a directed path, then they are incomparable. An edge label refers to the paper where the proper inclusion has been shown or the lemma of this paper where the proper inclusion will be shown.

We now present some languages which will serve later as witness languages for proper inclusions or incomparabilities. Due to space limitations, we give only proof sketches in some cases where we believe that the reader finds the idea feasible.

**Lemma 27** *Let $L = \{ a^n bbb \mid n \geq 1 \} \cup \{\lambda\}$. Then, it holds $L \in \mathcal{EC}(NIL) \setminus \mathcal{EC}(STAR)$.*

*Proof.* The contextual grammar $G = (\{a,b\}, \{\{a,b\}^*\{a,b\}^4 \to (a,\lambda)\}, \{abbb, \lambda\})$ generates $L$.

During the derivation, the number of the letter $a$ is increasing without changing the number of $b$. If the selection languages are from *STAR*, then such a context containing letters $a$ only could be wrapped around the empty word yielding a word without $b$ which is a contradiction. □

**Lemma 28** *Let $L = \{ b^n a \mid n \geq 0 \} \cup \{\lambda\}$. Then, it holds $L \in \mathcal{EC}(COMB) \setminus \mathcal{EC}(STAR)$.*

*Proof.* The contextual grammar $G = (\{a,b\}, \{\{a,b\}^*\{a\} \to (b,\lambda)\}, \{\lambda, a\})$ generates the language $L$ and the selection language is combinational.

Similarly to the proof before: With star selection languages, a word with the letter $b$ but without $a$ could be generated. □

**Lemma 29** *Let $L_1 = \{a,b\}^*\{ a^n b^m \mid n \geq 1,\ m \geq 1 \}$, $L_2 = \{ ca^n b^m c \mid n \geq 1,\ m \geq 1 \}$, and $L = L_1 \cup L_2$. Then, it holds $L \in \mathcal{EC}(SUF) \setminus \mathcal{EC}(STAR)$.*

*Proof.* It holds $L = L(G)$ for the contextual grammar $G = (\{a,b,c\}, \{(S_1,C_1),(S_2,C_2)\}, \{ab\})$ with

$$S_1 = \{a,b\}^*, \quad C_1 = \{(a,\lambda),(b,\lambda),(\lambda,b)\}, \qquad S_2 = \{ a^n b^m \mid n \geq 0,\ m \geq 1 \} \cup \{\lambda\}, \quad C_2 = \{(c,c)\}.$$

Using star selection languages, the two letters $c$ could be wrapped around a word with more than one $a$-to-$b$-change from $L_1$ which would yield a word not belonging to $L$. □

**Lemma 30** *Let $L_1 = \{ a^n \mid n \geq 2 \}$ and $L_2 = \{ ba^{2n}b \mid n \geq 1 \}$ be two languages and $L = L_1 \cup L_2$ its union. Then, the relation $L \in \mathcal{EC}(STAR) \setminus \mathcal{EC}(PS)$ holds.*

*Proof.* It holds $L = L(G)$ for the contextual grammar $G = (\{a,b\}, \{(S_1,C_1),(S_2,C_2)\}, \{aa\})$ with

$$S_1 = \{ a^n \mid n \geq 0 \}, \quad C_1 = \{(\lambda,a)\} \quad \text{and} \quad S_2 = \{ a^{2n} \mid n \geq 0 \}, \quad C_2 = \{(b,b)\}.$$

Now assume that $L \in \mathcal{EC}(PS)$. Then, $L = L(G')$ for a contextual grammar $G'$ where every selection language is power-separating.

For every selection language (since it is power-separating), there is a number $m_S \in \mathbb{N}$ such that, for every word $x \in \{a,b\}^*$, either $J_x^{m_S} \cap S = \emptyset$ or $J_x^{m_S} \subseteq S$ with $J_x^{m_S} = \{ x^n \mid n \geq m_S \}$. Let $m_S$ be the minimum of these numbers for $S$ and let $m$ be the maximum of all the values $m_S$ for a selection language $S$.

Further, let $p = m + \ell(G')$. Then, we have the following statement for every selection language $S$: For each word $x \in \{a,b\}^*$, it is

$$\text{either } J_x^p \cap S = \emptyset \text{ or } J_x^p \subseteq S \tag{1}$$

where $J_x^p = \{ x^n \mid n \geq p \}$.

The language $L_2$ contains words with an arbitrary even number of letters $a$ and a letter $b$ at each end. Hence, there is a derivation $w_0 \Longrightarrow^* w_1 \Longrightarrow uw_1v$ with $w_0 \in A$, $|w_1|_a > p$, $|w_1|_b = 0$, and $|uv|_b > 0$. This implies $w_1 = a^k$ with $k > p$.

Let $S$ be the selection language used in the last derivation step. Then, we have $a^k \in S$ and, with property (1), also $a^{k+1} \in S$. Since $a^{k+1}$ belongs to $L_1$ and therefore also to $L$, the last derivation step can also be applied to $a^{k+1}$ which yields the word $ua^{k+1}v$. Since $|uv|_b > 0$, the word $ua^{k+1}v$ belongs at most to $L_2$. Since $ua^k v \in L_1$, we know that $|ua^k v|_a$ is an even number and $|ua^{k+1}v|_a$ is an odd number. Therefore, the word $ua^{k+1}v$ does not belong to $L_2$ and neither to $L$ which is a contradiction to $L = L(G')$. Thus, we conclude $L \notin \mathcal{EC}(PS)$. □

**Lemma 31** *Let $L = \{ a^n b^n \mid n \geq 1 \} \cup \{ b^n a^n \mid n \geq 1 \}$. Then, it holds $L \in \mathcal{EC}(STAR) \setminus \mathcal{EC}(CIRC)$.*

*Proof.* It holds $L = L(G)$ for the contextual grammar $G = (\{a,b\}, \{(S_1,C_1),(S_2,C_2)\}, \{ab,ba\})$ with

$$S_1 = \{ a^n b^m \mid n \geq 1,\ m \geq 1 \}^*, \quad C_1 = \{(a,b)\} \quad \text{and} \quad S_2 = \{ b^n a^m \mid n \geq 1,\ m \geq 1 \}^*, \quad C_2 = \{(b,a)\}.$$

With circular selection languages, a context $(a^k, b^k)$ could be wrapped around a word $b^m a^m$ yielding a word which does not belong to the language $L$. □

**Lemma 32** *The language $L = \{a,b\}^* \cup \{c\}\{ab\}^*\{c\}$ belongs to the set $\mathcal{EC}(ORD) \setminus \mathcal{EC}(SYDEF)$.*

*Proof.* In Example 1, we have given a contextual grammar where all selection languages are accepted by ordered finite automata, and thus, have shown that $L \in \mathcal{EC}(ORD)$.

Suppose that the language $L$ is also generated by a contextual grammar $G'$ where all selection languages are symmetric definite.

Let us consider a word $w = c(ab)^n c \in L$ for some $n \geq \ell(G')$. Due to the choice of $n$, the word $w$ is derived in one step from some word $z$ by using a selection language $S$ and context $(u,v)$: $z \Longrightarrow uzv = w$. The word $u$ begins with the letter $c$; the word $v$ ends with $c$. Due to the choice of $n$, we also have $|z|_a > 0$ and $|z|_b > 0$. Since $S$ is symmetric definite over the alphabet $V = \{a,b\}$, it can be expressed as $S = EV^*H$ for some regular languages $E$ and $H$ over $V$. The sets $E$ and $H$ are not empty because $S$ contains at least the word $z$. Let $e$ be a word of $E$ and $h$ a word of $H$. Then, the word $ebbh$ belongs to the selection language $S$ as well. Since $ebbh \in \{a,b\}^*$ and $\{a,b\}^* \subseteq L$, we can apply the same derivation to this word and obtain $uebbhv$. This word starts and ends with $c$ but it does not have the form of those words from $L$ because of the double $b$. From this contradiction, it follows $L \notin \mathcal{EC}(SYDEF)$. □

**Lemma 33** *The language $L = \{a,b\}^* \cup \{c\}\{\lambda,b\}\{ab\}^*\{c\}$ belongs to $\mathcal{EC}(SUF) \setminus \mathcal{EC}(SYDEF)$.*

*Proof.* The language $L$ is generated by the contextual grammar $G = (\{a,b,c\}, \{(S_1,C_1),(S_2,C_2)\}, \{\lambda\})$ with

$$S_1 = \{a,b\}^*, \quad C_1 = \{(\lambda,a),(\lambda,b)\} \quad \text{and} \quad S_2 = Suf(\{ab\}^*), \quad C_2 = \{(c,c)\}$$

where $Suf(M)$ denotes the suffix-closure of the set $M$.

With the same argumentation as in the proof of Lemma 32, one can show also here $L \notin \mathcal{EC}(SYDEF)$ (the letters $c$ are in both cases wrapped around words which are an alternating sequence of $a$ and $b$ what cannot be checked by a symmetric definite selection language). □

**Lemma 34** *Let $L_1 = \{ a^n \mid n \geq 1 \}$, $L_2 = \{ ba^n b \mid n \geq 1 \}$, $L_3 = \{ cba^{2n}bc \mid n \geq 1 \}$, and $L = L_1 \cup L_2 \cup L_3$. Then, it holds $L \in \mathcal{EC}(SYDEF) \setminus \mathcal{EC}(NC)$.*

*Proof.* Let $V = \{a,b,c\}$. The contextual grammar $G = (V, \{(S_1,C_1),(S_2,C_2)\}, \{a\})$ with

$$S_1 = \{a\}V^*\{\lambda\}, \quad C_1 = \{(\lambda,a),(b,b)\} \quad \text{and} \quad S_2 = \{ ba^{2m}b \mid m \geq 1 \}V^*\{\lambda\}, \quad C_2 = \{(c,c)\}$$

generates the language $L$. This can be seen as follows: The shortest word of $L$ is $a$ which is the axiom. To every word of $L$ starting with the letter $a$ (hence, any word of $L_1$), another $a$ can be added or the letter $b$ is added at the beginning and the end of the word (using the first selection component) yielding all and only words of the languages $L_1$ and $L_2$. To every word of $L_2$ which also belongs to $S_2$, the letter $c$ is added at the beginning and the end of the word (using the second selection component) yielding exactly the words of the language $L_3$. To the words of $L_3$, no selection component can be applied. All the selection languages are symmetric definite as can be seen from the form in which they are given.

In [28], it was proved that the language $L$ does not belong to the family $\mathcal{EC}(NC)$. □

Next, we show some equalities.

**Lemma 35** *A restriction to comet languages (left, right, two-sided) as selection languages does not decrease the generative capacity of external contextual grammars:*

$$\mathcal{EC}(REG) = \mathcal{EC}(LCOM) = \mathcal{EC}(RCOM) = \mathcal{EC}(2COM).$$

*Proof.* With the inclusions $LCOM \subseteq 2COM$, $RCOM \subseteq 2COM$, and $2COM \subseteq REG$ (see Theorem 25 and Figure 1), we obtain also the inclusions $\mathcal{EC}(LCOM) \subseteq \mathcal{EC}(2COM)$, $\mathcal{EC}(RCOM) \subseteq \mathcal{EC}(2COM)$, and $\mathcal{EC}(2COM) \subseteq \mathcal{EC}(REG)$ according to Lemma 26.

Let $G = (V, \{(S_1,C_1), \dots, (S_n,C_n)\}, A)$ be a contextual grammar with arbitrary regular selection languages. Further, let $X$ be a new symbol ($X \notin V$). We set $S'_i = \{X\}^* S_i$ for $1 \leq i \leq n$. Then, the contextual grammar $G' = (V \cup \{X\}, \{(S'_1,C_1), \dots, (S'_n,C_n)\}, A)$ generates the same language as $G$. The selection languages are all right-sided comet languages. The letter $X$ neither occurs in an axiom nor in a context. Therefore, the part $\{X\}^*$ of the selection languages has no impact on the possible derivations (the only word used is $\lambda$). Thus, the inclusion $\mathcal{EC}(REG) \subseteq \mathcal{EC}(RCOM)$ holds.

With $S'_i = S_i\{X\}^*$ for $1 \leq i \leq n$, the same language is generated and the selection languages are left-sided comets. Hence, we also have the inclusion $\mathcal{EC}(REG) \subseteq \mathcal{EC}(LCOM)$. Hence, we obtain the chain of inclusions $\mathcal{EC}(REG) \subseteq \mathcal{C} \subseteq 2COM \subseteq \mathcal{EC}(REG)$ for $\mathcal{C} \in \{LCOM, RCOM\}$ which implies the equalities stated in the lemma. □

We now prove some proper inclusions.

**Lemma 36** *The family $\mathcal{EC}(MON)$ is a proper subset of the family $\mathcal{EC}(STAR)$.*

*Proof.* With the inclusion $MON \subseteq STAR$ (see Theorem 25 and Figure 1), we obtain also the inclusion $\mathcal{EC}(MON) \subseteq \mathcal{EC}(STAR)$ according to Lemma 26.

The language $L = \{\, a^n \mid n \geq 2 \,\} \cup \{\, ba^{2n}b \mid n \geq 1 \,\}$ from Lemma 30 belongs to the family $\mathcal{EC}(STAR)$ but not to the family $\mathcal{EC}(PS)$ and, hence, neither to $\mathcal{EC}(MON)$. Thus, the language is a witness for the properness of the inclusion. □

**Lemma 37** *The family $\mathcal{EC}(FIN)$ is a proper subset of the family $\mathcal{EC}(STAR)$*

*Proof.* According to [6], $\mathcal{EC}(FIN) \subset \mathcal{EC}(MON)$. According to Lemma 36, $\mathcal{EC}(MON) \subset \mathcal{EC}(STAR)$. Hence, the family $\mathcal{EC}(FIN)$ is also a proper subset of the family $\mathcal{EC}(STAR)$. □

**Lemma 38** *The family $\mathcal{EC}(STAR)$ is a proper subset of the families $\mathcal{EC}(LCOM)$ and $\mathcal{EC}(RCOM)$.*

*Proof.* The inclusions $STAR \setminus \{\{\lambda\}\} \subseteq LCOM$ and $STAR \setminus \{\{\lambda\}\} \subseteq RCOM$ hold as recalled in Section 3.2. Consider an external contextual grammar with a single selection component $(\{\lambda\}, C)$ (if there are more components with the selection language $\{\lambda\}$, they can be joined to one where the new set of contexts is the union of the single sets and the selection language is still the same). If the generated language contains the empty word, then this is an axiom since it cannot be obtained by derivation.

Then, exactly the (finitely many) words $uv$ with $(u,v) \in C$ are generated using this selection component. Thus, if we put all these words $uv$ with $(u,v) \in C$ into the set of axioms as well, we can remove the component $(\{\lambda\},C)$ and obtain a contextual grammar which generates the same language but has no selection language $\{\lambda\}$ anymore. Then, the remaining selection languages belong to the families $LCOM$ and $RCOM$. Hence, every language of $\mathcal{EC}(STAR)$ also belongs to the families $\mathcal{EC}(LCOM)$ and $\mathcal{EC}(RCOM)$.

According to Lemma 28, the language $L = \{ b^n a \mid n \geq 0 \} \cup \{\lambda\}$ belongs to $\mathcal{EC}(COMB)$ (and also to $\mathcal{EC}(LCOM)$ and $\mathcal{EC}(RCOM)$ by Theorem 25, Figure 1, and Lemma 26) but not to $\mathcal{EC}(STAR)$. This proves the properness of the inclusion. □

**Lemma 39** *The family $\mathcal{EC}(DEF)$ is a proper subset of the family $\mathcal{EC}(SYDEF)$.*

*Proof.* Let $G = (V,\{(S_1,C_1),\ldots,(S_n,C_n)\},A)$ be a contextual grammar where all selection languages are definite: $S_i = U_i^* B_i \cup A_i$ for $1 \leq i \leq n$. We first separate the finite parts and obtain the contextual grammar $G' = (V,\{(U_1^* B_1,C_1),(A_1,C_1),\ldots,(U_n^* B_n,C_n),(A_n,C_n)\},A)$ which generates the same language as $G$. Next, we eliminate the components with finite selection languages: If a set $B_i$ is empty, then the entire selection language is empty and cannot be used for derivation. Hence, we can simply omit such selection components without changing the generated language. For every component $(A_i,C_i)$ where $A_i$ is a finite language ($1 \leq i \leq n$), we move all words $uwv$ with $(u,v) \in C_i$ and $w \in A_i \cap L(G)$ into the set of axioms. These are finitely many (as $A_i$ and $C_i$ are finite) and are exactly the words generated by these components). Hence, we can remove these components afterwards. Then, we have obtained a contextual grammar which still generates the same language $L(G)$ but has only symmetric definite languages left.

The language $L = \{ a^n \mid n \geq 1 \} \cup \{ ba^n b \mid n \geq 1 \} \cup \{ cba^{2n}bc \mid n \geq 1 \}$ is a witness language for the properness of the inclusion which, according to Lemma 34, belongs to the family $\mathcal{EC}(SYDEF)$ but not to the family $\mathcal{EC}(NC)$ and, hence, not to (since $\mathcal{EC}(DEF) \subset \mathcal{EC}(NC)$ according to [6]). □

**Lemma 40** *The family $\mathcal{EC}(SYDEF)$ is a proper subset of the family $\mathcal{EC}(PS)$.*

*Proof.* From [19], we know the inclusion $SYDEF \subseteq PS$. Therefore, by Lemma 26, we have the inclusion $\mathcal{EC}(SYDEF) \subseteq \mathcal{EC}(PS)$. Its properness follows from Lemma 32 with $L = \{a,b\}^* \cup \{c\}\{ab\}^*\{c\}$ which belongs to the family $\mathcal{EC}(ORD)$ (and also to $\mathcal{EC}(PS)$ by [28]) but not to the family $\mathcal{EC}(SYDEF)$. □

Now, we prove the incomparability relations mentioned in Figure 2 which have not been proved earlier. These are the relations regarding the families $\mathcal{EC}(STAR)$ and $\mathcal{EC}(SYDEF)$ since the families $\mathcal{EC}(LCOM)$, $\mathcal{EC}(RCOM)$, and $\mathcal{EC}(2COM)$ coincide with $\mathcal{EC}(REG)$ and are therefore not incomparable to the other families mentioned.

**Lemma 41** *Let $\mathcal{F} = \{COMB, DEF, SYDEF, ORD, NC, PS\}$. The family $\mathcal{EC}(STAR)$ is incomparable to each family $\mathcal{EC}(F)$ with $F \in \mathcal{F}$.*

*Proof.* Due to the inclusion relations, it suffices to show that there are two languages $L_1$ and $L_2$ with the properties $L_1 \in \mathcal{EC}(COMB) \setminus \mathcal{EC}(STAR)$ and $L_2 \in \mathcal{EC}(STAR) \setminus \mathcal{EC}(PS)$. From Lemma 28, we have the language $L_1 = \{ b^n a \mid n \geq 0 \} \cup \{\lambda\}$. From Lemma 30, we have $L_2 = \{ ba^{2n}b \mid n \geq 1 \} \cup \{ a^n \mid n \geq 2 \}$. □

**Lemma 42** *Let $\mathcal{F} = \{NIL, COMM, CIRC\}$. The family $\mathcal{EC}(STAR)$ is incomparable to each family $\mathcal{EC}(F)$ with $F \in \mathcal{F}$.*

*Proof.* Due to the inclusion relations, it suffices to show that there are two languages $L_1$ and $L_2$ with the properties $L_1 \in \mathcal{EC}(NIL) \setminus \mathcal{EC}(STAR)$ and $L_2 \in \mathcal{EC}(STAR) \setminus \mathcal{EC}(CIRC)$. From Lemma 27, we have the language $L_1 = \{\, a^n bbb \mid n \geq 1 \,\} \cup \{\lambda\}$. From Lemma 31, we have $L_2 = \{\, a^n b^n \mid n \geq 1 \,\} \cup \{\, b^n a^n \mid n \geq 1 \,\}$. □

**Lemma 43** *The language family $\mathcal{EC}(STAR)$ is incomparable to the family $\mathcal{EC}(SUF)$.*

*Proof.* We have $L_1 = \{a,b\}^*\{\, a^n b^m \mid n \geq 1,\ m \geq 1 \,\} \cup \{\, ca^n b^m c \mid n \geq 1,\ m \geq 1 \,\} \in \mathcal{EC}(SUF) \setminus \mathcal{EC}(STAR)$ from Lemma 29. From Lemma 30, we know that $L_2 = \{\, a^n \mid n \geq 2 \,\} \cup \{\, ba^{2n}b \mid n \geq 1 \,\}$ belongs to the family $\mathcal{EC}(STAR)$ but not to $\mathcal{EC}(PS)$ (and neither to $\mathcal{EC}(SUF)$ by [28]). □

**Lemma 44** *The language family $\mathcal{EC}(SYDEF)$ is incomparable to the family $\mathcal{EC}(SUF)$.*

*Proof.* We have $L_1 = \{a,b\}^* \cup \{c\}\{\lambda,b\}\{ab\}^*\{c\} \in \mathcal{EC}(SUF) \setminus \mathcal{EC}(SYDEF)$ from Lemma 33. From [6], we know that $L_2 = \{\, ab^n \mid n \geq 1 \,\} \cup \{\lambda\}$ belongs to the family $\mathcal{EC}(COMB)$ but not to $\mathcal{EC}(SUF)$. By [28] and Lemma 39, the language $L_2$ also belongs to $\mathcal{EC}(SYDEF)$. □

**Lemma 45** *The family $\mathcal{EC}(SYDEF)$ is incomparable to each of the families $\mathcal{EC}(ORD)$ and $\mathcal{EC}(NC)$.*

*Proof.* Due to the inclusion relations, it suffices to show that there are two languages $L_1$ and $L_2$ with the properties $L_1 \in \mathcal{EC}(ORD) \setminus \mathcal{EC}(SYDEF)$ and $L_2 \in \mathcal{EC}(SYDEF) \setminus \mathcal{EC}(NC)$. From Lemma 32, we have $L_1 = \{a,b\}^* \cup \{c\}\{ab\}^*\{c\}$. As $L_2$, we take $L_2 = \{\, a^n \mid n \geq 1 \,\} \cup \{\, ba^n b \mid n \geq 1 \,\} \cup \{\, cba^{2n}bc \mid n \geq 1 \,\}$ from Lemma 34. □

**Lemma 46** *The family $\mathcal{EC}(SYDEF)$ is incomparable to each of the families $\mathcal{EC}(COMM)$ and $\mathcal{EC}(CIRC)$.*

*Proof.* Due to the inclusion relations, it suffices to show that there are two languages $L_1$ and $L_2$ with the properties $L_1 \in \mathcal{EC}(COMM) \setminus \mathcal{EC}(SYDEF)$ and $L_2 \in \mathcal{EC}(SYDEF) \setminus \mathcal{EC}(CIRC)$. In [28], it was proved that the language $L_1 = \{\, a^n \mid n \geq 2 \,\} \cup \{\, ba^{2n}b \mid n \geq 1 \,\}$ belongs to $\mathcal{EC}(COMM)$ but not to $\mathcal{EC}(PS)$ (this can be seen also in the proof of Lemma 30). By Lemma 40, the language $L_1$ neither belongs to the family $\mathcal{EC}(SYDEF)$.

In [6], it was proved that the language $L_2 = \{\, abc^n \mid n \geq 1 \,\} \cup \{\, c^n ab \mid n \geq 1 \,\}$ belongs $\mathcal{EC}(COMB)$ but not to $\mathcal{EC}(CIRC)$. By [28] and Lemma 39, the language $L_2$ also belongs to the family $\mathcal{EC}(SYDEF)$. □

**Theorem 47 (Hierarchy of the $\mathcal{EC}$ language families)** *The inclusion relations presented in Figure 2 hold. An arrow from an entry $X$ to an entry $Y$ depicts the proper inclusion $X \subset Y$; if two families are not connected by a directed path, then they are incomparable.*

*Proof.* An edge label refers to the paper or lemma in the present paper where the proper inclusion is shown. The incomparability results are proved in Lemmas 41 to 46. □

## 5 Conclusion and future work

In this paper, we have extended the previous hierarchy of subregular language families and families generated by external contextual grammars with selection in certain subregular language families.

Various other subregular language families have also been investigated in the past (for instance, in [1, 12, 19]). Future research will be on extending and unifying current hierarchies of subregular language families (presented, for instance, in [9, 28]) by additional families and to use them as control in external contextual grammars. We already started investigations on the position of prefix- and infix-closed as well as prefix-, suffix-, and infix-free languages in the current hierarchy and their impact on the generative power of external contextual grammars when used for selection. The extension of the hierarchy with other families of definite-like languages (for instance, ultimate definite, central definite, noninital definite) has also already begun.

The research will be also extended to internal contextual grammars or tree-controlled grammars where results are already available in [9, 28, 29, 30].

## References

[1] Henning Bordihn, Markus Holzer & Martin Kutrib (2009): *Determination of finite automata accepting subregular languages*. Theoretical Computer Science 410(35), pp. 3209–3222, doi:`10.1016/j.tcs.2009.05.019`.

[2] Janusz A. Brzozowski (1962): *Regular expression techniques for sequential circuits*. Ph.D. thesis, Princeton University, Princeton, NJ, USA.

[3] Janusz A. Brzozowski (1967): *Roots of star events*. Journal of the ACM 14(3), pp. 466–477, doi:`10.1109/SWAT.1966.21`.

[4] Janusz A. Brzozowski & Rina Cohen (1969): *On decompositions of regular events*. Journal of the ACM 16(1), pp. 132–144, doi:`10.1145/321495.321505`.

[5] Janusz A. Brzozowski, Galina Jirásková & Chenglong Zou (2014): *Quotient complexity of closed languages*. Theory of Computing Systems 54, pp. 277–292, doi:`10.1007/s00224-013-9515-7`.

[6] Jürgen Dassow (2005): *Contextual grammars with subregular choice*. Fundamenta Informaticae 64(1–4), pp. 109–118.

[7] Jürgen Dassow (2015): *Contextual languages with strictly locally testable and star free selection languages*. Analele Universitatii Bucuresti 62, pp. 25–36.

[8] Jürgen Dassow, Florin Manea & Bianca Truthe (2012): *On external contextual grammars with subregular selection languages*. Theoretical Computer Science 449, pp. 64–73, doi:`10.1016/j.tcs.2012.04.008`.

[9] Jürgen Dassow & Bianca Truthe (2023): *Relations of contextual grammars with strictly locally testable selection languages*. RAIRO – Theoretical Informatics and Applications 57, p. #10, doi:`10.1051/ita/2023012`.

[10] Ference Gécseg & István Peák (1972): *Algebraic Theory of Automata*. Academiai Kiado, Budapest.

[11] Arthur Gill & Lawrence T. Kou (1974): *Multiple-entry finite automata*. Journal of Computer and System Sciences 9(1), pp. 1–19, doi:`10.1016/S0022-0000(74)80034-6`.

[12] Yo-Sub Han & Kai Salomaa (2009): *State complexity of basic operations on suffix-free regular languages*. Theoretical Computer Science 410(27), pp. 2537–2548, doi:`10.1016/j.tcs.2008.12.054`.

[13] Ivan M. Havel (1969): *The theory of regular events II*. Kybernetika 5(6), pp. 520–544.

[14] Markus Holzer & Bianca Truthe (2015): *On relations between some subregular language families*. In Rudolf Freund, Markus Holzer, Nelma Moreira & Rogério Reis, editors: Seventh Workshop on Non-Classical Mod-

*els of Automata and Applications – NCMA 2015, Porto, Portugal, August 31 – September 1, 2015. Proceedings*, *books@ocg.at* 318, Österreichische Computer Gesellschaft, pp. 109–124.

[15] Manfred Kudlek (2004): *On languages of cyclic words*. In Natasha Jonoska, Gheorghe Păun & Grzegorz Rozenberg, editors: *Aspects of Molecular Computing, Essays Dedicated to Tom Head on the Occasion of His 70th Birthday*, *LNCS* 2950, Springer-Verlag, pp. 278–288, doi:10.1007/978-3-540-24635-0_20.

[16] Solomon Marcus (1969): *Contextual grammars*. *Revue Roumaine de Mathématique Pures et Appliquées* 14, pp. 1525–1534.

[17] Robert McNaughton & Seymour Papert (1971): *Counter-Free Automata*. MIT Press, Cambridge, USA.

[18] Benedek Nagy (2019): *Union-Freeness, Deterministic Union-Freeness and Union-Complexity*. In Michal Hospodár, Galina Jirásková & Stavros Konstantinidis, editors: *Descriptional Complexity of Formal Systems, 21st IFIP WG 1.02 International Conference, DCFS 2019, Košice, Slovakia, July 17–19, 2019, Proceedings*, Springer, Cham, pp. 46–56, doi:10.1007/978-3-030-23247-4_3.

[19] Viktor Olejár & Alexander Szabari (2023): *Closure Properties of Subregular Languages Under Operations*. *International Journal of Foundations of Computer Science*, pp. 1–25, doi:10.1142/S0129054123450016.

[20] Azaria Paz & Bezalel Peleg (1965): *Ultimate-definite and symmetric-definite events and automata*. *Journal of the ACM* 12(3), pp. 399–410, doi:10.1145/321281.321292.

[21] Micha A. Perles, Michael O. Rabin & Eli Shamir (1963): *The theory of definite automata*. *IEEE Transactions of Electronic Computers* 12, pp. 233–243, doi:10.1109/PGEC.1963.263534.

[22] Grzegorz Rozenberg & Arto Salomaa, editors (1997): *Handbook of Formal Languages*. Springer-Verlag, Berlin, doi:10.1007/978-3-642-59136-5.

[23] Huei-Jan Shyr (1991): *Free Monoids and Languages*. Hon Min Book Co., Taichung, Taiwan.

[24] Huei-Jan Shyr & Gabriel Thierrin (1974): *Ordered automata and associated languages*. *Tamkang Journal of Mathematics* 5(1), pp. 9–20.

[25] Huei-Jan Shyr & Gabriel Thierrin (1974): *Power-separating regular languages*. *Mathematical Systems Theory* 8(1), pp. 90–95, doi:10.1007/BF01761710.

[26] Bianca Truthe (2014): *A relation between definite and ordered finite automata*. In Suna Bensch, Rudolf Freund & Friedrich Otto, editors: *Sixth Workshop on Non-Classical Models for Automata and Applications – NCMA 2014, Kassel, Germany, July 28–29, 2014. Proceedings*, *books@ocg.at* 304, Österreichische Computer Gesellschaft, pp. 235–247.

[27] Bianca Truthe (2018): *Hierarchy of Subregular Language Families*. Technical Report, Justus-Liebig-Universität Giessen, Institut für Informatik, IFIG Research Report 1801.

[28] Bianca Truthe (2021): *Generative Capacity of Contextual Grammars with Subregular Selection Languages*. *Fundamenta Informaticae* 180(1–2), pp. 123–150, doi:10.3233/FI-2021-2037.

[29] Bianca Truthe (2023): *Merging two Hierarchies of Internal Contextual Grammars with Subregular Selection*. In Benedek Nagy & Rudolf Freund, editors: *Proceedings of the 13th International Workshop on Non-Classical Models of Automata and Applications, NCMA 2023, Famagusta, North Cyprus, 18th–19th September, 2023*, *EPTCS* 388, pp. 125–139, doi:10.4204/EPTCS.388.12.

[30] Bianca Truthe (2023): *Strictly Locally Testable and Resources Restricted Control Languages in Tree-Controlled Grammars*. In Zsolt Gazdag, Szabolcs Iván & Gergely Kovásznai, editors: *Proceedings of the 16th International Conference on Automata and Formal Languages, AFL 2023, Eger, Hungary, September 5–7, 2023*, *EPTCS* 386, pp. 253–268, doi:10.4204/EPTCS.386.20.

[31] Barbara Wiedemann (1978): *Vergleich der Leistungsfähigkeit endlicher determinierter Automaten*. Diplomarbeit, Universität Rostock.

# Complexity of Unary Exclusive Nondeterministic Finite Automata

Martin Kutrib Andreas Malcher
Matthias Wendlandt

Institut für Informatik, Universität Giessen
Arndtstr. 2, 35392 Giessen, Germany

{kutrib, andreas.malcher, matthias.wendlandt}@informatik.uni-giessen.de

Exclusive nondeterministic finite automata (XNFA) are nondeterministic finite automata with a special acceptance condition. An input is accepted if there is exactly one accepting path in its computation tree. If there are none or more than one accepting paths, the input is rejected. We study the descriptional complexity of XNFA accepting unary languages. While the state costs for mutual simulations with DFA and NFA over general alphabets differ significantly from the known types of finite automata, it turns out that the state costs for the simulations in the unary case are in the order of magnitude of the general case. In particular, the state costs for the simulation of an XNFA by a DFA or an NFA are $e^{\Theta(\sqrt{n \cdot \ln n})}$. Conversely, converting an NFA to an equivalent XNFA may cost $e^{\Theta(\sqrt{n \cdot \ln n})}$ states as well. All bounds obtained are also tight in the order of magnitude. Finally, we investigate the computational complexity of different decision problems for unary XNFAs and it is shown that the problems of emptiness, universality, inclusion, and equivalence are coNP-complete, whereas the general membership problem is NL-complete.

## 1 Introduction

The ability of using nondeterminism for finite automata does not increase their computational power in comparison with the deterministic variant, but the simulation costs for a deterministic finite automaton (DFA) can be exponentially higher in terms of states than for an equivalent nondeterministic finite automaton (NFA) [21, 23].

In the last decades several structural extensions of finite automata have been examined. One such extension is, for example, to give the reading head of the finite automaton the power of two-way motion. Such *two-way* finite automata do also not increase the computational power of finite automata [30], but they are interesting from a descriptional complexity point of view, since the costs for one-way deterministic finite automata for the simulation of two-way deterministic finite automata can be exponential in the number of states [23]. Similar results can also be shown for the nondeterministic case [31].

A more fine-grained look on the range between nondeterministic and deterministic finite automata leads to the model of *unambiguous* finite automata [32]. Here, nondeterminism is allowed, but for every accepted word there has to be exactly one accepting path. From a descriptional complexity perspective it is known that the trade-off from unambiguous finite automata to DFAs is exponential as well [17, 18, 32].

In contrast to these structural extensions, another extension is examined in [13, 14] that is based on the acceptance conditions of the automata and which leads to *exclusive* nondeterministic finite automata (XNFA). In this model, the computation tree of an input is defined in the same way as for nondeterministic finite automata, but its interpretation is different. Namely, an input word $w$ is accepted, if there is exactly one accepting path for $w$. If there is no accepting path for $w$ or two or more accepting paths for $w$, then $w$ is rejected. Clearly, any unambiguous finite automaton can be considered as an XNFA, but in comparison to unambiguous finite automata, multiple accepting paths are allowed and lead to non-acceptance in an XNFA. In [13, 14] complexity aspects of XNFAs have been investigated. Concerning the descriptional complexity, it is shown that $n$-state XNFAs can be determinized as well, but the upper bound turns out to be $3^n - 2^n + 1$ and is shown to be tight. Moreover, $n \cdot 2^{n-1}$ states are shown to be a tight bound for the simulation of an XNFA by an equivalent NFA. The simulation of an NFA by an equivalent XNFA leads to an upper bound of $2^n - 1$ which is shown to be tight as well. Concerning the computational complexity, it is shown that the problems of emptiness, universality, inclusion, and equivalence are PSPACE-complete, whereas the general membership problem is NL-complete. It should be noted that a computational model with exactly one accepting computation on every accepted input has already been known in the context of complexity theory as the class US (unique solution). It is defined (see [1]) as the class of languages $L$ for which there exists a nondeterministic polynomial time Turing machine $M$ such that $w \in L$ if and only if $M$ has on input $w$ exactly one accepting computation path. A short overview on the properties of the class US may be found in [8].

In this paper, we investigate the descriptional and computational complexity of XNFAs accepting *unary* languages. The descriptional complexity of unary regular languages has extensively been studied in the literature. A fundamental result was obtained by Chrobak in [2, 3]. He shows that $O(F(n))$ is a tight bound for the simulation of an NFA by an equivalent DFA. Here, $F(n)$ denotes Landau's function [15] that is the maximal order of the cyclic subgroups of the symmetric group on $n$ elements and can be estimated as $F(n) \in e^{\Theta(\sqrt{n \cdot \ln n})}$. Landau's function plays a crucial role in many results on the descriptional complexity of unary regular languages. One line of research in the past years is that many automata models such as, for example, one-way finite automata, two-way finite automata, pushdown automata, and context-free grammars have been investigated and compared to each other with respect to simulation results and the size costs of the simulation (see, for example, [6, 20, 25, 26, 29]). Another line of research in recent years concerns investigations on the state complexity of operations on unary languages which can be found, for example, in [9, 12, 19, 28].

The paper is structured as follows. In Section 2, we give the basic definitions that are used in the further sections. In Section 3, we study the descriptional costs for determinizing a given unary XNFA. As a fundamental preparatory step we show that any unary $n$-state XNFA can be converted to an equivalent $O(n^3)$-state XNFA in Chrobak normal form. This result is in slight contrast to NFAs where the conversion of an arbitrary NFA to Chrobak normal form may induce only a quadratic blow-up of the number of states. Based on the XNFA in Chrobak normal form we can construct an equivalent DFA whose number of states is bounded by $e^{\Theta(\sqrt{n \cdot \ln n})}$. This upper bound is also tight in the order of magnitude. In Section 4, we obtain similar upper and lower bounds for the conversion of unary XNFAs to equivalent NFAs and of unary NFAs to equivalent XNFAs. Finally, in Section 5 we study the computational complexity of decidability questions. In particular, we consider general membership, emptiness, universality, inclusion, and equivalence with respect to the unary case and show that for unary XNFAs the general membership problem is NL-complete, whereas the questions of emptiness, finiteness, inclusion, and equivalence are coNP-complete.

## 2 Definitions and Preliminaries

Let $\Sigma^*$ denote the set of all words over the finite alphabet $\Sigma$. The *empty word* is denoted by $\lambda$, and $\Sigma^+ = \Sigma^* \setminus \{\lambda\}$. The *reversal* of a word $w$ is denoted by $w^R$. For the *length* of $w$ we write $|w|$. We use $\subseteq$ for *inclusions* and $\subset$ for *strict inclusions*. We write $2^S$ for the power set and $|S|$ for the cardinality of a set $S$.

A *nondeterministic finite automaton* (NFA) is a system $M = \langle Q, \Sigma, \delta, q_0, F \rangle$, where $Q$ is the finite set of *states*, $\Sigma$ is the finite set of *input symbols*, $q_0 \in Q$ is the *initial state*, $F \subseteq Q$ is the set of *accepting states*, and $\delta \colon Q \times \Sigma \to 2^Q$ is the *transition function*.

With an eye towards further modes of acceptance, we define the *acceptance of an input* in terms of computation trees. For any input $w = a_1 a_2 \cdots a_n \in \Sigma^*$ read by some NFA $M$, a *(complete) path for $w$* is a sequence of states $q_0, q_1, \ldots, q_n$ such that $q_{i+1} \in \delta(q_i, a_{i+1})$, $0 \leq i \leq n-1$. All possible paths on $w$ are combined into a *computation tree* of $M$ on $w$. So, a computation tree of $M$ is a finite rooted tree whose nodes are labeled with states of $M$. In particular, the root is labeled with the initial state, and the successor nodes of a node labeled $q$ are the nodes $p_1, p_2, \ldots, p_m$ if and only if $\delta(q,a) = \{p_1, p_2, \ldots, p_m\}$, for the current input symbol $a$. A path in the computation tree is an *accepting path* if it ends in an accepting state.

Now, an input $w$ is accepted by an NFA if at least one path in the computation tree of $w$ is accepting.

An NFA, where for acceptance it is required that *exactly* one path is accepting, is called an *exclusive nondeterministic finite automaton* (XNFA).

The *language accepted* by the XNFA $M$ is $L(M) = \{\, w \in \Sigma^* \mid w \text{ is accepted by } M \,\}$.

Finally, an NFA is a *deterministic finite automaton* (DFA) if and only if $|\delta(q,a)| = 1$, for all $q \in Q$ and $a \in \Sigma$. In this case we simply write $\delta(q,a) = p$ for $\delta(q,a) = \{p\}$ assuming that the transition function is a mapping $\delta \colon Q \times \Sigma \to Q$. So, any DFA is complete, that is, the transition function is total, whereas for the other automata types it is possible that $\delta$ maps to the empty set. A finite automaton is called *unary* if its set of input symbols is a singleton. In this case we use $\Sigma = \{a\}$ throughout the paper.

## 3 Determinization of unary XNFAs

The problem of evaluating the costs of unary automata simulations was raised in [34], and has led to emphasize some relevant differences with the general case. For example, unary NFAs can be much more concise than DFAs, but yet not as much as for the general case. Moreover, the sophisticated studies in [20] reveal tight bounds for many other types of unary finite automata conversions. The paper and the survey [27] are also a valuable source for further references.

For state complexity issues of unary finite automata, Landau's function

$$F(n) = \max\{\, \mathrm{lcm}(c_1, c_2 \ldots, c_l) \mid l \geq 1, c_1, c_2, \ldots, c_l \geq 1, c_1 + c_2 + \cdots + c_l = n \,\}$$

which gives the maximal order of the cyclic subgroups of the symmetric group on $n$ elements, plays a crucial role, where lcm denotes the *least common multiple* [15, 16]. It is well known that the $c_i$ always can be chosen to be relatively prime. Moreover, an easy consequence of the definition is that the $c_i$ always can be chosen such that $c_1, c_2, \ldots, c_l \geq 2$, $c_1 + c_2 + \cdots + c_l \leq n$, and $\mathrm{lcm}(c_1, c_2, \ldots, c_l) = F(n)$ (cf., for example, [24]).

Since $F$ depends on the irregular distribution of the prime numbers we cannot expect to express $F(n)$ explicitly by $n$. In [15, 16] the asymptotic growth rate $\lim_{n\to\infty}(\ln F(n)/\sqrt{n \cdot \ln n}) = 1$ was determined, which for our purposes implies the (sufficient) rough estimate $F(n) \in e^{\Theta(\sqrt{n \cdot \ln n})}$ (see also [4, 36] for bounds on $F$).

The asymptotically tight bound of $F(n)$ for the unary NFA-to-DFA conversion was presented in [2, 3]. The proof is based on a normal form for unary NFAs derived in [2]. Each $n$-state unary NFA can effectively be converted into an equivalent $O(n^2)$-state NFA in this so-called Chrobak normal form. However, the original proof in [2] contains an error that has been discovered and fixed in [37]. While the correction increases the state costs, their order of magnitude is not affected. In connection with magic numbers, more precise and improved state bounds have been shown in [5] by a completely different proof.

Let $t,d \geq 0$ be two integers. An *arithmetic progression with offset $t$ and period $d$* is the set

$$\{\, t + x \cdot d \mid x \geq 0 \,\}.$$

We recall a well-known useful fact which is related to number theory and Frobenius numbers (see, for example, [33] for a survey).

**Lemma 1.** *Let $0 < c_1 < c_2 < \cdots < c_r \leq n$ be positive integers. Then the set of integers $z > n^2$ that can be written as a non-negative integer linear combination of the $c_i$ is $\{\, t + x \cdot d \mid x \geq 0 \,\}$, where $t$ is the least integer greater than $n^2$ that is a multiple of $d = \gcd(c_1, c_2, \ldots, c_r)$.*

A unary XNFA $M = \langle Q, \{a\}, \delta, q_0, F \rangle$ is in *Chrobak normal form* if, for some $m \geq 0$ and $k \geq 0$, $Q = \{\, q_i \mid 0 \leq i \leq m \,\} \cup C_1 \cup C_2 \cup \cdots \cup C_k$, where, for each $1 \leq i \leq k$, $C_i = \{\, p_{i,0}, p_{i,1}, \ldots, p_{i,j_i-1} \,\}$ for some $j_i \geq 1$, $\delta(q_i, a) = \{q_{i+1}\}$ for $0 \leq i \leq m-1$, and for each $1 \leq i \leq k$ and $0 \leq h \leq j_i - 1$, $\delta(p_{i,h}, a) = \{p_{i,(h+1) \bmod j_i}\}$, and $\delta(q_m, a) = \{p_{1,0}, p_{2,0}, \ldots, p_{k,0}\}$.

So, an XNFA is in Chrobak normal form if its structure is a deterministic tail from $q_0$ to $q_m$, where the automaton makes only a single nondeterministic decision, which chooses one of the disjoint cycles $C_i$.

Next, we show how to convert a unary XNFA into Chrobak normal form. The idea of the construction is along the lines of the construction in [37] but with modifications with respect to the exclusiveness of the XNFA.

**Lemma 2.** *Let $n \geq 1$. For every unary $n$-state XNFA, an equivalent $O(n^3)$-state XNFA in Chrobak normal form can effectively be constructed, such that the sum of the cycle lengths is of order $O(n)$.*

*Proof.* Let $M = \langle Q, \Sigma, \delta, q_0, F \rangle$ be an $n$-state XNFA. Since any unary language over some alphabet is completely determined by the lengths of the words in the language, we can safely disregard $\Sigma$ and consider the state graph of $M$ only. For $L(M) = \emptyset$, the theorem is trivial. So, in the sequel we assume that $L(M)$ is not empty. Moreover, we may safely assume that all states $q \in Q$ are reachable and productive, that is, there is a path from $q_0$ to $q$ and a path from $q$ to a final state. Now, by adding states and possibly removing some states and transitions, we modify $M$ such that there is no incoming transition to the initial state, such that $F = \{q_+\}$ is a singleton, and such that $q_+$ is the only state without outgoing transitions. To this end, all unreachable states together with their incoming and outgoing transitions are removed. Similarly, all unproductive states together with their incoming and outgoing transitions are removed as well. Next, if the initial state has incoming transitions, a new state without incoming transitions is added whose outgoing transitions go to the successor states of the initial state. This new state becomes the new initial state. In order to make $F$ a singleton, we have to take care about words that are accepted on more than one path. So, first a new accepting state $q_+$ is added. For each pair of old accepting states, if both states do not share a common predecessor state, from each of their predecessor states a transition to $q_+$ is added. Both states become non-accepting. However, if both states have at least one common predecessor, say $p$, then there are two paths via $p$ to accepting states. This means that inputs following these paths do not belong to $L(M)$. In this case, both states become non-accepting, some state $p'$ is added, and all incoming transitions to $p$ are doubled and are directed to $p'$ as well. Furthermore, a transition from $p$ to $q_+$ and a transition from $p'$ to $q_+$ is added. Similarly, for all common predecessors of the old accepting states. In this way, we obtain an XNFA equivalent to $M$ that has the desired properties. For convenience, we call it also $M$. The modified XNFA has at most $m = 2n$ states.

From now on, we identify $M$ with its state graph. Let $S$ be the set of non-trivial strongly connected components of $M$. A *superpath* in $M$ is a subgraph

$$\alpha = P_1 S_1 P_2 S_2 \cdots P_\ell S_\ell P_{\ell+1},$$

where, for $1 \le i \le \ell$, $S_i \in S$; for $1 \le i \le \ell+1$, $P_i$ is a path in $M$ whose inner nodes do not belong to non-trivial strongly component components of $M$; the first node of $P_1$ is $q_0$; the last node of $P_{\ell+1}$ is $q_+$; for $1 \le i \le \ell$, the last node of $P_i$ belongs to $S_i$; for $2 \le i \le \ell+1$, the first node of $P_i$ belongs to $S_{i-1}$.

For every superpath $\alpha$ in $M$, let $L_\alpha$ be the set of all lengths of paths in $M$ from $q_0$ to $q_+$ that are in $\alpha$. It follows that the length of any accepting path in $M$ belongs to $\bigcup_\alpha L_\alpha$, where the union ranges over all superpaths in $M$.

We define the set $\Psi_\alpha$ to be the subset of paths from $q_0$ to $q_+$ in $\alpha$ that are simple, that is, no state appears twice. Clearly, the length of any path in $\Psi_\alpha$ does not exceed $m$.

Next, we define $\Pi_\alpha$ to be another subset of paths from $q_0$ to $q_+$ in $\alpha$. In particular, for every path $\sigma$ in $\Psi_\alpha$, we put the following extensions $\sigma'$ of $\sigma$ into $\Pi_\alpha$. Whenever $\sigma$ enters a strongly connected component $S_i$ in some state $v$, then a Hamiltonian walk in $S_i$ (that is, a tour that visits all nodes in $S_i$) that cannot be shortened and that starts and ends in $v$ is inserted into $\sigma$. Note that a Hamiltonian walk that cannot be shortened is a path from which no nodes can be removed without obtaining a path that is no longer Hamiltonian. It needs not to be the shortest Hamiltonian walk in $S_i$. Since $S_i$ is strongly connected, such Hamiltonian walks exist. Results in [7] show that the lengths of such Hamiltonian walks in $S_i$ do not exceed $|S_i|^2$, where $|S_i|$ denotes the number of nodes in $S_i$. Therefore, the length of any path in $\Pi_\alpha$ does not exceed $m^2+m$.

Now we consider a fixed superpath $\alpha$ in $M$. Let $0 < c_1 < c_2 < \cdots < c_r \le m$ be the lengths of all simple cycles in $\alpha$, $\sigma$ in $\Psi_\alpha$, and $\sigma'$ be an extension of $\sigma$ in $\Pi_\alpha$. Since $\sigma'$ visits each node in $\alpha$ at least once, the set $Z_{\alpha,\sigma'}$ of all lengths $z$ for which $z = |\sigma'| + x_1c_1 + x_2c_2 + \cdots + x_rc_r$ is solvable in non-negative integers is contained in $L_\alpha$. By Lemma 1, $Z_{\alpha,\sigma'} = X_\alpha \cup \{\, t_{\sigma'} + x \cdot d \mid x \ge 0 \,\}$, where $X_\alpha$ contains lengths not larger than $2m^2+m$ and $t_{\sigma'}$ is the least integer greater than $2m^2+m$ such that $t_{\sigma'} \equiv |\sigma'| \pmod{d}$, where $d = \gcd(c_1, c_2, \ldots, c_r)$. Since the Hamiltonian walks in $\sigma'$ are (compound) cycles, that is, linear combinations of $c_1, c_2, \ldots, c_r$, the number $d$ divides their lengths and, thus, we have $t_{\sigma'} \equiv |\sigma| \pmod{d}$.

On the other hand, the set of all lengths $y$ for which there is a $\sigma$ in $\Psi_\alpha$ such that

$$y = |\sigma| + x_1c_1 + x_2c_2 + \cdots + x_rc_r$$

is solvable in non-negative integers, clearly contains $L_\alpha$. Therefore, if $w \in L_\alpha$ and $w > 2m^2+m$ then Lemma 1 implies that there is a $\sigma$ in $\Psi_\alpha$ such that $w \equiv |\sigma| \pmod{d}$. Since $\{\, t_\sigma + x \cdot d \mid x \ge 0 \,\} \subseteq Z_{\alpha,\sigma'}$, we conclude $w \in Z_{\alpha,\sigma'}$.

Altogether, we have $L_\alpha = N_\alpha \cup \bigcup_{\sigma' \in \Pi_\alpha} \{\, t_{\sigma'} + x \cdot d \mid x \ge 0 \,\}$, where $N_\alpha$ contains lengths not larger than $2m^2+m$.

So far, we have created the prerequisites for constructing the normal form without specifically addressing XNFAs. So, the next task is to assemble an XNFA $M' = \langle Q', \{a\}, \delta', q'_0, F' \rangle$ equivalent to $M$ in Chrobak normal form.

To this end, we start with a deterministic tail consisting of the $m^3+2$ states $\{\, q'_i \mid 0 \le i \le m^3+1 \,\}$ with $\delta'(q'_i, a) = \{q'_{i+1}\}$, for $0 \le i \le m^3$. A state $q_i$ of the tail becomes accepting if and only if the input of length $i$ belongs to $L(M)$. So, all words whose length does not exceed $m^3+1$ are correctly accepted or rejected.

Next, we want to add the cycles to the initial tail of $M'$.

To construct the cycles appropriately, we consider each superpath $\alpha$ of $M$ and distinguish three cases, respectively. As before, let $0 < c_1 < c_2 < \cdots < c_r \leq m$ be the lengths of all simple cycles in $\alpha$ and $d = \gcd(c_1, c_2, \ldots, c_r)$. We consider all inputs of lengths $z > m^3 + 1 \in L_\alpha$.

Case 1: There are at least two simple cycles $C_1$ and $C_2$ in $\alpha$. Then, each path of length $z$ in $\alpha$ that can be shortened to some path in $\Pi_\alpha$ by deleting cycles, sees at least $z - (m^2 + m)$ nodes in complete simple cycles of $\alpha$. If one of these paths contains at least two different cycles of the same length, then these cycles can replace each other and, thus, there are at least two accepting paths of length $z$ in $\alpha$. Therefore, the input of length $z$ does not belong to $L(M)$. Assume now that all cycles in these paths have different lengths. Then there are at most $m$ cycles. Assume that each of these cycles is passed through at most $m-1$ times. Then,

$$z \leq m^2 + m + \sum_{i=1}^{m} i(m-1) = m^2 + m + \frac{m^2+m}{2}(m-1)$$

$$= \frac{m^2+m}{2}(m+1) = \frac{m^3+2m^2+m}{2} \leq m^3 + 1 < z.$$

From the contradiction we conclude that there is at least one cycle, say $C_1$, that is passed through for $x_1 \geq m$ times. Let $C_2$ be passed through for $x_2$ times. We have $x_1 \geq m \geq |C_2| \geq 1$ and $|C_1| \geq 1$. So, $x_1|C_1| + x_2|C_2| = (x_1 - |C_2|)|C_1| + (x_2 + |C_1|)|C_2|$. The equality means that passing $x_1$ times through the cycle $C_1$ and $x_2$ times through the cycle $C_2$ is equivalent to passing $(x_1 - |C_2|)$ times through the cycle $C_1$ and $(x_2 + |C_1|)$ times through the cycle $C_2$. So, there are at least two accepting paths of length $z$ in $\alpha$. Therefore, the input of length $z$ does not belong to $L(M)$.

Case 2: There is exactly one simple cycle $C_1$ in $\alpha$. So, there is at most one non-trivial strongly connected component in $\alpha$ and this strongly connected component is the cycle $C_1$. Clearly, in this case we have $d = |C_1|$ and the input length $z$ is uniquely accepted along $\alpha$.

Case 3: There is no simple cycle in $\alpha$. In this case, there is no non-trivial strongly connected component in $\alpha$ and the unique path of length $z$ from the initial state ends in the initial tail and, by construction, the input of length $z$ is correctly accepted or rejected.

Now we are ready to add the cycles for $\alpha$ to the tail of $M'$. To this end, nothing has to be done for Case 3.

For the remaining cases, the cycle length must be $d$. If there is no cycle of length $d$, we add two disjoint cycles $A_\alpha$ and $R_\alpha$ each of length $d$. In particular, $A_\alpha$ consists of states $\{s_0, s_1, \ldots, s_{d-1}\}$ with $\delta'(s_h, a) = \{s_{(h+1) \bmod d}\}$, and similarly, $R_\alpha$ consists of states $\{r_0, r_1, \ldots, r_{d-1}\}$ with $\delta'(r_h, a) = \{r_{(h+1) \bmod d}\}$. The cycles are connected to the tail by the transitions $\delta(q'_{m^3+1}, a) = \{s_0\}$ and $\delta(q'_{m^3+1}, a) = \{r_0\}$. If there are already two cycles $A$ and $R$ of length $d$ that have already been constructed for some other superpath, then they are reused and nothing is added.

Next, we identify the accepting states on the cycles.

For Case 1, we consider each $\sigma \in \Psi_\alpha$ and states $s_i$ and $r_i$ become accepting if $m^3 + 1 + i + 1 \equiv |\sigma| \pmod{d}$. In this way, Case 1 is treated correctly, since now two different paths in $M'$ are accepting for the same length.

For case 2, we also consider each $\sigma \in \Psi_\alpha$. Here, only state $s_i$ becomes accepting if $m^3 + 1 + i + 1 \equiv |\sigma| \pmod{d}$.

In this way, Case 2 is treated correctly, since only one path is made accepting. However, it may be that $r_i$ was already accepting. This means that the corresponding inputs are also accepted by another superpath.

This concludes the construction of $M'$. Note, if an input is accepted by different superpaths having different cycle length, then it clearly does not belong to $L(M')$, but is also does not belong to $L(M)$.

Conversely, if an input is accepted unambiguously by $M$ then it is accepted also unambiguously by $M'$. So, we conclude $L(M) = L(M')$. Moreover, since the sum of the different cycle lengths is at most $m$ and each cycle length appears at most twice, the total sum of the cycle lengths is at most $2m$. □

Next, we can utilize the normal form to show that the costs for the determinization of *unary* XNFAs are the same (in the order of magnitude) as for NFAs. This is in strict contrast to XNFAs over a general alphabet. The backbone of the construction is similar to the backbone of the construction given in [2]. However, here we have to treat the cases when inputs are accepted at multiple paths.

**Theorem 3.** *Let $n \geq 1$ and $M$ be a unary $n$-state XNFA. Then $e^{\Theta(\sqrt{n \cdot \ln n})}$ states are sufficient for a DFA to accept $L(M)$.*

*Proof.* Given a unary $n$-state XNFA $M$, we first construct an equivalent $O(n^3)$-state XNFA $M'$ in Chrobak normal form as in the proof of Lemma 2. Let $A_1, R_1, A_2, R_2, \ldots, A_k, R_k$, for $k \geq 1$, be the cycles of $M'$, where $|A_i| = |R_i|$, for $1 \leq i \leq k$. We construct the equivalent DFA $M'' = \langle Q, \Sigma, \delta, q_0, F \rangle$ as follows.

First, we take over the initial deterministic tail of $M'$, which has the $m^3 + 2$ states $\{ q'_i \mid 0 \leq i \leq m^3 + 1 \}$, where $m = 2n$ as in the proof of Lemma 2. Then we add one big cycle of length $\ell = \text{lcm}\{|A_1|, |A_2|, \ldots, |A_k|\}$ to the tail. To this end the states from the set $\{ p_i \mid 0 \leq i \leq \ell - 1 \}$ are cyclically connected and a transition from $q_{m^3+1}$ to $p_0$ is added.

Next, we have to identify the accepting states. To this end, all accepting states on the tail remain accepting. So, as for $M'$ all words up to length $m^3 + 1$ are treated correctly.

Then, we assume that each state $p_i$ of the cycle has a counter attached that is initially set to 0. Now, we consider each cycle $A_i$ of $M'$ consisting of the states $\{s_0, s_1, \ldots, s_{d-1}\}$. Whenever a state $s_j$ is accepting, then the counters of all states $\{ p_t \mid t = j + x \cdot d,$ for $0 \leq x \leq \frac{\ell}{d} - 1 \}$ are increased by one. Similarly, for each cycle $R_i$ of $M'$ consisting of the states $\{r_0, r_1, \ldots, r_{d-1}\}$. If a state $r_j$ is accepting, then the counters of all states $\{ p_t \mid t = j + x \cdot d,$ for $0 \leq x \leq \frac{\ell}{d} - 1 \}$ are increased by one.

In a last construction step, all states whose counters are exactly one become accepting, all the others become non-accepting. In this way, all inputs that are accepted by more than one path in $M'$ are rejected in $M''$, and all inputs that are accepted in $M'$ and, thus, in $M$ by exactly one path are accepted by $M''$ as well. So, $L(M) = L(M'')$ and, clearly, $M''$ is a DFA. Moreover, $M''$ has at most

$$m^3 + 2 + \ell \leq (2n)^3 + 2 + \ell \leq (2n)^3 + 2 + F(n) \in e^{\Theta(\sqrt{n \cdot \ln n})}$$

many states. □

It will turn out after Proposition 5 that the upper bound for the determinization in Theorem 3 is tight in the order of magnitude.

## 4 Converting unary NFAs to XNFAs and Vice Versa

Here, again Landau's function

$$F(n) = \max\{ \text{lcm}(c_1, c_2 \ldots, c_l) \mid l \geq 1, c_1, c_2, \ldots, c_l \geq 1, c_1 + c_2 + \cdots + c_l = n \}$$

plays a crucial role. Recall that the $c_i$ always can be chosen to be relatively prime such that $c_1, c_2, \ldots, c_l \geq 2$, $c_1 + c_2 + \cdots + c_l \leq n$, and $\text{lcm}(c_1, c_2, \ldots, c_l) = F(n)$. This, for example, means that the $c_i$ can be prime powers. An interesting and simplifying result in [22] revealed that, instead of prime powers, one can sum up the first prime numbers such that the sum does not exceed the limit $n$. More, precisely, it has been shown in [22] that the following function $G(n)$ is of the same order of magnitude as $F(n)$, that is, $G(n) \in \Theta(F(n))$. Let $p_i$ denote here the $i$th prime number with $p_1 = 2$.

$$G(n) = \max\{\, p_1 \cdot p_2 \cdots p_l \mid l \geq 1 \text{ and } p_1 + p_2 + \cdots + p_l \leq n \,\}$$

In the following theorem we use the function $G(n)$ to describe the worst case state costs of an NFA simulating a unary XNFA.

**Theorem 4.** *Let $n \geq 2$. There exists a unary $(n+1)$-state XNFA $M$ such that every NFA in Chrobak normal form accepting $L(M)$ has at least $G(n)$ states.*

*Proof.* For $n \geq 2$, let $G(n)$ be represented by the product $p_1 \cdot p_2 \cdots p_l$ of the first $l \geq 1$ prime numbers. We consider the XNFA $M = \langle Q, \{a\}, \delta, q_0, F \rangle$ whose state graph has $l$ disjoint cycles. Each cycle $1 \leq i \leq l$ has length $p_i$ and consists of the states $\{\, r_{i,0}, r_{i,1}, \ldots, r_{i,p_i-1} \,\}$, where $\delta(r_{i,h}, a) = \{r_{i,(h+1) \bmod p_i}\}$, for $0 \leq h \leq p_i - 1$. Now, the initial state $q_0$ is nondeterministically connected to the cycles by $\delta(q_0, a) = \{r_{1,1}, r_{2,1}, \ldots, r_{l,1}\}$. The set of accepting states is $F = \{\, r_{i,0} \mid 1 \leq i \leq l \,\}$. By construction, $M$ has at most $n+1$ states.

The language $L(M)$ accepted by $M$ is

$$\{\, a^m \mid \text{there is exactly one } i \in \{1, 2, \ldots, \ell\} \text{ such that } m \equiv 0 \pmod{p_i} \,\}.$$

We define the set of all integers that are not divisible by all $p_i$, $1 \leq i \leq l$, as

$$K = \{\, k \in \mathbb{N} \mid k \text{ is not divisible by all } p_i, 1 \leq i \leq l \,\}.$$

Assume now, that $L(M)$ is accepted by an NFA $M'$ in Chrobak normal form with less than $G(n)$ states, say $m < G(n)$ states.

Our first goal is to show the claim that for any $p_i$, $1 \leq i \leq l$, all cycles in the state graph of $M'$ on which infinitely many words from $\{\, a^{x \cdot p_i} \mid x \in K \,\}$ are accepted, have a length that is divisible by $p_i$.

Since all words from the infinite set $\{\, a^{x \cdot p_i} \mid x \in K \,\}$ belong to $L(M)$, cycles on which infinitely many such words are accepted exist. Assume that one of these cycles has a length $c$ not divisible by $p_i$ and let $a^{x_0 \cdot p_i}$ with $x_0 \in K$ be one of the accepted words. Then, the word $w = a^{x_0 \cdot p_i + c \cdot p}$ with $p = \frac{G(n)}{p_i}$ is accepted as well. But since $c$ and $p$ are not divisible by $p_i$, we have that $|w|$ is not divisible by $p_i$, either. Moreover, since $x_0 \cdot p_i$ is not divisible by any $p_j$ with $i \neq j$ but $c \cdot p$ is, we have that $|w|$ is not divisible by any $p_j$ with $i \neq j$, either. So, $w$ cannot belong to $L(M')$. From this contradiction the claim follows.

Since $m < G(n)$, there must be two cycles $C_1$ and $C_2$, say of length $c_1$ and $c_2$, such that there are two different prime numbers $p_i \neq p_j$ with $1 \leq i, j \leq l$, where $c_1$ is divisible by $p_i$ but not divisible by $p_j$ and infinitely many words from $\{\, a^{x \cdot p_i} \mid x \in K \,\}$ are accepted in $C_1$, and where $c_2$ is divisible by $p_j$ but not divisible by $p_i$ and infinitely many words from $\{\, a^{x \cdot p_j} \mid x \in K \,\}$ are accepted in $C_2$. Since $p_j$ is relatively prime to $c_1$, there is an integer $p$ such that $p \cdot c_1 \equiv 1 \pmod{p_j}$. Consider some word $w = a^{x_0 \cdot p_i}$ with $x_0 \in K$ that is accepted in $C_1$. Then, the word $a^{x_1 \cdot p \cdot c_1 + |w|}$ with $(x_1 + |w|) \equiv 0 \pmod{p_j}$ is accepted in $C_1$ as well. However, this word does not belong to $L(M)$, since it is divisible by $p_i$ and $p_j$.

So, from this contradiction we conclude there is no NFA in Chrobak normal form with less than $G(n)$ states. $\square$

Clearly the upper bound for the simulation of an XNFA by an NFA is given by determinization. Thus, we have the following proposition.

**Proposition 5.** *Let $n \geq 2$ and $M$ be a unary $n$-state XNFA. Then $e^{\Theta(\sqrt{n \cdot \ln n})}$ states are sufficient for an NFA to accept $L(M)$.*

The lower bound in Theorem 4 says that there are $(n+1)$-state XNFAs such that any equivalent NFA *in Chrobak normal form* has at least $G(n)$ states. Moreover, any $n$-state NFA can be converted into an equivalent NFA in Chrobak normal form that has at most $O(n^2)$ states. So, since $G(n) \in \Theta(F(n))$ [22], the lower bound for the state costs of the simulation of an $n$-state XNFA by an NFA (not necessarily in Chrobak normal form) is

$$\Theta(\sqrt{G(n-1)}) = \Theta(\sqrt{e^{\Theta(\sqrt{(n-1)\cdot\ln(n-1)})}}) = e^{\Theta(\sqrt{n\cdot\ln n})}.$$

So, we conclude that the upper bound for the unary XNFA-to-DFA conversion shown in Theorem 3 and the upper bound for the unary XNFA-to-NFA conversion shown in Proposition 5 are tight in the order of magnitude.

We turn to the simulation of NFAs by XNFAs. In [25] it has been shown that the language

$$L = \{\, a^n \mid n \not\equiv 0 \;(\bmod\; \mathrm{lcm}(c_1, c_2, \ldots, c_k))\,\} \cup \{\lambda\},$$

for $k \geq 1$ and $c_1, c_2, \ldots, c_k \geq 2$ is accepted by an NFA with $1 + \sum_{i=1}^{k} c_i$ states, while the smallest UFA for $L$ needs at least $1 + \mathrm{lcm}(c_1, c_2, \ldots, c_k)$ many states. The proof of the lower bound is based on a method given in [32] which is based on a rank argument on certain matrices. After a thorough analysis of the arguments of the method, it turned out that exclusively accepting computations of the UFAs are used. In other words, the arguments can be applied to XNFAs as well. So, we derive that also the smallest XNFA needs at least $1 + \mathrm{lcm}(c_1, c_2, \ldots, c_k)$ states to accept the language $L$. So, we have the following lower bound.

**Theorem 6.** *Let $n \geq 2$. There exists a unary $(n+1)$-state NFA $M$ such that every XNFA accepting $L(M)$ has at least $F(n)+1$ states.*

Clearly the upper bound for the simulation of an NFA by an XNFA is given by determinization. Thus, we have the following proposition.

**Proposition 7.** *Let $n \geq 2$ and $M$ be a unary $n$-state NFA. Then $e^{\Theta(\sqrt{n\cdot\ln n})}$ states are sufficient for an XNFA to accept $L(M)$.*

As before, we also conclude here that the lower bound and upper bound are tight in the order of magnitude.

## 5 Computational Complexity

In this section, we discuss the computational complexity of decidability questions. In particular, we consider general membership, emptiness, universality, inclusion, and equivalence with respect to the unary case. These problems have been studied in [13, 14] in case of general alphabets. It turns out here that the general membership problem in the unary case shares the same computational complexity with the general case, namely, both problems are NL-complete. However, the questions of emptiness, universality, inclusion, and equivalence turn out to be coNP-complete in the unary case, whereas these questions have been shown to be PSPACE-complete in the general case [13, 14].

**Theorem 8.** *The problem of testing the general membership for unary XNFAs is* NL*-complete.*

*Proof.* To show that the problem is in NL for unary XNFAs we can use the same construction that has been described in [13, 14] for general alphabets. The basic idea is to test whether an input $w$ is not accepted by a given XNFA $A$. This means that either there is no accepting path in the computation tree for $w$ or there are at least two accepting paths. In the first case, the input $w$ is not accepted by $A$ even if $A$ is considered as an NFA. Hence, this case can be solved in NL using the known algorithms for NFAs. The second case can be checked by guessing two different accepting paths in the computation tree. To this end, one has to keep track of two states representing the current position on the two paths. Since this can be realized in NL, the general membership problem is in NL in particular for unary XNFAs.

To show the NL-hardness of the general membership problem for unary XNFAs we can in principle apply the reduction that is described in [14] for general alphabets. To adapt it to the unary case we have to use the fact that the membership problem for unary NFAs remains NL-complete (see, e.g., [11]) and we have to observe that the XNFA constructed in the reduction is unary, since the given NFA is unary. Since the reduction described in [14] is not yet published we provide the reduction here for the sake of completeness.

To show the NL-hardness of the general membership problem we reduce the non-membership problem for NFAs which is known to be NL-complete, since the membership problem for NFAs is NL-complete.

Let $\langle A,w\rangle$ be the encoding of an NFA $A = \langle Q,\{a\},\delta,q_0,F\rangle$ and an input word $w$. We construct an XNFA $A' = \langle Q\cup\{p_0,p\},\{a\},\delta',p_0,F'\rangle$, where $p_0$ and $p$ are two new states not belonging to $Q$. The accepting states $F'$ are defined as $F' = F\cup\{p_0,p\}$, if $\lambda \in L(A)$, and $F' = F\cup\{p\}$ otherwise. The transition function $\delta'$ is defined as follows. First, $A'$ has the same behavior as $A$ on states from $Q$. Formally, $\delta'(q,a) = \delta(q,a)$ for all $q \in Q$. Second, from the new initial state $p_0$ all states are reached that are reached from the initial state $q_0$ of $A$. Additionally, the new state $p$ is reached from $p_0$. Formally, $q' \in \delta'(p_0,a)$, if $q' \in \delta(q_0,a)$, and $p \in \delta'(p_0,a)$. Finally, the state $p$ acts as an accepting sink state, that is, $p \in \delta'(p,a)$.

The reduction from the encoding $\langle A,w\rangle$ to an encoding $\langle A',w\rangle$ can be realized by a deterministic logarithmically space-bounded Turing machine.

For the correctness of the reduction we have to show that the XNFA $A'$ accepts $w$ if and only if $w$ is not accepted by the NFA $A$. On the one hand, if $w$ is accepted by $A'$, then $p \in \delta'(p_0,w)$ and $\delta'(p_0,w)\cap F = \emptyset$, since otherwise there would be at least two accepting paths for $w$. Hence, $w$ is not accepted by the NFA $A$. On the other hand, if $w$ is not accepted by $A'$, then $p \in \delta'(p_0,w)$ and $\delta'(p_0,w)\cap F \neq \emptyset$, since there must be at least two accepting paths for $w$. Hence, $w$ is accepted by the NFA $A$. This concludes the correctness of the reduction and shows the NL-hardness of the general membership problem for XNFAs. Altogether, we obtain that the general membership problem for XNFAs is NL-complete. □

It is known that the emptiness problem for unary NFAs is NL-complete. In contrast, we show the problem becomes coNP-complete for unary XNFAs. In the following proofs we need a result obtained in [13, 14] on the conversion of XNFAs to DFAs in case of general alphabets.

**Theorem 9.** *[13, 14] Let $n \geq 1$ and $M$ be an $n$-state XNFA. Then $3^n - 2^n + 1$ states are sufficient for a DFA to accept $L(M)$.*

**Theorem 10.** *The emptiness problem for unary XNFAs is* coNP*-complete.*

*Proof.* We will show that the non-emptiness problem for unary XNFAs is NP-complete which implies that the emptiness problem is coNP-complete. To show that the non-emptiness problem belongs to NP we use a similar approach as described in Theorem 6.1 in [35]. Let $M$ be an XNFA over a unary alphabet $\{a\}$ with state set $Q = \{q_1,q_2,\dots,q_n\}$, initial state $q_1$, and transition function $\delta$. By applying Theorem 9 we know that there exists an equivalent DFA that has at most $3^n$ states. It is clear that $L(M)$ is not empty if and only if $M$ accepts a word of length $m \leq 3^n$.

Now, the idea is first to guess a length $m \leq 3^n$ in ternary representation $m_1 m_2 \cdots m_n$ and to check whether there is exactly one path of length $m$ in $M$ leading from the initial state to an accepting state. The latter can be realized by mapping the transition function of $M$ to its corresponding adjacency matrix $A_M$ where we set an entry $A_M[i,j] = 1$ if and only if $q_j \in \delta(q_i, a)$, for $1 \leq i,j \leq n$. Then, $a^m \in L(M)$ if and only if the first row of $A_M^m$ has exactly one entry corresponding to an accepting state with value 1. Thus, we have as second task to compute the matrix product $A_M^m = A_M^{m_1 \cdot 3^{n-1}} \cdot A_M^{m_2 \cdot 3^{n-2}} \cdots \cdot A_M^{m_n}$ by inspecting the ternary counter. The matrix $A_M^m$ can be computed by successively cubing and multiplying $A_M$. For example, let $m = 22$ and its ternary notation be 211. Then, we have to multiply $A_M \cdot A_M^3 \cdot A_M^9 \cdot A_M^9$. In general, we have at most $3 \cdot 2\log_3(m) \leq 6n$ matrix multiplications. Since every matrix multiplication can be realized in time $n^2$, we obtain that $A_M^m$ can be computed in deterministic time bounded by a polynomial in $n$. Finally, the first row of the resulting matrix $A_M^m$ has to be inspected. Altogether, these three tasks can be realized in nondeterministic time bounded by a polynomial in $n$. Hence, the complete procedure is in NP.

To show that the non-emptiness problem is NP-hard we use again a similar approach as described in Theorem 6.1 in [35]. It is shown there that a given Boolean formula in conjunctive form with exactly three literals per conjunct is satisfiable if and only if a regular unary language $L$ described by a regular expression is not equal to $\{a\}^*$. Moreover, the reduction is computable in logarithmic space. Since a language described by a regular expression can equivalently be described by an NFA of similar size, we let now $L$ be described by an NFA $M$. Moreover, we construct a one-state DFA $M'$ that accepts $\{a\}^*$. Then, we construct an XNFA $M''$ that initially guesses whether it simulates for the complete input the NFA $M$ or the DFA $M'$. Since $M''$ is an XNFA we obtain that $L(M) = \{a\}^*$ if and only if $L(M'') = \emptyset$. Hence, we have $L(M'') \neq \emptyset$ if and only if $L(M) \neq \{a\}^*$ if and only if the given Boolean formula is satisfiable. Since the constructions of $M$, $M'$, and $M''$ can be realized in logarithmic space, we obtain the NP-hardness of the non-emptiness problem for XNFAs and, thus, the coNP-hardness of the emptiness problem for XNFAs. □

**Theorem 11.** *The problems of testing universality, inclusion, and equivalence for unary XNFAs are* coNP*-complete.*

*Proof.* Let us first show that the problems of testing non-universality, non-inclusion, and non-equivalence for unary XNFAs are in NP. We start with the non-universality problem. Let $M$ be an $n$-state XNFA. By applying Theorem 9 we know that there exists an equivalent DFA that has at most $3^n$ states. Hence, $L(M) \neq \{a\}^*$ if and only if there is a word of length $m \leq 3^n$ that is not accepted by $M$. Similar to the proof of Theorem 10 we can guess a ternary representation of that word, compute $A_M^m$, and check that the guessed word is not accepted by $M$ by inspecting the first row whether there is no entry corresponding to an accepting state with value 1. According to the considerations made in the proof of Theorem 10 the procedure can be realized in nondeterministic polynomial time and we obtain that the non-universality problem is in NP. Hence, the universality problem is in coNP.

Next, we consider the non-inclusion problem. Let $M_1$ be an $n_1$-state XNFA and $M_2$ be an $n_2$-state XNFA. By applying Theorem 9 we know that there exist equivalent DFAs having at most $3^{n_1}$ states and $3^{n_2}$ states, respectively. Hence, $L(M_1) \not\subseteq L(M_2)$ if and only if $L(M_1) \cap \overline{L(M_2)} \neq \emptyset$ if and only if there is a word of length $m \leq 3^{n_1+n_2}$ that is accepted by $M_1$, but not accepted by $M_2$. Similar to the proof of Theorem 10 and to the above construction for the non-universality problem we obtain that the non-inclusion problem is in NP. Hence, the inclusion problem is in coNP.

Finally, we consider the equivalence problem. Let $M_1$ and $M_2$ be two XNFAs. Since the inclusion problem is in coNP, we obtain that the equivalence problem is coNP by testing $L(M_1) \subseteq L(M_2)$ and

$L(M_2) \subseteq L(M_1)$.

To show the coNP-hardness of the problems we shortly describe how the reduction given in the proof of Theorem 10 has to be extended. We recall that we have constructed an XNFA $M''$ such that $L(M'') \neq \emptyset$ if and only if the given Boolean formula is satisfiable.

For non-universality we construct another XNFA $A$ that initially guesses whether it simulates for the complete input the XNFA $M''$ or the DFA $M'$ accepting $\{a\}^*$. Then, we have $L(A) \neq \{a\}^*$ if and only if $L(M'') \neq \emptyset$ and obtain the NP-hardness of non-universality. For the equivalence problem we consider $M'$ as an XNFA and have $L(A) = L(M') = \{a\}^*$ if and only if $L(M'') = \emptyset$, which gives the coNP-hardness of the equivalence problem. Finally, we have $L(M') \subseteq L(A)$ if and only if $L(A) = L(M')$ if and only if $L(M'') = \emptyset$ and obtain the coNP-hardness of the inclusion problem. □

The computational complexity results in the unary case are summarized in Table 1.

| | DFA | NFA | XNFA | AFA |
|---|---|---|---|---|
| membership | L | NL | NL | P |
| emptiness | L | NL | coNP | PSPACE |
| universality | L | coNP | coNP | PSPACE |
| inclusion | L | coNP | coNP | PSPACE |
| equivalence | L | coNP | coNP | PSPACE |

Table 1: Computational complexity results for the decidability problems in the unary case. All problems are complete with respect to the complexity class indicated. The results for XNFAs are obtained in this paper. The remaining results and pointers to the literature are summarized, for example, in the survey [10].

## References

[1] Andreas Blass & Yuri Gurevich (1982): *On the Unique Satisfiability Problem.* Inform. Control 55, pp. 80–88, doi:10.1016/S0019-9958(82)90439-9.

[2] Marek Chrobak (1986): *Finite automata and unary languages.* Theor. Comput. Sci. 47, pp. 149–158, doi:10.1016/0304-3975(86)90142-8. Errata: [3].

[3] Marek Chrobak (2003): *Errata to "Finite automata and unary languages".* Theor. Comput. Sci. 302, pp. 497–498, doi:10.1016/S0304-3975(03)00136-1.

[4] Keith Ellul (2004): *Descriptional Complexity Measures of Regular Languages.* Master's thesis, University of Waterloo, Ontario, Canada.

[5] Viliam Geffert (2007): *Magic numbers in the state hierarchy of finite automata.* Inform. Comput. 205(11), pp. 1652–1670, doi:10.1016/j.ic.2007.07.001.

[6] Viliam Geffert, Carlo Mereghetti & Giovanni Pighizzini (2003): *Converting two-way nondeterministic unary automata into simpler automata.* Theor. Comput. Sci. 295, pp. 189–203, doi:10.1016/S0304-3975(02)00403-6.

[7] Yahya Ould Hamidoune (1979): *Sur les parcours hamiltoniens dans les graphes orientes.* Discrete Mathematics 26, pp. 227–234, doi:10.1016/0012-365X(79)90028-1.

[8] Lane A. Hemaspaandra & Mitsunori Ogihara (2002): *The Complexity Theory Companion.* Springer, doi:10.1007/978-3-662-04880-1.

[9] Markus Holzer & Martin Kutrib (2003): *Unary Language Operations and Their Nondeterministic State Complexity*. In M. Ito & M. Toyama, editors: Developments in Language Theory (DLT 2002), LNCS 2450, Springer, pp. 162–172, doi:10.1007/3-540-45005-X_14.

[10] Markus Holzer & Martin Kutrib (2011): *Descriptional and Computational Complexity of Finite Automata – A Survey*. Inform. Comput. 209, pp. 456–470, doi:10.1016/J.IC.2010.11.013.

[11] Neil D. Jones (1975): *Space-Bounded Reducibility among Combinatorial Problems*. J. Comput. Syst. Sci. 11, pp. 68–85, doi:10.1016/S0022-0000(75)80050-X.

[12] Michal Kunc & Alexander Okhotin (2012): *State complexity of operations on two-way finite automata over a unary alphabet*. Theor. Comput. Sci. 449, pp. 106–118, doi:10.1016/J.TCS.2012.04.010.

[13] Martin Kutrib, Andreas Malcher & Matthias Wendlandt (2023): *Complexity of Exclusive Nondeterministic Finite Automata*. In Henning Bordihn, Nicholas Tran & György Vaszil, editors: Descriptional Complexity of Formal Systems (DCFS 2023), LNCS 13918, Springer, pp. 121–133, doi:10.1007/978-3-031-34326-1_9.

[14] Martin Kutrib, Andreas Malcher & Matthias Wendlandt (2024): *Complexity of Exclusive Nondeterministic Finite Automata*. submitted for journal publication.

[15] Edmund Landau (1903): *Über die Maximalordnung der Permutationen gegebenen Grades*. Archiv der Math. und Phys. 3, pp. 92–103.

[16] Edmund Landau (1909): *Handbuch der Lehre von der Verteilung der Primzahlen*. Teubner, Leipzig.

[17] Hing Leung (1998): *Separating Exponentially Ambiguous Finite Automata from Polynomially Ambiguous Finite Automata*. SIAM J. Comput. 27, pp. 1073–1082, doi:10.1137/S0097539793252092.

[18] Hing Leung (2005): *Descriptional complexity of NFA of different ambiguity*. Int. J. Found. Comput. Sci. 16, pp. 975–984, doi:10.1142/S0129054105003418.

[19] Filippo Mera & Giovanni Pighizzini (2005): *Complementing unary nondeterministic automata*. Theor. Comput. Sci. 330, pp. 349–360, doi:10.1016/J.TCS.2004.04.015.

[20] Carlo Mereghetti & Giovanni Pighizzini (2001): *Optimal Simulations between Unary Automata*. SIAM J. Comput. 30, pp. 1976–1992, doi:10.1137/S009753979935431X.

[21] Albert R. Meyer & Michael J. Fischer (1971): *Economy of Description by Automata, Grammars, and Formal Systems*. In: Symposium on Switching and Automata Theory (SWAT 1971), IEEE, pp. 188–191, doi:10.1109/SWAT.1971.11.

[22] William Miller (1987): *The maximum order of an element of a finite symmetric group*. Am. Math. Mon. 94, pp. 497–506, doi:10.1080/00029890.1987.12000673.

[23] Frank R. Moore (1971): *On the Bounds for State-Set Size in the Proofs of Equivalence Between Deterministic, Nondeterministic, and Two-Way Finite Automata*. IEEE Trans. Comput. 20(10), pp. 1211–1214, doi:10.1109/T-C.1971.223108.

[24] J.-L. Nicolas (1968): *Sur l'ordre maximum d'un élément dans le groupe $S_n$ des permutations*. Acta Arith. 14, pp. 315–332, doi:10.4064/aa-14-3-315-332.

[25] Alexander Okhotin (2012): *Unambiguous finite automata over a unary alphabet*. Inform. Comput. 212, pp. 15–36, doi:10.1016/J.IC.2012.01.003.

[26] Giovanni Pighizzini (2009): *Deterministic Pushdown Automata and Unary Languages*. Int. J. Found. Comput. Sci. 20(4), pp. 629–645, doi:10.1142/S0129054109006784.

[27] Giovanni Pighizzini (2015): *Investigations on Automata and Languages Over a Unary Alphabet*. Int. J. Found. Comput. Sci. 26, pp. 827–850, doi:10.1142/S012905411540002X.

[28] Giovanni Pighizzini & Jeffrey Shallit (2002): *Unary Language Operations, State Complexity and Jacobsthal's Function*. Int. J. Found. Comput. Sci. 13, pp. 145–159, doi:10.1142/S012905410200100X.

[29] Giovanni Pighizzini, Jeffrey Shallit & Ming-Wei Wang (2002): *Unary Context-Free Grammars and Pushdown Automata, Descriptional Complexity and Auxiliary Space Lower Bounds*. J. Comput. Syst. Sci. 65, pp. 393–414, doi:10.1006/JCSS.2002.1855.

[30] Michael Oser Rabin & Dana Scott (1959): *Finite Automata and Their Decision Problems*. *IBM J. Res. Dev.* 3, pp. 114–125, doi:10.1147/rd.32.0114.

[31] William J. Sakoda & Michael Sipser (1978): *Nondeterminism and the size of two way finite automata*. In ACM, editor: *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing (STOC 1978)*, ACM, ACM Press, New York, pp. 275–286, doi:10.1145/800133.804357.

[32] Erik Meineche Schmidt (1978): *Succinctness of Dscriptions of Context-Free, Regular and Finite Languages*. Ph.D. thesis, Cornell University, Ithaca, NY.

[33] Jeffrey Shallit (2008): *The Frobenius Problem and Its Generalizations*. In Masami Ito & Masafumi Toyama, editors: *Developments in Language Theory (DLT 2008)*, *LNCS* 5257, Springer, pp. 72–83, doi:10.1007/978-3-540-85780-8_5.

[34] Michael Sipser (1980): *Lower Bounds on the Size of Sweeping Automata*. *J. Comput. Syst. Sci.* 21, pp. 195–202, doi:10.1016/0022-0000(80)90034-3.

[35] Larry. J. Stockmeyer & A. R. Meyer (1973): *Word Problems Requiring Exponential Time*. In ACM, editor: *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing (STOC 1973)*, ACM Press, New York, NY, USA, pp. 1–9, doi:10.1145/800125.804029.

[36] M. Szalay (1980): *On the maximal order in $S_n$ and $S_n^*$*. *Acta Arithm.* 37, pp. 321–331, doi:10.4064/aa-37-1-321-331.

[37] Anthony Widjaja To (2009): *Unary finite automata vs. arithmetic progressions*. *Inform. Process. Lett.* 109, pp. 1010–1014, doi:10.1016/J.IPL.2009.06.005.

# Repetitive Finite Automata With Translucent Letters

František Mráz
Charles University
Department of Computer Science
Malostranské nám. 25
118 00 PRAHA, Czech Republic
frantisek.mraz@mff.cuni.cz

Friedrich Otto
Universität Kassel
Fachbereich Elektrotechnik/Informatik
34109 KASSEL, Germany
f.otto@uni-kassel.de

Here we propose an extension of the (deterministic and the nondeterministic) finite automaton with translucent letters (DFAwtl and NFAwtl), which lies between these automata and their non-returning variants (that is, the nr-DFAwtl and the nr-NFAwtl). This new model works like a DFAwtl or an NFAwtl, but on seeing the end-of-tape marker, it may change its internal state and continue with its computation instead of just ending it, accepting or rejecting. This new type of automaton is called a *repetitive deterministic* or *nondeterministic finite automaton with translucent letters* (*RDFAwtl* or *RNFAwtl*). In the deterministic case, the new model is strictly more expressive than the DFAwtl, but less expressive than the nr-DFAwtl, while in the nondeterministic case, the new model is equivalent to the NFAwtl.

## 1 Introduction

While a finite automaton reads its input strictly from left to right, letter by letter, by now many types of automata have been considered in the literature that process their inputs in a different, more involved way. Under this aspect, the most extreme is the *jumping finite automaton* of Meduna and Zemek [7] (see also [5]), which, after reading a letter, jumps to an arbitrary position of the remaining input. It is known that the jumping finite automaton accepts languages that are not even context-free, like the language $\{ w \in \{a,b,c\}^* \mid |w|_a = |w|_b = |w|_c \}$, but at the same time, it does not even accept the finite language $\{ab\}$.

Another example is the *restarting automaton* as introduced by Jančar, Mráz, Plátek, and Vogel in [6], which processes a given input in cycles. In each cycle, a restarting automaton scans its tape contents from left to right, using a window of a fixed finite size, until it executes a delete/restart operation. Such an operation deletes one or more letters from the current contents of the window, returns the window to the left end of the tape, and resets the automaton to its initial state. If a window of size larger than one is used, these so-called R-automata accept a proper superclass of the regular languages that is incomparable to the context-free and the growing context-sensitive languages with respect to inclusion (see, e.g., [17]). However, with a window of size one, the R-automata accept exactly the regular languages [8].

Finally, there is the (deterministic and nondeterministic) finite automaton *with translucent letters* (or DFAwtl and NFAwtl) of Nagy and Otto [13], which is equivalent to a cooperating distributed system of stateless deterministic R-automata with windows of size one. For each state $q$ of an NFAwtl, there is a set $\tau(q)$ of *translucent letters*, which is a subset of the input alphabet that contains those letters that the automaton cannot see when it is in state $q$. Accordingly, in each step, the NFAwtl just reads (and deletes) the first letter from the left which it can see, that is, which is not translucent for the current state. Here, it is important to notice that, in each step, an NFAwtl reads the current tape contents from the very left, that is, after deleting a letter, it returns its head to the first letter of the remaining tape contents. It has been shown that the NFAwtl accepts a class of semi-linear languages that properly contains all rational trace languages, while its deterministic variant, the DFAwtl, is properly less expressive. In fact, the DFAwtl just accepts a class of languages that is incomparable to the rational trace languages with respect to inclusion [12, 14, 15, 16]. Although the NFAwtl is quite expressive, it cannot even accept the deterministic linear language $L_2 = \{ a^n b^n \mid n \geq 0 \}$, as such an automaton cannot possibly compare the number of occurrences of the letter $a$ with the number of occurrences of the letter $b$ and ensure, at the same time, that all $a$'s precede the first $b$.

To make up for this shortcoming, a variant of the finite automaton with translucent letters has been proposed in [9], which, after reading and deleting a letter, does not return its head to the first letter of the remaining tape contents, but that rather continues from the position of the letter just deleted. This means that, in general, only a scattered subword of the input has been read and deleted before the head reaches the end of the input. If the computation is now required to halt, either accepting or rejecting, then it can easily be shown that this type of automaton just accepts the class of regular languages. For the right one-way jumping finite automaton of [1, 3], this problem is overcome by cyclically shifting all the translucent letters that are encountered during a computation to the end of the current tape contents. In this way, these letters may be read and deleted at a later stage of the computation. For the type of automaton proposed in [9], a different approach was taken. When the head of the automaton reaches the end of the input, which is marked by a special end-of-tape marker, then the automaton can decide whether to accept, reject, or continue, which means that it changes its state and again reads the remaining tape contents from the beginning.

It has been established that this type of automaton, called a *non-returning finite automaton with translucent letters* or an *nr-NFAwtl*, is strictly more expressive than the NFAwtl. This result also holds for the deterministic case, although the deterministic variant, the *nr-DFAwtl*, is still not sufficiently expressive to accept all rational trace languages. In [10], the nr-DFAwtl and the nr-NFAwtl are compared to the jumping finite automaton, the right one-way jumping finite automaton of [1, 3], and the right-revolving finite automaton of [2], deriving the complete taxonomy of the resulting classes of languages. As it turns out, the nr-DFAwtl can be seen as an extension of the right one-way jumping finite automaton that can detect the end of its input.

When we look at the above description of the generalization of the NFAwtl to the nr-NFAwtl, then we realize that this generalization actually consists of two steps:

- The head of the automaton does not return to the left end of the tape after a letter has been read and deleted. This is the *non-returning* property (in contrast to the *returning* property of the NFAwtl).
- Once the end-of-tape marker is reached, an nr-NFAwtl may execute a step that changes its state and returns its head to the left end of the tape. We call this the property of being *repetitive* (in contrast to the *non-repetitiveness* of the NFAwtl, which must immediately halt as soon as its head reaches the end-of-tape marker).

In this paper, we study the influence that these two properties have on the expressive capacity of the finite automaton with translucent letters in detail. As we consider both, the deterministic and non-deterministic variants of the resulting types of automata, we obtain eight different classes of automata with translucent letters. In addition, we also include the (deterministic and the non-deterministic) right one-way jumping finite automaton in our study. We shall derive the complete taxonomy of the resulting language classes, which will nicely illustrate the effects that the non-returning property and the repetitiveness have.

Actually, we shall only encounter one new language class that has not been considered before: the class $\mathscr{L}(\mathsf{RDFAwtl})$ of languages that are accepted by repetitive DFAwtls. After presenting the necessary notation and definitions in Section 2, we shall present our results on repetitive DFAwtls and repetitive

NFAwtls in Section 3. Here, it turns out that the repetitive DFAwtl (the RDFAwtl) is strictly more expressive than the DFAwtl, while its nondeterministic variant, the repetitive NFAwtl (the RNFAwtl), is equivalent to the NFAwtl. Then, in Section 4, we consider closure and non-closure properties for the language class $\mathscr{L}(\mathsf{RDFAwtl})$. Finally, in the concluding section, we address the membership problem and some other decision problems for the RDFAwtl in short, and we state some open problems.

## 2 Definitions

First, we restate the definition of the nondeterministic finite automaton with translucent letters and its deterministic variant, the DFAwtl, from [13].

**Definition 1** *A* finite automaton with translucent letters*, or an* NFAwtl *for short, is defined as a 7-tuple $A = (Q, \Sigma, \lhd, \tau, I, F, \delta)$, where $Q$ is a finite set of internal states, $\Sigma$ is a finite alphabet of input letters, $\lhd \notin \Sigma$ is a special letter that is used as an* end-of-tape marker, *$\tau : Q \to \mathscr{P}(\Sigma)$ is a* translucency mapping, *$I \subseteq Q$ is a set of initial states, $F \subseteq Q$ is a set of final states, and $\delta : Q \times \Sigma \to \mathscr{P}(Q)$ is a* transition relation. *Here, we require that, for each state $q \in Q$ and each letter $a \in \Sigma$, if $a \in \tau(q)$, then $\delta(q,a) = \emptyset$.*

*An NFAwtl $A = (Q, \Sigma, \lhd, \tau, I, F, \delta)$ is a* deterministic finite automaton with translucent letters*, abbreviated as* DFAwtl*, if $|I| = 1$ and $|\delta(q,a)| \leq 1$ for all $q \in Q$ and all $a \in \Sigma$.*

A configuration of $A$ is a word from the set $Q \cdot \Sigma^* \cdot \lhd \cup \{\mathsf{Accept}, \mathsf{Reject}\}$. A configuration of the form $qw \cdot \lhd$, where $q \in Q$ and $w \in \Sigma^*$, expresses the situation that $A$ is in state $q$, its tape contains the word $w$ followed by the sentinel $\lhd$, and the head of $A$ is on the first letter of $w \cdot \lhd$. For an input word $w \in \Sigma^*$, a corresponding initial configuration is of the form $q_0 w \cdot \lhd$, where $q_0 \in I$. The NFAwtl $A$ induces the following single-step computation relation on its set of configurations:

$$qw \cdot \lhd \vdash_A \begin{cases} q'uv \cdot \lhd, & \text{if } w = uav,\ u \in (\tau(q))^*,\ a \in \Sigma \smallsetminus \tau(q),\ v \in \Sigma^*, \text{ and } q' \in \delta(q,a), \\ \mathsf{Reject}, & \text{if } w = uav,\ u \in (\tau(q))^*,\ a \in \Sigma \smallsetminus \tau(q),\ v \in \Sigma^*, \text{ and } \delta(q,a) = \emptyset, \\ \mathsf{Accept}, & \text{if } w \in (\tau(q))^* \text{ and } q \in F, \\ \mathsf{Reject}, & \text{if } w \in (\tau(q))^* \text{ and } q \notin F. \end{cases}$$

Thus, in each step, $A$ reads and deletes the first letter from the left that is not translucent for the current state. In addition, if all letters on the tape are translucent for the current state, then $A$ halts, and it accepts if the current state is final. A word $w \in \Sigma^*$ is *accepted by $A$* if there exists an initial state $q_0 \in I$ and a computation $q_0 w \cdot \lhd \vdash_A^* \mathsf{Accept}$, where $\vdash_A^*$ denotes the reflexive transitive closure of the single-step computation relation $\vdash_A$. Now, $L(A) = \{ w \in \Sigma^* \mid w \text{ is accepted by } A \}$ is the *language accepted by $A$*, and $\mathscr{L}(\mathsf{NFAwtl})$ denotes the class of all languages that are accepted by NFAwtls. Analogously, $\mathscr{L}(\mathsf{DFAwtl})$ denotes the class of all languages that are accepted by DFAwtls.

Next, we define the first of the two possible extensions of the automaton with translucent letters that we mentioned above.

**Definition 2** *A* repetitive *finite automaton with translucent letters, or an* RNFAwtl*, is specified through a 6-tuple $A = (Q, \Sigma, \lhd, \tau, I, \delta)$, where $Q$, $\Sigma$, $\lhd$, $\tau$, and $I$ are defined as for an NFAwtl, while the transition relation $\delta$ is a mapping $\delta : (Q \times (\Sigma \cup \{\lhd\})) \to (\mathscr{P}(Q) \cup \{\mathsf{Accept}\})$. Here, it is required that, for each state $q \in Q$ and each letter $a \in \Sigma$, $\delta(q,a) \subseteq Q$ and, if $a \in \tau(q)$, then $\delta(q,a) = \emptyset$. In addition, for each state $q \in Q$, $\delta(q, \lhd)$ is either a subset of $Q$ or $\delta(q, \lhd) = \mathsf{Accept}$.*

*The set of configurations for an RNFAwtl is the same as for an NFAwtl. The RNFAwtl A induces the following single-step computation relation on its set of configurations:*

$$qw\cdot\lhd \vdash_A \begin{cases} q'uv\cdot\lhd & \text{if } w=uav,\, u\in(\tau(q))^*, a\in\Sigma\smallsetminus\tau(q), v\in\Sigma^*, \text{ and } q'\in\delta(q,a),\\ \mathsf{Reject} & \text{if } w=uav,\, u\in(\tau(q))^*, a\in\Sigma\smallsetminus\tau(q), v\in\Sigma^*, \text{ and } \delta(q,a)=\emptyset,\\ \mathsf{Accept} & \text{if } w\in(\tau(q))^* \text{ and } \delta(q,\lhd)=\mathsf{Accept},\\ \mathsf{Reject} & \text{if } w\in(\tau(q))^* \text{ and } \delta(q,\lhd)=\emptyset,\\ q'w\cdot\lhd & \text{if } w\in(\tau(q))^* \text{ and } q'\in\delta(q,\lhd).\end{cases}$$

*Thus, if all letters on the tape are translucent for the current state $q$ and $\delta(q,\lhd)\subseteq Q$ is nonempty, then A changes its state to $q'\in\delta(q,\lhd)$ and continues with its computation. The language $L(A)$ accepted by A is defined as $L(A)=\{w\in\Sigma^*\mid w \text{ is accepted by } A\}$, that is, it consists of all words for which A has an accepting computation, and $\mathscr{L}(\mathsf{RNFAwtl})$ denotes the class of all languages that are accepted by RNFAwtls.*

*An RNFAwtl $A=(Q,\Sigma,\lhd,\tau,I,\delta)$ is a* repetitive deterministic finite automaton with translucent letters, *or an* RDFAwtl, *if $|I|=1$ and $|\delta(q,a)|\leq 1$ for all $q\in Q$ and all $a\in\Sigma\cup\{\lhd\}$. $\mathscr{L}(\mathsf{RDFAwtl})$ denotes the class of all languages that are accepted by RDFAwtls.*

Obviously, each NFAwtl can easily be turned into an RNFAwtl for the same language. Of course, the same applies to a DFAwtl, giving an RDFAwtl for the same language. Thus, we have the following inclusion relations.

**Proposition 3** $\mathscr{L}(\mathsf{DFAwtl})\subseteq\mathscr{L}(\mathsf{RDFAwtl})$ *and* $\mathscr{L}(\mathsf{NFAwtl})\subseteq\mathscr{L}(\mathsf{RNFAwtl})$.

The following simple example illustrates the way in which a repetitive finite automaton with translucent letters works.

**Example 4** *Let $A_{\vee,c}=(Q,\Sigma,\lhd,\tau,I,\delta)$ be the RDFAwtl that is defined as follows:*

- $Q=\{q_0,q_1,q_2,q_3,q_4,q_5,q_6,q_7\}$ *and* $I=\{q_0\}$,
- $\Sigma=\{a,b,c\}$,
- $\tau(q_0)=\{a,b\}$, $\tau(q_1)=\emptyset$, $\tau(q_2)=\{a\}$, $\tau(q_3)=\{b\}$,
  $\tau(q_4)=\emptyset$, $\tau(q_5)=\{a\}$, $\tau(q_6)=\{a\}$, $\tau(q_7)=\{b\}$,
- $\delta(q_0,c)=q_1$, $\delta(q_0,\lhd)=q_4$,
  $\delta(q_1,a)=q_2$, $\delta(q_1,b)=q_3$, $\delta(q_1,\lhd)=\mathsf{Accept}$,
  $\delta(q_2,b)=q_1$, $\delta(q_3,a)=q_1$,
  $\delta(q_4,a)=q_5$, $\delta(q_4,b)=q_7$, $\delta(q_4,\lhd)=\mathsf{Accept}$,
  $\delta(q_5,b)=q_6$, $\delta(q_6,b)=q_4$, $\delta(q_7,a)=q_6$.

*while $\delta$ yields the empty set for all other pairs from $Q\times(\Sigma\cup\{\lhd\})$.*

*Using the graphical notation introduced in [10] for describing non-returning NFAwtls, the RDFAwtl $A_{\vee,c}$ can be depicted more compactly by the diagram in Fig. 1.*

*For example, the RDFAwtl $A_{\vee,c}$ can execute the following accepting computations:*

$$\begin{array}{llll} q_0aabbcba\cdot\lhd & \vdash_{A_{\vee,c}} q_1aabbba\cdot\lhd & \vdash_{A_{\vee,c}} q_2abbba\cdot\lhd & \vdash_{A_{\vee,c}} q_1abba\cdot\lhd\\ & \vdash_{A_{\vee,c}} q_2bba\cdot\lhd & \vdash_{A_{\vee,c}} q_1ba\cdot\lhd & \vdash_{A_{\vee,c}} q_3a\cdot\lhd\\ & \vdash_{A_{\vee,c}} q_1\cdot\lhd & \vdash_{A_{\vee,c}} \mathsf{Accept}, & \end{array}$$

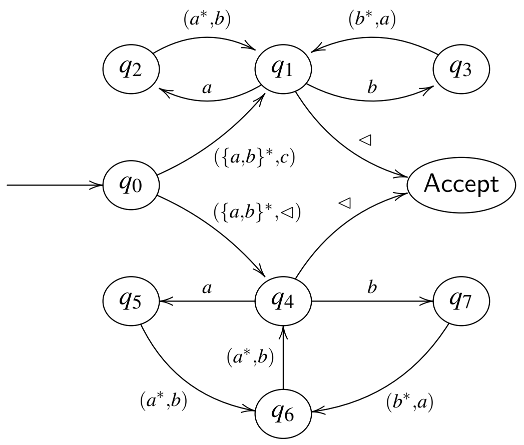


Figure 1: The diagram describing the RDFAwtl $A_{\vee,c}$. An arrow from a state $q$ to a state $q'$ labeled with a single letter $x$ means that $\tau(q) = \emptyset$ and $q' \in \delta(q,x)$. An arrow labeled with a pair $(\Delta^*,x)$ means that $\tau(q) = \Delta$ and $q' \in \delta(q,x)$.

*and*

$$\begin{array}{llll} q_0 bbaabb\cdot\lhd & \vdash_{A_{\vee,c}} q_4 bbaabb\cdot\lhd & \vdash_{A_{\vee,c}} q_7 baabb\cdot\lhd & \vdash_{A_{\vee,c}} q_6 babb\cdot\lhd \\ & \vdash_{A_{\vee,c}} q_4 abb\cdot\lhd & \vdash_{A_{\vee,c}} q_5 bb\cdot\lhd & \vdash_{A_{\vee,c}} q_6 b\cdot\lhd \\ & \vdash_{A_{\vee,c}} q_4\cdot\lhd & \vdash_{A_{\vee,c}} \mathsf{Accept}. & \end{array}$$

*In fact, it can be checked that*

$$L(A_{\vee,c}) = \{\, w \in \{a,b,c\}^* \mid |w|_c = 1 \text{ and } |w|_a = |w|_b \,\} \cup \{\, w \in \{a,b\}^* \mid 2\cdot|w|_a = |w|_b \,\}. \qquad \blacksquare$$

Finally, we come to the second extension of the automaton with translucent letters that we mentioned in the introduction.

**Definition 5** *A* non-repetitive non-returning *finite automaton with translucent letters, or an* nr-nr-NFAwtl*, is defined like an NFAwtl, but its computation relation is defined differently. Let $A = (Q,\Sigma,\lhd,\tau,I,F,\delta)$ be an nr-nr-NFAwtl. Its set of configurations is $\Sigma^*\cdot Q\cdot\Sigma^*\cdot\lhd \cup \{\mathsf{Accept},\mathsf{Reject}\}$. A configuration of the form $xqw\cdot\lhd$, where $q \in Q$ and $x,w \in \Sigma^*$, expresses the situation that $A$ is in state $q$, the tape contains the word $xw\cdot\lhd$, and the head of $A$ is on the first letter of the suffix $w\cdot\lhd$. The single-step computation relation that $A$ induces on this set of configurations is defined as follows, where $q \in Q$ and $x,w \in \Sigma^*$:*

$$xqw\cdot\lhd \vdash_A \begin{cases} xuq'v\cdot\lhd, & \text{if } w = uav,\ u \in (\tau(q))^*,\ a \in \Sigma\smallsetminus\tau(q),\ v \in \Sigma^*, \text{ and } q' \in \delta(q,a), \\ \mathsf{Reject}, & \text{if } w = uav,\ u \in (\tau(q))^*,\ a \in \Sigma\smallsetminus\tau(q),\ v \in \Sigma^*, \text{ and } \delta(q,a) = \emptyset, \\ \mathsf{Accept}, & \text{if } w \in (\tau(q))^* \text{ and } q \in F, \\ \mathsf{Reject}, & \text{if } w \in (\tau(q))^* \text{ and } q \notin F. \end{cases}$$

*Thus, in each step, A reads and deletes the first letter to the right of the current position of its head that is not translucent for the current state. In particular, after reading and deleting a letter, the head does* not *return to the left end of the tape (that is why this type of automaton is called* non-returning*), but it rather moves to the next letter. In addition, if all letters on the tape that are to the right of the current head* *position are translucent for the current state, then A halts, and it accepts if the current state is final. A word $w \in \Sigma^*$ is* accepted by *A if there exists an initial state $q_0 \in I$ and a computation* $q_0 w \cdot \lhd \vdash_A^* \mathsf{Accept}$, *where $\vdash_A^*$ denotes the reflexive transitive closure of the single-step computation relation $\vdash_A$. Now, $L(A) = \{ w \in \Sigma^* \mid w$ is accepted by $A \}$ is the* language accepted by *A, and $\mathscr{L}(\mathsf{nr\text{-}nr\text{-}NFAwtl})$ denotes the class of all languages that are accepted by nr-nr-NFAwtls.*

*An nr-nr-NFAwtl $A = (Q, \Sigma, \lhd, \tau, I, F, \delta)$ is a* non-repetitive non-returning deterministic finite automaton with translucent letters*, or an* nr-nr-DFAwtl*, if $|I| = 1$ and $|\delta(q,a)| \leq 1$ for all $q \in Q$ and all $a \in \Sigma$. Then, $\mathscr{L}(\mathsf{nr\text{-}nr\text{-}DFAwtl})$ denotes the class of all languages that are accepted by nr-nr-DFAwtls.*

The following result states that the non-repetitive non-returning finite automata with translucent letters of Def. 5 are very weak in that they just accept the regular languages.

**Theorem 6** *From an nr-nr-NFAwtl A, one can construct an NFA B such that $L(B) = L(A)$. In addition, if A is deterministic, then so is B.*

**Proof.** Let $A = (Q, \Sigma, \lhd, \tau, I, F, \delta)$ be an nr-nr-NFAwtl. From the definition of the computation relation of $A$, we see immediately that, whenever $quav \cdot \lhd \vdash_A uq'v \cdot \lhd$ is a step in an accepting computation of $A$, where $q, q' \in Q$, $u \in (\tau(q))^+$, and $a \in \Sigma$, then the prefix $u$ will not be read again during the remaining part of this accepting computation. Thus, instead of ignoring these letters, we could simply delete them. Accordingly, $A$ accepts the same language as the NFA $B = (Q, \Sigma, I, F, \delta_B)$ that is defined through the following transition relation:

$$\delta_B(q,a) = \begin{cases} q, & \text{if } a \in \tau(q), \\ \delta(q,a), & \text{if } a \notin \tau(q). \end{cases}$$

Trivially, if $A$ is deterministic, the constructed automaton $B$ is deterministic, too. □

It was actually this observation that led us to define the nr-NFAwtl and the nr-DFAwtl in [9].

**Definition 7 ([9])** *An* nr-NFAwtl *$A = (Q, \Sigma, \lhd, \tau, I, \delta)$ is defined like an RNFAwtl but with the additional extension that it is non-returning, that is, it behaves like an nr-nr-NFAwtl, but for some states $q \in Q$, $\delta(q, \lhd)$ may be a subset of Q. Thus, if A is in a configuration of the form $xqw \cdot \lhd$, where $q \in Q$ satisfying $\delta(q, \lhd) \subseteq Q$, $x \in \Sigma^*$, and $w \in (\tau(q))^*$, then $xqw \cdot \lhd \vdash_A q'xw \cdot \lhd$ for each state $q' \in \delta(q, \lhd)$. If $|I| = 1$ and $|\delta(q,a)| \leq 1$ for all $q \in Q$ and all $a \in \Sigma \cup \{\lhd\}$, then A is an* nr-DFAwtl*. We use $\mathscr{L}(\mathsf{nr\text{-}NFAwtl})$ and $\mathscr{L}(\mathsf{nr\text{-}DFAwtl})$ to denote the corresponding classes of languages.*

Thus, an nr-NFAwtl is both at the same time, non-returning in the sense of Def. 5 and repetitive in the sense of Def. 2. In particular, the nr-NFAwtl (nr-DFAwtl) should not be confused with the nr-nr-NFAwtl (nr-nr-DFAwtl) of Def. 5. The latter has been introduced here only to complete the picture. Theorem 6 above shows that they are not really interesting types of automata with translucent letters. Finally, we should also mention another related type of automaton, the right one-way jumping finite automaton.

**Definition 8 ([1, 3])** *A* nondeterministic right one-way jumping finite automaton*, or an* NROWJFA*, is given through a 6-tuple $J = (Q, \Sigma, \lhd, I, F, \delta)$, where $Q$, $\Sigma$, $\lhd$, $I$, and $F$ are defined as for an NFAwtl, and $\delta : Q \times \Sigma \to \mathscr{P}(Q)$ is a transition relation. For each state $q \in Q$, let $\Sigma_q = \{ a \in \Sigma \mid \delta(q,a) \neq \emptyset \}$ be the set of letters that J can read in state q.*

*A configuration of the NROWJFA J is a word $qw \cdot \lhd$ from the set $Q \cdot \Sigma^* \cdot \lhd$. The* computation relation *$\circlearrowright_J^*$ that J induces on its set of configurations is the reflexive and transitive closure of the* right one-way jumping relation *$\circlearrowright_J$ that is defined as follows, where $q, q' \in Q$, $x, y \in \Sigma^*$, and $a \in \Sigma$ :*

$$qxay \cdot \lhd \circlearrowright_J q'yx \cdot \lhd \text{ if } x \in (\Sigma \smallsetminus \Sigma_q)^* \text{ and } q' \in \delta(q,a).$$

*Thus, being in state q, J reads and deletes the first letter to the right of the actual head position that it can actually read in that state, while the prefix that consists of letters for which J has no transitions in the current state is cyclically shifted to the end of the current tape inscription. Then,*

$$L(J) = \{\, w \in \Sigma^* \mid \exists q_0 \in I\, \exists q_f \in F : q_0 w \cdot \lhd \circlearrowright_J^* q_f \cdot \lhd \,\}$$

*is the language accepted by the NROWJFA J.*

*The NROWJFA J is* deterministic*, that is, a* right one-way jumping finite automaton *or an* ROWJFA*, if $|I| = 1$ and $|\delta(q,a)| \leq 1$ for all $q \in Q$ and $a \in \Sigma$.*

Actually, as defined in [1, 3], the (N)ROWJFA does not have an end-of-tape marker, but it is obvious that our definition is equivalent to the original one. We just introduced this end-of-tape marker to ensure consistency with our other types of automata. We see that the (N)ROWJFA overcomes the problem of processing letters that are skipped over by cyclically shifting these letters to the right so that they can be read later. The following results are known concerning the various types of automata introduced above.

**Theorem 9 ([10])** (a) $\mathsf{REG} \subsetneq \mathscr{L}(\mathsf{DFAwtl}) \subsetneq \mathscr{L}(\mathsf{nr\text{-}DFAwtl}) \subsetneq \mathscr{L}(\mathsf{nr\text{-}NFAwtl})$.
(b) $\mathsf{REG} \subsetneq \mathscr{L}(\mathsf{DFAwtl}) \subsetneq \mathscr{L}(\mathsf{NFAwtl}) \subsetneq \mathscr{L}(\mathsf{nr\text{-}NFAwtl})$.
(c) $\mathscr{L}(\mathsf{ROWJFA}) \subsetneq \mathscr{L}(\mathsf{nr\text{-}DFAwtl})$.
(d) $\mathscr{L}(\mathsf{NROWJFA}) \subsetneq \mathscr{L}(\mathsf{nr\text{-}NFAwtl})$.
(e) $\mathscr{L}(\mathsf{ROWJFA}) \subsetneq \mathscr{L}(\mathsf{NROWJFA})$.

In addition, $\mathscr{L}(\mathsf{ROWJFA})$ is incomparable under inclusion to $\mathscr{L}(\mathsf{DFAwtl})$ and $\mathscr{L}(\mathsf{NFAwtl})$, and $\mathscr{L}(\mathsf{NROWJFA})$ is incomparable to $\mathscr{L}(\mathsf{DFAwtl})$, $\mathscr{L}(\mathsf{NFAwtl})$, and $\mathscr{L}(\mathsf{nr\text{-}DFAwtl})$. Thus, it remains to compare the deterministic and nondeterministic repetitive finite automata with translucent letters to these other types of automata.

## 3 Comparing the Repetitive Automata to the Non-Repetitive Ones

We claim that the language $L_{\vee,c} = L(A_{\vee,c})$ of Example 4 is not accepted by any DFAwtl. As each DFAwtl can be simulated by an RDFAwtl, this shows that $\mathscr{L}(\mathsf{DFAwtl}) \subsetneq \mathscr{L}(\mathsf{RDFAwtl})$.

**Lemma 10** $L_{\vee,c} \notin \mathscr{L}(\mathsf{DFAwtl})$.

**Proof.** Assume that $A = (Q, \Sigma, \lhd, \tau, I, F, \delta)$ is a DFAwtl that accepts the language $L_{\vee,c}$, where $Q = \{q_0, q_1, \ldots, q_{m-1}\}$, $\Sigma = \{a,b,c\}$, and $I = \{q_0\}$.

Let $n > 2m$, and let $w = a^n b^n c \in L_{\vee,c}$. Then the computation of $A$ on input $w$ is accepting, that is, it is of the form

$$q_0 a^n b^n c \cdot \lhd \vdash_A q_{i_1} w_1 \cdot \lhd \vdash_A \cdots \vdash_A q_{i_r} w_r \cdot \lhd \vdash_A \mathsf{Accept},$$

where $w_r \in (\tau(q_{i_r}))^*$ and $q_{i_r} \in F$. If $|w_r|_a > 0$, then $A$ would also accept on input $a^{n+1}b^n c \notin L_{\vee,c}$, if $|w_r|_b > 0$, then $A$ would also accept on input $a^n b^{n+1} c \notin L_{\vee,c}$, and if $|w_r|_c > 0$, then $A$ would also accept on input $a^n b^n \notin L_{\vee,c}$. Hence, it follows that $w_r = \lambda$, that is, the accepting computation above consists of $2n+1$ transition steps, each of which deletes a letter, and the final accepting step. In particular, the only occurrence of the letter $c$ is read and deleted during the above computation, that is, there exist an index $j$ and integers $r,s \geq 0$ such that $r+s=j$ and

$$q_0 a^n b^n c \cdot \lhd \vdash_A^j q_{i_j} a^{n-r} b^{n-s} c \cdot \lhd \vdash_A q_{i_{j+1}} a^{n-r} b^{n-s} \cdot \lhd \vdash_A^* q_{i_r} \cdot \lhd \vdash_A \mathsf{Accept}.$$

We now distinguish several cases.

(1) Assume that $a, b \in \tau(q_{i_j})$. Then $A$ executes the following computation on input $a^n b^{2n}$:

$$q_0 a^n b^{2n} \cdot \lhd \vdash_A^j q_{i_j} a^{n-r} b^{2n-s} \cdot \lhd \vdash_A \begin{cases} \mathsf{Accept}, & \text{if } q_{i_j} \in F, \\ \mathsf{Reject}, & \text{if } q_{i_j} \notin F. \end{cases}$$

As $a^n b^{2n} \in L_{\vee,c}$, we see that the latter computation must be accepting, that is, $q_{i_j} \in F$. Thus, $A$ can also execute the following accepting computation:

$$q_0 a^{r+s+1} b^{r+s+1} \cdot \lhd \vdash_A^j q_{i_j} a^{s+1} b^{r+1} \cdot \lhd \vdash_A \mathsf{Accept},$$

which, however, contradicts the fact that $a^{r+s+1} b^{r+s+1} \notin L_{\vee,c}$.

(2) Assume that $a \notin \tau(q_{i_j})$, but $b \in \tau(q_{i_j})$. Then, $r = n$ and $s \leq n$, that is, $a^{n-r} b^{n-s} c = b^{n-s} c$. Now, $A$ executes the following computation on input $a^n b^{2n}$:

$$q_0 a^n b^{2n} \cdot \lhd \vdash_A^j q_{i_j} a^{n-r} b^{2n-s} \cdot \lhd = q_{i_j} b^{2n-s} \cdot \lhd \vdash_A \begin{cases} \mathsf{Accept}, & \text{if } q_{i_j} \in F, \\ \mathsf{Reject}, & \text{if } q_{i_j} \notin F. \end{cases}$$

As $a^n b^{2n} \in L_{\vee,c}$, we see that the latter computation must be accepting, that is, $q_{i_j} \in F$. Thus, $A$ can also execute the following accepting computation:

$$q_0 a^n b^{3n+s} \cdot \lhd \vdash_A^j q_{i_j} b^{3n} \cdot \lhd \vdash_A \mathsf{Accept},$$

which, however, contradicts the fact that $a^n b^{3n+s} \notin L_{\vee,c}$.

(3) Assume that $b \notin \tau(q_{i_j})$. Then $r \leq n$ and $s = n$, that is, $a^{n-r} b^{n-s} c = a^{n-r} c$. As $n > m$, there exist a state $q$ and integers $k_0, k_1, t_0 \geq 0$ and $t_1 \geq 1$ such that the accepting computation above has the form

$$q_0 a^n b^n c \cdot \lhd \vdash_A^* q a^{n-k_0} b^{n-t_0} c \cdot \lhd \vdash_A^+ q a^{n-k_0-k_1} b^{n-t_0-t_1} c \cdot \lhd \vdash_A^* q_{i_j} a^{n-r} c \cdot \lhd \vdash_A^* \mathsf{Accept}.$$

Hence, we also obtain the following accepting computation:

$$q_0 a^{n+k_1} b^{n+t_1} c \cdot \lhd \vdash_A^* q a^{n+k_1-k_0} b^{n+t_1-t_0} c \cdot \lhd \vdash_A^+ q a^{n-k_0} b^{n-t_0} c \cdot \lhd \vdash_A^* \mathsf{Accept}.$$

This implies that $a^{n+k_1} b^{n+t_1} c \in L_{\vee,c}$, which yields $k_1 = t_1$.

Now we consider the computations of $A$ on input $a^{n+\nu \cdot t_1} b^{n+\nu \cdot t_1}$ for all $\nu \geq 0$:

$$q_0 a^{n+\nu \cdot t_1} b^{n+\nu \cdot t_1} \cdot \lhd \vdash_A^* q_{i_j} a^{n-r} \cdot \lhd.$$

As $a^{n+\nu \cdot t_1} b^{2n+2\nu \cdot t_1} \in L_{\vee,c}$, we see that the computation of $A$ that begins with the configuration $q_{i_j} a^{n-r} b^{n+\nu \cdot t_1}$ leads to acceptance for all $\nu \geq 0$. Hence, we obtain

$$q_0 a^n b^n b^{n+t_1} \cdot \lhd \vdash_A^* q_{i_j} a^{n-r} b^{n+t_1} \cdot \lhd \vdash_A^* \mathsf{Accept},$$

but we have $a^n b^n b^{n+t_1} \notin L_{\vee,c}$, as $t_1 > 0$, a contradiction.

As this covers all cases, we see that the language $L_{\vee,c}$ is indeed not accepted by any DFAwtl. $\square$

On the other hand, it can be shown quite easily that the RDFAwtl (the RNFAwtl) is a special case of the nr-DFAwtl (the nr-NFAwtl).

**Lemma 11** *From an RNFAwtl $A$, one can construct an nr-NFAwtl $B$ such that $L(B) = L(A)$. In addition, if $A$ is deterministic, then so is $B$.*

**Proof.** Let $A = (Q, \Sigma, \lhd, \tau, I, \delta)$ be an RNFAwtl. We define an nr-NFAwtl $B = (Q_B, \Sigma, \lhd, \tau_B, I_B, \delta_B)$ that simulates the computations of $A$ as follows:

- $Q_B = Q \cup \{\, q' \mid q \in Q \,\}$, where for each state $q \in Q$, $q'$ is an additional auxiliary state, and $I_B = I$,
- for each state $q \in Q$, $\tau_B(q) = \tau(q)$ and $\tau_B(q') = \Sigma$,
- for each state $q \in Q$ and each letter $a \in \Sigma$, $\delta_B(q,a) = \{\, p' \mid p \in \delta(q,a) \,\}$ and $\delta_B(q',a) = \emptyset$.
- Furthermore, for each state $q \in Q$, $\delta_B(q,\lhd) = \mathsf{Accept}$, if $\delta(q,\lhd) = \mathsf{Accept}$, and $\delta_B(q,\lhd) = \emptyset$, otherwise. Finally, $\delta_B(q',\lhd) = \{q\}$.

It remains to verify that $B$ just simulates the computations of $A$.

Assume that $qw \cdot \lhd$ is a configuration of $A$, that is, $q \in Q$ and $w \in \Sigma^*$. From the definition of the computation relation $\vdash_A$, we see that there are four different cases that we must consider.

First, if $w = uav$ for some words $u \in (\tau(q))^*$, $v \in \Sigma^*$, and a letter $a \in (\Sigma \smallsetminus \tau(q))$, then $A$ executes a transition from $\delta(q,a)$.

- If $p \in \delta(q,a)$, then $qw \cdot \lhd = quav \cdot \lhd \vdash_A puv \cdot \lhd$ is a possible step of $A$. In this case, $B$ can execute the following sequence of steps:

$$qw \cdot \lhd = quav \cdot \lhd \vdash_B up'v \cdot \lhd \vdash_B puv \cdot \lhd.$$

- On the other hand, if $\delta(q,a) = \emptyset$, then $A$ halts and rejects. However, in this case, also $\delta_B(q,a) = \emptyset$, and hence, $B$ halts and rejects as well.

Finally, if $w \in (\tau(q))^*$, then $A$ halts.

- If $\delta(q,\lhd) = \mathsf{Accept}$, then $\delta_B(q,\lhd) = \mathsf{Accept}$, too.
- If $\delta(q,\lhd) = \emptyset$, then $A$ rejects. In this case, $\delta_B(q,\lhd) = \emptyset$, that is, $B$ halts and rejects as well.

It follows that $L(A) \subseteq L(B)$.

Conversely, if $w \in L(B)$, then it is easily verified that each accepting computation of $B$ on input $w$ is just a simulation of an accepting computation of $A$ on input $w$. Thus, we see that $L(B) = L(A)$.

Finally, the above definition of $B$ shows that $B$ is deterministic, if $A$ is. □

The language

$$L_\vee = \{\, w \in \{a,b\}^* \mid |w|_b = |w|_a \text{ or } |w|_b = 2 \cdot |w|_a \,\}$$

is a rational trace language, and as such, it is accepted by an NFAwtl. However, as proved in [10], this language is not accepted by any nr-DFAwtl. Hence, $L_\vee$ is not accepted by any RDFAwtl, either. Thus, we immediately obtain the following non-inclusion result.

**Corollary 12** $\mathscr{L}(\mathsf{NFAwtl}) \nsubseteq \mathscr{L}(\mathsf{RDFAwtl})$.

As defined above, an RNFAwtl $A = (Q, \Sigma, \lhd, \tau, I, \delta)$ may run into an infinite computation. Just assume that $q$ is a state of $A$, $w \in (\tau(q))^*$, and $q \in \delta(q,\lhd)$. Then $qw \cdot \lhd \vdash_A qw \cdot \lhd \vdash_A qw \cdot \lhd$, and so forth. However, we can avoid this by converting $A$ into an equivalent RNFAwtl $B$ as follows.

Let $B = (Q', \Sigma, \lhd, \tau', I', \delta')$, where $Q' = \{\, (q,S) \mid q \in Q \text{ and } S \subseteq Q \,\}$, $I' = \{\, (q,\emptyset) \mid q \in I \,\}$, $\tau'(q,S) = \tau(q)$ for all $q \in Q$ and all $S \subseteq Q$,

$$\delta'((q,S),a) = \{\, (p,\emptyset) \mid p \in \delta(q,a) \,\} \text{ for all } q \in Q,\ S \subseteq Q, \text{ and all } a \in \Sigma,$$

and

$$\delta'((q,S),\lhd) = \{\,(p,S\cup\{q\}) \mid p\in\delta(q,\lhd) \text{ and } q\notin S\,\} \text{ for all } q\in Q \text{ and all } S\subseteq Q.$$

Finally, take $\delta'((q,S),\lhd) = \mathsf{Accept}$ if $\delta(q,\lhd) = \mathsf{Accept}$. The set $S$ is used to record those states in which the end-of-tape marker has been reached, and the computation has continued. In the next cycle, when a non-translucent letter is read, then this set is emptied, otherwise, the next state is added to it. This process continues until either a letter is read and deleted, or until no new state can be added to the current set $S$, in which case the computation fails.

Moreover, an RNFAwtl $A = (Q,\Sigma,\lhd,\tau,I,\delta)$ may accept without having read and deleted its input completely. However, we can easily extend the RNFAwtl $A$ into an equivalent RNFAwtl $C$ that always reads and deletes its input completely before it accepts. Just take $C = (Q\cup\{q_e\},\Sigma,\lhd,\tau',I,\delta')$, where $q_e$ is a new state, $\tau'(q) = \tau(q)$ for all $q\in Q$ and $\tau'(q_e) = \emptyset$, and $\delta'$ is defined as follows:

$$\begin{array}{lcll}
- \;\delta'(q,a) & = & \delta(q,a) & \text{for all } q\in Q \text{ and all } a\in\Sigma,\\
- \;\delta'(q,\lhd) & = & \left\{\begin{array}{ll}\delta(q,\lhd), & \text{if } \delta(q,\lhd)\neq \mathsf{Accept},\\ \{q_e\}, & \text{if } \delta(q,\lhd) = \mathsf{Accept},\end{array}\right. & \\
- \;\delta'(q_e,a) & = & \{q_e\} & \text{for all } a\in\Sigma,\\
- \;\delta'(q_e,\lhd) & = & \mathsf{Accept}. &
\end{array}$$

Given a word $w\in\Sigma^*$ as input, the RNFAwtl $C$ will execute exactly the same steps as the RNFAwtl $A$ until $A$ accepts. Now, the accept step of $A$ is simulated by $C$ through changing into state $q_e$. As $\tau'(q_e) = \emptyset$ and as $\delta'(q_e,a) = \{q_e\}$ for all $a\in\Sigma$, $C$ will now read and delete the remaining tape contents and accept on reaching the end-of-tape marker $\lhd$. It follows easily that $L(C) = L(A)$. Together, the two constructions above yield the following technical result.

**Lemma 13** *Each RNFAwtl A can effectively be converted into an equivalent RNFAwtl C that never gets into an infinite computation and that accepts only after reading and deleting its input completely. In addition, if A is deterministic, then so is C.*

Finally, we are ready to establish the following equality, which will be proved by simulation.

**Theorem 14** $\mathscr{L}(\mathsf{RNFAwtl}) = \mathscr{L}(\mathsf{NFAwtl})$.

**Proof.** From Proposition 3, we know already that $\mathscr{L}(\mathsf{NFAwtl}) \subseteq \mathscr{L}(\mathsf{RNFAwtl})$ holds. Thus, it remains to prove the converse inclusion. Accordingly, we show how to simulate an RNFAwtl by an NFAwtl.

Let $A = (Q,\Sigma,\lhd,\tau,I,\delta)$ be an RNFAwtl. By Lemma 13, we can assume that $A$ never gets into an infinite computation and that it accepts only after reading and deleting its input completely. We now construct an NFAwtl $B = (Q_B,\Sigma,\lhd,\tau_B,I_B,F_B,\delta_B)$ with the set of states $Q_B = \{\,(q,\Gamma) \mid q\in Q \text{ and } \Gamma\subseteq\Sigma\,\}$. The automaton $B$ uses the second component of its states to keep track of the set of letters that may still occur on its tape. At the beginning of its computation, $\Gamma = \Sigma$. When $B$ simulates a step $qw\cdot\lhd \vdash_A q'w\cdot\lhd$ in which $A$ changes its state at the right sentinel because all symbols on its tape are translucent for the state $q$, that is, $w\in(\tau(q))^*$, then the second component will be restricted to $\Gamma\cap\tau(q)$. The NFAwtl $B$ is defined as follows:

- $I_B = \{\,(q,\Sigma) \mid q\in I\,\}$, and
- $F_B = \{\,(q,\Gamma) \mid \delta(q,\lhd) = \mathsf{Accept} \text{ and } \Gamma\subseteq\Sigma\,\}$.
- The translucency relation $\tau_B$ is defined through $\tau_B((q,\Gamma)) = \tau(q)\cap\Gamma$ for all $q\in Q$ and all $\Gamma\subseteq\Sigma$, and

- the transition relation $\delta_B$ is initialized as follows, where $q \in Q$, $\Gamma \subseteq \Sigma$, and $a \in \Sigma$:

$$\delta_B((q,\Gamma),a) = \begin{cases} \{ (q',\Gamma) \mid q' \in \delta(q,a) \}, & \text{if } a \in \Gamma \text{ and } \delta(q,a) \neq \emptyset, \\ \emptyset, & \text{if } a \in \Sigma \smallsetminus \Gamma \text{ or } \delta(q,a) = \emptyset. \end{cases}$$

It remains to add further transitions to $\delta_B$ that are to simulate the transitions of the form $q' \in \delta(q,\lhd)$ of $A$. Assume that $q' \in \delta(q,\lhd)$. This transition can be applied by $A$ to a configuration of the form $qw \cdot \lhd$ for which $w \in (\tau(q))^*$, and it yields the configuration $q'w \cdot \lhd$. Thus, $A$ simply executes a change of state, but after that, it 'knows' that the word $w$ only contains occurrences of letters from the set $\tau(q)$. Accordingly, for each state $p \in Q$ and each letter $a \in \Sigma$, if $q \in \delta(p,a)$, then we add the state $(q',\Gamma \cap \tau(q))$ to $\delta_B((p,\Gamma),a)$ for each subset $\Gamma$ containing the letter $a$. This transition allows $B$ to simulate the sequence of two transitions $puav \cdot \lhd \vdash_A quv \cdot \lhd \vdash q'uv \cdot \lhd$, where $u \in (\tau(p))^*$ and $u,v \in (\tau(q))^*$, by the single transition $(p,\Gamma)uav \cdot \lhd \vdash_B (q',\Gamma \cap \tau(q))uv \cdot \lhd$. In addition, if $q \in I$, then the state $(q',\tau(q))$ is added to the set $I_B$, as $A$ may start a computation by executing the step $qw \cdot \lhd \vdash_A q'w \cdot \lhd$, if $w \in (\tau(q))^*$.

It can now be checked that $B$ just simulates the computations of $A$. If during such a simulation, $B$ is in a state $(q,\Gamma)$ but an occurrence of a letter $c \notin \Gamma$ is encountered, then $B$ gets stuck and, so, rejects, as in that situation, the letter $c$ is neither translucent for the state $(q,\Gamma)$ nor is the transition $\delta_B((q,\Gamma),c)$ defined. It follows that $L(B) = L(A)$, which completes the proof. $\square$

It remains to compare the RDFAwtl to the nr-DFAwtl, the nr-NFAwtl, and the (N)ROWJFA. The deterministic linear language $L_2 = \{ a^n b^n \mid n \geq 0 \}$ is accepted by an nr-DFAwtl [9]. However, it is not accepted by any NFAwtl [13]. Hence, we get the following result from Lemma 11.

**Corollary 15** $\mathscr{L}(\mathsf{RDFAwtl}) \subsetneq \mathscr{L}(\mathsf{nr\text{-}DFAwtl})$.

According to [10], the language classes $\mathscr{L}(\mathsf{DFAwtl})$ and $\mathscr{L}(\mathsf{NFAwtl})$ are incomparable under inclusion to the classes $\mathscr{L}(\mathsf{ROWJFA})$ and $\mathscr{L}(\mathsf{NROWJFA})$. Accordingly, we also have the following incomparability result.

**Corollary 16** *The language class* $\mathscr{L}(\mathsf{RDFAwtl})$ *is incomparable under inclusion to the classes* $\mathscr{L}(\mathsf{ROWJFA})$ *and* $\mathscr{L}(\mathsf{NROWJFA})$.

The diagram in Figure 2 summarizes the inclusion and incomparability results obtained for the various types of automata with translucent letters. All arrows in that diagram denote proper inclusions, and classes that are not connected by a sequence of arrows are incomparable under inclusion. Here, LRAT denotes the class of rational trace languages (see, e.g., [15]), GCSL is the class of growing context-sensitive languages (see, e.g., [4]), and (D)LIN is the class of (deterministic) linear context-free languages.

## 4 Closure Properties

In this section, we study closure properties of the classes of languages accepted by deterministic and nondeterministic repetitive finite automata with translucent letters. As the class $\mathscr{L}(\mathsf{RNFAwtl})$ of languages accepted by RNFAwtls coincides with the class $\mathscr{L}(\mathsf{NFAwtl})$, based on [15], we obtain immediately that $\mathscr{L}(\mathsf{RNFAwtl})$ is closed under union, product, Kleene star, inverse projections, disjoint shuffle, and the operation of taking the commutative closure, but it is neither closed under intersection (with regular sets), nor under complementation, nor under non-erasing morphisms.

On the other hand, for the class $\mathscr{L}(\mathsf{DFAwtl})$, it is known [16] that it is closed under complementation, but it is not closed under any of the following operations: union, intersection (with regular sets), product, Kleene star, reversal, alphabetic morphisms, and commutative closure.

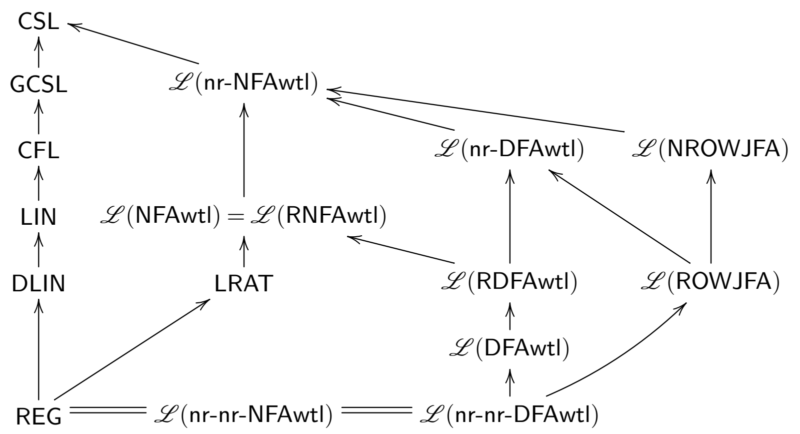


Figure 2: The inclusion relations between the various types of finite automata with translucent letters.

Here, the commutative closure of a language $L$ is based on the letter-equivalence of words. We say that two words $u$ and $v$ over an alphabet $\Sigma$ are *letter-equivalent* if, for each letter $a \in \Sigma$, $|u|_a = |v|_a$. The *commutative closure* $\mathrm{com}(L)$ of a language $L \subseteq \Sigma^*$ is the set of all words that are letter-equivalent to a word from $L$, that is, $\mathrm{com}(L) = \{\, w \in \Sigma^* \mid \exists u \in L : u \text{ is letter-equivalent to } w \,\}$.

In addition, we are interested in the shuffle operation. For two words $u$ and $v$ from $\Sigma^*$, the *shuffle* $u \sqcup\!\sqcup v$ of $u$ and $v$ is the set of words

$$u \sqcup\!\sqcup v = \{\, u_1v_1u_2v_2\cdots u_mv_m \mid m \geq 1, \forall i = 1,2,\ldots,m : u_i, v_i \in \Sigma^*, u = u_1u_2\cdots u_m, \text{ and } v = v_1v_2\cdots v_m \,\},$$

and the *shuffle* of two languages $L_1, L_2 \subseteq \Sigma^*$ is the language $L_1 \sqcup\!\sqcup L_2 = \bigcup_{u\in L_1, v\in L_2}(u \sqcup\!\sqcup v)$.

We shall show that $\mathscr{L}$(RDFAwtl) is closed under complementation, left quotient with respect to a single word, and disjoint shuffle. However, it is not closed under union, intersection (with regular sets), product, Kleene star, reversal, alphabetic morphism, commutative closure, and shuffle.

To begin with, observe that the languages

$$L_= = \{\, w \in \{a,b\}^* \mid |w|_a = |w|_b \,\} \text{ and } L_{2=} = \{\, w \in \{a,b\}^* \mid 2\cdot|w|_a = |w|_b \,\}$$

are both accepted by DFAwtls. However, their union is the language $L_\vee$ that is not even accepted by any nr-DFAwtl. In addition, if $R$ denotes the regular language that is defined through the regular expression $(ab)^* + (abb)^*$, then $\mathrm{com}(R) = L_\vee$. Moreover, let $L'_{2=} = \{\, w \in \{c,d\}^* \mid 2\cdot|w|_c = |w|_d \,\}$. Then, it is easily verified that the language $L_= \cup L'_{2=}$ is also accepted by a DFAwtl. However, if $\varphi : \{a,b,c,d\}^* \to \{a,b\}^*$ denotes the alphabetic morphism that is defined through $a \mapsto a$, $b \mapsto b$, $c \mapsto a$, and $d \mapsto b$, then $\varphi(L_= \cup L'_{2=}) = L_\vee$. Finally, the language $L_2 = \{\, a^nb^n \mid n \geq 0 \,\} = L_= \cap (\{a\}^* \cdot \{b\}^*)$ is not accepted by any NFAwtl [13]. In summary, these examples yield the following non-closure properties.

**Corollary 17** *The language class* $\mathscr{L}$(RDFAwtl) *is not closed under union, intersection (with regular sets), alphabetic morphisms, or commutative closure.*

Next, we prove that the class $\mathscr{L}$(RDFAwtl) is closed under the operation of complementation.

**Proposition 18** $\mathscr{L}$(RDFAwtl) *is closed under complementation.*

**Proof.** Let $A = (Q_A, \Sigma, \lhd, \tau_A, q_0, \delta_A)$ be an RDFAwtl. To obtain an RDFAwtl $B = (Q_B, \Sigma, \lhd, \tau_B, q_0, \delta_B)$ accepting the complement of $L(A)$, we need to change all accepting steps into rejecting ones and all rejecting steps into accepting ones. For that matter, we extend the set of states of $A$ with an additional state $q_a$ that will always lead to acceptance, that is, $Q_B = Q_A \cup \{q_a\}$. The translucency mapping $\tau_B(q)$ equals $\tau_A(q)$ for all states $q \in Q_A$, and $\tau(q_a) = \emptyset$. Finally, we define the transition relation of $B$ as follows, where $q \in Q_A$ and $x \in \Sigma$:

$$\begin{array}{lcl}
\delta_B(q,x) & = & \begin{cases} \delta_A(q,x), & \text{if } \delta_A(q,x) \neq \emptyset, \\ q_a, & \text{if } \delta_A(q,x) = \emptyset \text{ and } x \notin \tau(q), \end{cases} \\
\delta_B(q,\lhd) & = & \begin{cases} \delta_A(q,\lhd), & \text{if } \delta_A(q,\lhd) \in Q_A, \\ \emptyset, & \text{if } \delta_A(q,\lhd) = \mathsf{Accept}, \\ q_a, & \text{if } \delta_A(q,\lhd) = \emptyset, \end{cases} \\
\delta_B(q_a,x) & = & q_a, \\
\delta_B(q_a,\lhd) & = & \mathsf{Accept}.
\end{array}$$

Now, whenever a computation of the automaton $A$ on an input word $w \in \Sigma^*$ is accepting, it is of the following form:

$$q_0 w \cdot \lhd \vdash_A^* q w' \cdot \lhd \vdash_A \mathsf{Accept}, \text{ where } q \in Q_A, w' \in (\tau(q))^*, \text{ and } \delta_A(q,\lhd) = \mathsf{Accept}.$$

Then, the automaton $B$ will execute the computation $q_0 w \cdot \lhd \vdash_B^* q w' \cdot \lhd \vdash_B \mathsf{Reject}$. Whenever a computation of $A$ on an input word $w \in \Sigma^*$ is rejecting, then

$$q_0 w \cdot \lhd \vdash_A^* q w' \cdot \lhd \vdash_A \mathsf{Reject}, \text{ where } q \in Q_A \text{ and } w' \in \Sigma^*.$$

Here either

1. $w' = uxv$ for some $u \in (\tau(q))^*, v \in \Sigma^*, x \in \Sigma \smallsetminus \tau(q)$ such that $\delta_A(q,x) = \emptyset$, or
2. $w' \in (\tau(q))^*$ and $\delta_A(q,\lhd) = \emptyset$.

In both cases, the automaton $B$ will execute an accepting computation of the form

$$q_0 w \cdot \lhd \vdash_B^* q w' \cdot \lhd \vdash_B q_a w' \cdot \lhd \vdash_B^* \mathsf{Accept}.$$

Thus, we see that $L(B) = \Sigma^* \smallsetminus L(A)$, the complement of the language $L(A)$. □

The next lemma states that an RDFAwtl can be assumed to always read the very first letter during the first step of each accepting computation. This is a purely technical result that will be useful below.

**Lemma 19** *From an RDFAwtl A, one can effectively construct an equivalent RDFAwtl B such that, in the first step of each computation, B reads the very first letter of the given input.*

**Proof.** Let $A = (Q_A, \Sigma, \lhd, \tau_A, q_0, \delta_A)$ be an RDFAwtl, and let $L$ be the language accepted by $A$. By Lemma 13, we can assume that $A$ never gets into an infinite computation and that it reads and deletes its input completely in each accepting computation. In fact, we can even assume that $A$ reads and deletes its input completely in each and every computation, that is, even in all its rejecting computations (see, e.g., the proof of Prop. 18). Moreover, we may assume without loss of generality that the initial state $q_0$ is not entered by any transition, that is, this state can only occur in initial configurations of $A$.

We now describe the RDFAwtl $B = (Q_B, \Sigma, \lhd, \tau_B, q_0, \delta_B)$ through the following definition:

– $Q_B = Q_A \cup \{(q,a) \mid q \in Q_A \text{ and } a \in \Sigma \text{ such that } a \in \tau_A(q)\}$,

– $\tau_B(p) = \begin{cases} \emptyset, & \text{if } p = q_0, \\ \tau_A(p), & \text{if } p \in Q_A \smallsetminus \{q_0\}, \\ \tau_A(q), & \text{if } p = (q,a) \text{ for some } q \in Q_A \text{ and } a \in \Sigma, \end{cases}$

– and the transition function $\delta_B$ is defined as follows, where $q \in Q_A \smallsetminus \{q_0\}$ and $a \in \Sigma$ :

$$\delta_B(q_0,a) = \begin{cases} \delta_A(q_0,a), & \text{if } a \notin \tau_A(q_0), \\ (q_0,a), & \text{if } a \in \tau_A(q_0), \end{cases}$$

$$\delta_B(q_0,\lhd) = \delta_A(q_0,\lhd),$$

$$\delta_B(q,b) = \delta_A(q,b) \text{ for all } q \in Q_A \smallsetminus \{q_0\} \text{ and } b \in \Sigma \cup \{\lhd\},$$

$$\delta_B((q,a),b) = \begin{cases} (\delta_A(q,b),a), & \text{if } b \neq a \text{ and } a \in \tau_A(\delta_A(q,b)), \\ \delta_A(\delta_A(q,b),a), & \text{if } b \neq a \text{ and } a \notin \tau_A(\delta_A(q,b)), \\ \emptyset, & \text{if } b = a, \end{cases}$$

$$\delta_B((q,a),\lhd) = \begin{cases} (\delta_A(q,\lhd),a), & \text{if } a \in \tau_A(\delta_A(q,\lhd)), \\ \delta_A(\delta_A(q,\lhd),a), & \text{if } a \notin \tau_A(\delta_A(q,\lhd)). \end{cases}$$

The states of the form $(q,a)$ are used to encode the fact that $A$ is in state $q$ and that the first letter on the tape was an $a$, which, however, has not yet been read by $A$ (but was already read by $B$). Hence, by our above assumptions on the computations of $A$, we see that, if $B$ is in state $(q,a)$ reading the sentinel $\lhd$, then $\delta_A(q,\lhd) \in Q$ holds.

From the above definition, it follows immediately that, in each computation, the RDFAwtl $B$ reads and deletes the first letter on the tape. Moreover, the initial state $q_0$, which is not entered by any transition of $A$, is not entered by any transition of $B$, either. Now, by comparing the computations of $B$ on a given word $w$ to that of $A$ on $w$, it can be verified that $B$ is equivalent to $A$, that is, that $L(B) = L(A)$ holds. □

For a language $L \subseteq \Sigma^*$ and a word $w \in \Sigma^*$, the *left quotient* of $L$ with respect to $w$ is the language

$$w \diagdown L = \{z \in \Sigma^* \mid wz \in L\}.$$

If $L$ is accepted by an RDFAwtl $A = (Q,\Sigma,\lhd,\tau,q_0,\delta)$, then by Lemma 19, we can assume that $A$ always reads the first letter of the given input during the first step of each computation. Hence, for each letter $a \in \Sigma$, the RDFAwtl $B_a = (Q,\Sigma,\lhd,\tau,\delta(q_0,a),\delta)$ accepts the language $a \diagdown L$, which shows that $a \diagdown L$ is accepted by an RDFAwtl. By induction on $|w|$, it now follows that $w \diagdown L$ is accepted by an RDFAwtl for each word $w$.

**Corollary 20** *The language class $\mathscr{L}$(RDFAwtl) is closed under the operation of taking the left quotient with respect to a single word.*

To derive the other non-closure properties stated above, we need the following technical results.

**Lemma 21** *None of the following languages is accepted by an RDFAwtl:*

(a) $L_c = \{wc \mid w \in \{a,b\}^*, |w|_a \geq |w|_b\}$,
(b) $(L_c^R)^+$ *and* $(L_c^R)^*$, *and*
(c) $L = \{w \in \{a,b\}^* \mid |w|_a \leq |w|_b \leq 2 \cdot |w|_a\}$.

**Proof.** (a) Let $\Sigma = \{a,b,c\}$. For deriving a contradiction, we assume that $A = (Q,\Sigma,\lhd,\tau,q_0,\delta)$ is an RDFAwtl such that $L(A) = L_c$. Without loss of generality, we may assume that $A$ reads and deletes its input completely during each accepting computation.

For each $n \geq 1$, the word $a^nb^nc$ belongs to the language $L_c$. Accordingly, the computation of $A$ on input $w_n = a^nb^nc$ is of the form $q_0a^nb^nc\cdot\lhd \vdash_A^* q_f\cdot\lhd \vdash_A \mathsf{Accept}$, where $q_f \in Q$. Thus, in particular, the single occurrence of the letter $c$ is read and deleted during this computation. Accordingly, this computation can be written as follows:

$$q_0a^nb^nc\cdot\lhd \vdash_A^* qa^{n-i}b^{n-j}c\cdot\lhd \vdash_A q'a^{n-i}b^{n-j}\cdot\lhd \vdash_A^* q_f\cdot\lhd \vdash_A \mathsf{Accept},$$

where $q,q' \in Q$, $0 \leq i,j \leq n$, and $\delta(q,c) = q'$. Then $A$ can also execute the following computation:

$$q_0a^ib^jca^{n-i}b^{n-j}\cdot\lhd \vdash_A^* qca^{n-i}b^{n-j}\cdot\lhd \vdash_A q'a^{n-i}b^{n-j}\cdot\lhd \vdash_A^* q_f\cdot\lhd \vdash_A \mathsf{Accept}.$$

However, the word $a^ib^jca^{n-i}b^{n-j}$ is not an element of the language $L_c$ unless $i = n$ and $j = n$, which means that, in the accepting computation above, $A$ first reads and deletes all occurrences of the letters $a$ and $b$ before it reads the single occurrence of the letter $c$. In particular, it follows that this computation has the form $q_0a^nb^nc\cdot\lhd \vdash_A^* qc\cdot\lhd \vdash_A q'\cdot\lhd \vdash_A^* q_f\cdot\lhd \vdash_A \mathsf{Accept}$.

Now, let $n > |Q|$. If the computation $q_0a^nb^nc\cdot\lhd \vdash_A^* qc\cdot\lhd$ begins with $|Q|$ many steps that each read an occurrence of the letter $a$, then there is a state $q \in Q$ that is used twice during these steps, that is

$$q_0a^nb^nc\cdot\lhd \vdash_A^* qa^{n-i}b^nc\cdot\lhd \vdash_A^+ qa^{n-i-j}b^nc\cdot\lhd \vdash_A^* \mathsf{Accept},$$

where $i \geq 0$, $j \geq 1$, and $i + j \leq |Q|$. But then $A$ can also execute the following accepting computation:

$$q_0a^{n-j}b^nc\cdot\lhd \vdash_A^* qa^{n-i-j}b^nc\cdot\lhd \vdash_A^* \mathsf{Accept}.$$

However, as $j \geq 1$, the word $a^{n-j}b^nc$ does not belong to the language $L_c$, a contradiction. This implies that, within the first $|Q|$ many steps in the accepting computation considered, an occurrence of the letter $b$ is read and deleted.

Let us now consider the first step in the above computation during which an occurrence of the letter $b$ is deleted:

$$q_0a^nb^nc\cdot\lhd \vdash_A^* q_1a^{n-i}b^nc\cdot\lhd \vdash_A q_2a^{n-i}b^{n-1}c\cdot\lhd,$$

where $0 \leq i < |Q|$, $q_1,q_2 \in Q$, $a \in \tau(q_1)$, and $\delta(q_1,b) = q_2$. Given the input $a^nc \in L_c$, $A$ will also accept. However, as $A$ is deterministic, the corresponding accepting computation begins with

$$q_0a^nc\cdot\lhd \vdash_A^* q_1a^{n-i}c\cdot\lhd,$$

where $a \in \tau(q_1)$.

If $c \in \tau(q_1)$, then together with the partial computation $q_0a^nbc\cdot\lhd \vdash_A^* q_1a^{n-i}bc\cdot\lhd \vdash_A q_2a^{n-i}c\cdot\lhd$, the automaton $A$ could also execute the following partial computation:

$$q_0a^ncb\cdot\lhd \vdash_A^* q_1a^{n-i}cb\cdot\lhd \vdash_A q_2a^{n-i}c\cdot\lhd.$$

As $a^nbc \in L_c$, the former computation leads to acceptance and, hence, so does the latter. This yields a contradiction, as $a^ncb \notin L_c$. Hence, it follows that $c \notin \tau(q_1)$.

This implies that, in the above configuration $q_1a^{n-i}c\cdot\lhd$, $A$ reads and deletes the letter $c$, that is, $\delta(q_1,c) = q'$ for some $q' \in Q$. Thus, we obtain the accepting computation

$$q_0a^nc\cdot\lhd \vdash_A^* q_1a^{n-i}c\cdot\lhd \vdash_A q'a^{n-i}\cdot\lhd \vdash_A^* \mathsf{Accept}.$$

But then, we also obtain the accepting computation

$$q_0 a^i c a^{n-i} \cdot \lhd \vdash_A^* q_1 c a^{n-i} \cdot \lhd \vdash_A q' a^{n-i} \cdot \lhd \vdash_A^* \mathsf{Accept},$$

which yields a contradiction, as $a^i c a^{n-i} \notin L_c$. In summary, this shows that the language $L_c$ is not accepted by any RDFAwtl.

(b) Assume to the contrary that $(L_c^R)^+$ or $(L_c^R)^*$ is accepted by an RDFAwtl. Lemma 19 implies that the language

$$\begin{array}{rcl} L & = & c \setminus (L_c^R)^+ \;\; = \;\; c \setminus (L_c^R)^* \\ & = & \{ w_1 c w_2 c \cdots c w_k \mid k \geq 1 \text{ and } \forall i = 1, 2, \ldots, k : w_i \in \{a,b\}^* \wedge |w_i|_a \geq |w_i|_b \} \end{array}$$

is also accepted by an RDFAwtl $A = (Q, \Sigma, \lhd, \tau, q_0, \delta)$, where $\Sigma = \{a,b,c\}$. For all $m \geq 0$, the word $w(m) = a^m b^m cab$ belongs to the language $L$, which means that the computation of $A$ on input $w(m)$ is accepting. However, using pumping arguments as in the proof of (a), it can now be shown that, together with the words $w(m)$, the RDFAwtl $A$ also accepts some words that do not belong to the language $(L_c^R)^*$. Accordingly, the languages $(L_c^R)^+$ and $(L_c^R)^*$ are not accepted by any RDFAwtls.

(c) By using pumping arguments, it can be proved that an RDFAwtl accepting the language

$$L = \{ w \in \{a,b\}^* \mid |w|_a \leq |w|_b \leq 2 \cdot |w|_a \}$$

also accepts some words that do not belong to this language. Again, this shows that this language is not accepted by any RDFAwtl. □

It is easily seen that the languages

$$L_c^R = \{ cw \mid w \in \{a,b\}^*, |w|_a \geq |w|_b \}, L_{\geq} = \{ w \in \{a,b\}^* \mid |w|_a \geq |w|_b \}, \text{ and } \{c\}$$

are all accepted by DFAwtls. On the other hand, $L = L_{=} \sqcup\!\sqcup L_{2=} = \{ w \in \{a,b\}^* \mid |w|_a \leq |w|_b \leq 2 \cdot |w|_a \}$ is not. Hence, Lemma 21 shows the following.

**Corollary 22** *The language class $\mathscr{L}(\mathsf{RDFAwtl})$ is not closed under reversal, (disjoint) product, Kleene plus, Kleene star, or shuffle.*

Finally, the class $\mathscr{L}(\mathsf{RDFAwtl})$ is closed under a restricted variant of the shuffle operation. If $\Sigma$ is an alphabet and $\Delta$ is a subalphabet of $\Sigma$, we shall use $P_\Delta$ to denote the projection $P_\Delta : \Sigma \to \Delta$ that maps each letter from $\Delta$ to itself and each letter from $\Sigma \setminus \Delta$ to the empty word $\lambda$. The projection $P_\Delta$ can be extended to words and languages in a natural way.

If a word $w \in (\Sigma_A \cup \Sigma_B)^*$ is in the set $u \sqcup\!\sqcup v$ for some words $u \in \Sigma_A^*$ and $v \in \Sigma_B^*$, where $\Sigma_A$ and $\Sigma_B$ are two disjoint alphabets, then $P_{\Sigma_A}(w) = u$ and $P_{\Sigma_B}(w) = v$. The shuffle of two words or languages over disjoint alphabets is called a *disjoint shuffle*.

**Proposition 23** *The language class $\mathscr{L}(\mathsf{RDFAwtl})$ is closed under disjoint shuffle.*

**Proof.** Let $\Sigma_A$ and $\Sigma_B$ be two disjoint alphabets, let $A = (Q_A, \Sigma_A, \lhd, \tau_A, q_0^{(A)}, \delta_A)$ be an RDFAwtl on $\Sigma_A$ that accepts a language $L(A) = L_A$, and let $B = (Q_B, \Sigma_B, \lhd, \tau_B, q_0^{(B)}, \delta_B)$ be an RDFAwtl on $\Sigma_B$ that accepts a language $L(B) = L_B$. We shall construct an RDFAwtl $M$ for the disjoint shuffle $L = L_A \sqcup\!\sqcup L_B$ of $L_A$ and $L_B$.

By Lemma 13, we can assume without loss of generality that $A$ never gets into an infinite computation and that it reads and deletes its input completely in each accepting computation. The RDFAwtl

$M = (Q, \Sigma, \lhd, \tau, q_0, \delta)$ is constructed as follows, where we assume that the sets of states $Q_A$ and $Q_B$ are disjoint:

- $Q \quad = Q_A \cup Q_B$ and $q_0 = q_0^{(A)}$,
- $\tau(q) = \begin{cases} \tau_A(q) \cup \Sigma_B, & \text{if } q \in Q_A, \\ \tau_B(q), & \text{if } q \in Q_B, \end{cases}$
- $\delta(q,a) = \begin{cases} \delta_A(q,a), & \text{if } q \in Q_A \text{ and } a \in \Sigma_A, \\ \delta_B(q,a), & \text{if } q \in Q_B \text{ and } a \in \Sigma_B, \\ \emptyset, & \text{otherwise,} \end{cases}$

  $\delta(q,\lhd) = \begin{cases} \delta_A(q,\lhd), & \text{if } q \in Q_A \text{ and } \delta_A(q,\lhd) \neq \mathsf{Accept}, \\ q_0^{(B)}, & \text{if } q \in Q_A \text{ and } \delta_A(q,\lhd) = \mathsf{Accept}, \\ \delta_B(q,\lhd), & \text{if } q \in Q_B. \end{cases}$

For an input $w \in u \sqcup\!\sqcup v$, where $u \in \Sigma_A^*$ and $v \in \Sigma_B^*$, the RDFAwtl $M$ first simulates the computation of $A$ on $u$, ignoring all occurrences of letters from $\Sigma_B$. When $A$ accepts, then $M$ simulates the computation of $B$ on $v$. As $A$ reads and deletes all letters of $u$ in its accepting computation, it is obvious that $M$ accepts on input $w$ if and only if $A$ accepts on input $u$ and $B$ accepts on input $v$. It follows that $L(M) = L(A) \sqcup\!\sqcup L(B) = L_A \sqcup\!\sqcup L_B$. Hence, $\mathscr{L}(\mathsf{RDFAwtl})$ is closed under the operation of disjoint shuffle. □

## 5 Conclusion

Concerning the complexity of the membership problem, it is easily seen that the algorithm for the membership problem of a DFAwtl presented by Nagy and Kovács in [11] applies to an RDFAwtl as well. This yields the following result. Note that the complexity of the membership problem of a DFAwtl is measured using a logarithmic cost of instructions.

**Corollary 24** *The membership problem for an RDFAwtl is decidable in time $O(n \cdot \log n)$.*

Furthermore, emptiness and finiteness are decidable for RDFAwtls, as they are decidable for NFAwtls [16]. As $\mathscr{L}(\mathsf{RDFAwtl})$ is effectively closed under complementation, universality is also decidable for these automata. On the other hand, the problem of deciding whether the language accepted by a given RDFAwtl has a non-empty intersection with a given regular language is undecidable, as this problem is already undecidable for DFAwtls. The same holds for the problem of deciding whether the language accepted by a given RDFAwtl contains (or is contained in) a given regular language, and therewith, the inclusion problem for RDFAwtls is undecidable, too. However, it remains open whether the equivalence problem is decidable for RDFAwtls.

In summary, our results show that the RDFAwtl is just slightly more expressive than the DFAwtl, but it seems to have the same closure and non-closure properties, and the same decidability and undecidability results seem to hold. Moreover, we have seen that, when we add the property of being non-returning to the DFAwtl (or the NFAwtl), we actually weaken the model. When we add the property of repetitiveness to the DFAwtl, then we obtain a language class that is just a bit more expressive than the DFAwtl, while in the nondeterministic case, this generalization has no effect on the expressive capacity of the model. However, when we add both these properties, repetitiveness and the property of being non-returning, then the resulting types of automata, that is, the nr-DFAwtl and the nr-NFAwtl, have indeed a much larger expressive capacity than the original models. Thus, it is really the combination of these two properties that implicates the enormous increase in the expressive capability from the DFAwtl and the NFAwtl to the nr-DFAwtl and the nr-NFAwtl.

## References

[1] Simon Beier & Markus Holzer (2022): *Nondeterministic right one-way jumping finite automata*. *Information and Computation* 284, doi:10.1016/j.ic.2021.104687.

[2] Suna Bensch, Henning Bordihn, Markus Holzer & Martin Kutrib (2009): *On input-revolving deterministic and nondeterministic finite automata*. *Information and Computation* 207, pp. 1140–1155, doi:10.1016/j.ic.2009.03.002.

[3] Hiroyuki Chigahara, Szilárd Zsolt Fazekas & Akihiro Yamamura (2016): *One-way jumping finite automata*. *International Journal of Foundations of Computer Science* 27, pp. 391–405, doi:10.1142/S0129054116400165.

[4] Elias Dahlhaus & Manfred K. Warmuth (1986): *Membership for growing context-sensitive grammars is polynomial*. *Journal of Computer and System Sciences* 33, pp. 456–472, doi:10.1016/0022-0000(86)90062-0.

[5] Henning Fernau, Meenakshi Paramasivan & Markus L. Schmid (2015): *Jumping Finite Automata: Characterizations and Complexity*. In Frank Drewes, editor: *CIAA 2015, Proc.*, *LNCS* 9223, Springer, Berlin, pp. 89–101, doi:10.1007/978-3-319-22360-5_8.

[6] Petr Jančar, František Mráz, Martin Plátek & Jörg Vogel (1995): *Restarting automata*. In Horst Reichel, editor: *FCT'95, Proc.*, *LNCS* 965, Springer, Berlin, pp. 283–292, doi:10.1007/3-540-60249-6_60.

[7] Alexander Meduna & Petr Zemek (2012): *Jumping finite automata*. *International Journal of Foundations of Computer Science* 23, pp. 1555–1578, doi:10.1142/S0129054112500244.

[8] František Mráz (2001): *Lookahead hierarchies of restarting automata*. *Journal of Automata, Languages and Combinatorics* 6, pp. 493–506, doi:10.25596/jalc-2001-493.

[9] František Mráz & Friedrich Otto (2022): *Non-returning finite automata with translucent letters*. In Henning Bordihn, Géza Horváth & György Vaszil, editors: *12th International Workshop on Non-Classical Models of Automata and Applications (NCMA 2022)*, *EPTCS* 367, Open Publishing Association, Waterloo, Australia, pp. 143–159, doi:10.4204/EPTCS.367.10.

[10] František Mráz & Friedrich Otto (2023): *Non-returning deterministic and nondeterministic finite automata with translucent letters*. *RAIRO Theoretical Informatics and Applications* 57, doi:10.1051/ita/2023009.

[11] Benedek Nagy & László Kovács (2014): *Finite Automata with Translucent Letters Applied in Natural and Formal Language Theory*. In Ngoc Thanh Nguyen, Ryszard Kowalczyk, Ana Fred & Filipe Joaquim, editors: *Transactions on Computational Collective Intelligence XVII*, *LNCS* 8790, Springer, Heidelberg, pp. 107–127, doi:10.1007/978-3-662-44994-3_6.

[12] Benedek Nagy & Friedrich Otto (2010): *CD-systems of stateless deterministic R(1)-automata accept all rational trace languages*. In Adrian-Horia Dediu, Henning Fernau & Carlos Martín-Vide, editors: *LATA 2010, Proc.*, *LNCS* 6031, Springer, Berlin, pp. 463–474, doi:10.1007/978-3-642-13089-2_39.

[13] Benedek Nagy & Friedrich Otto (2011): *Finite-state acceptors with translucent letters*. In Gemma Bel-Enguix, Veronica Dahl & Alfonso O. de la Puente, editors: *BILC 2011: AI Methods for Interdisciplinary Research in Language and Biology, Proc.*, SciTePress, Portugal, pp. 3–13, doi:10.5220/0003272500030013.

[14] Benedek Nagy & Friedrich Otto (2011): *Globally deterministic CD-systems of stateless R(1)-automata*. In Adrian-Horia Dediu, Shunsuke Inenaga & Carlos Martín-Vide, editors: *LATA 2011, Proc.*, *LNCS* 6638, Springer, Berlin, pp. 390–401, doi:10.1007/978-3-642-21254-3_31.

[15] Benedek Nagy & Friedrich Otto (2012): *On CD-systems of stateless deterministic R-automata with window size one*. *Journal of Computer and System Sciences* 78, pp. 780–806, doi:10.1016/j.jcss2011.12.009.

[16] Benedek Nagy & Friedrich Otto (2013): *Globally deterministic CD-systems of stateless R-automata with window size 1*. *International Journal of Computer Mathematics* 90, pp. 1254–1277, doi:10.1080/00207160.2012.688820.

[17] Friedrich Otto (2006): *Restarting automata*. In Zoltan Ésik, Carlos Martín-Vide & Victor Mitrana, editors: *Recent Advances in Formal Languages and Applications*, *Studies in Computational Intelligence* 25, Springer, Heidelberg, pp. 269–303, doi:10.1007/978-3-540-33461-3_11.

# $5' \rightarrow 3'$ Watson-Crick Automata accepting Necklaces

Benedek Nagy
Department of Mathematics, Eastern Mediterranean University
99628 Famagusta, North Cyprus, Mersin-10, Turkey
Department of Computer Science, Institute of Mathematics and Informatics,
Eszterházy Károly Catholic University, Eger, Hungary
nbenedek.inf@gmail.com

Watson-Crick (WK) finite automata work on a Watson-Crick tape representing a DNA molecule. They have two reading heads. In $5' \rightarrow 3'$ WK automata, the heads move and read the input in opposite physical directions. In this paper, we consider such inputs which are necklaces, i.e., they represent circular DNA molecules. In sensing $5' \rightarrow 3'$ WK automata, the computation on the input is finished when the heads meet. As the original model is capable of accepting the linear context-free languages, the necklace languages we are investigating here have strong relations to that class. Here, we use these automata in two different acceptance modes. On the one hand, in *weak* acceptance mode the heads are starting nondeterministically at any point of the input, like the necklace is cut at a nondeterministically chosen point), and if the input is accepted, it is in the accepted necklace language. These languages can be seen as the languages obtained from the linear context-free languages by taking their closure under cyclic shift operation. On the other hand, in *strong* acceptance mode, it is required that the input is accepted starting the heads in the computation from every point of the cycle. These languages can be seen as the maximal cyclic shift closed languages included in a linear language. On the other hand, as it will be shown, they have a kind of locally testable property. We present some hierarchy results based on restricted variants of the WK automata, such as stateless or all-final variants.

**Keywords:** Watson-Crick automata, $5' \rightarrow 3'$ WK automata, languages of circular words, finite state acceptors, hierarchy, bio-inspired computing, weak and strong acceptance

## 1 Introduction

On the one hand, there are numerous new computational paradigms that emerged in the last decades, usually based on or motivated by some natural phenomena [31]. A number of them are connected to DNA molecules, thus DNA computing has various theoretical [30] and various experimental branches (based e.g., on [1]). Both Watson-Crick automata and the theory/combinatorics of circular words (also called necklaces) are belonging to theoretical DNA motivated models. On the other hand, as their names already hint, they have strong connections to classical computing theory, including automata and formal languages. Watson-Crick automata (abbreviated by the first and last letters of the names of the Nobel prize winner discoverers of the DNA molecule structure, i.e., WK automata), were introduced in [5] as an automata type model of DNA computing [33, 4]. These automata are interesting both from theoretical aspects of computations and also from their applicability in bioinformatical problems [34]. The DNA molecules, from a computational point of view, can be seen as linear or circular double stranded words over the alphabet of nucleotides, such that the two strands are related by the Watson-Crick complementarity relation (that is, in nature, a bijective pairing relation on the used 4 nucleotides). The original models of WK automata work on double-stranded tapes called Watson-Crick tapes that represent (linear) DNA molecules and the two read-only heads scanning the two strands in a correlated manner. These automata are closely related to finite automata having two heads. From the biological point of view there are some restrictions that could be applied on the model, e.g., on the number of states or on the number of input letters being read in a transition. Relationships between various restricted classes of the Watson-Crick automata were presented in [5, 30, 12]. From another important biological motivation, the reverse and $5' \to 3'$ WK automata make more sense: each (linear) DNA strand has its own $5'$ and $3'$ end, where these names come from the position of the carbon atoms in the sugar part to which the next nucleotide can connect by covalent bond. The two strands of a DNA molecule have opposite chemical direction, i.e., the $5'$ end of a strand gives the $3'$ end of the other and vice versa. Thus, if one believes that in these automata a biochemical sensor, an enzyme, may read the strands, then, most probably, the enzyme reads the two strands in the same chemical direction, i.e., from their $5'$ ends to the direction of their $3'$ ends [5, 17, 13, 14]. While the reverse variant of WK automata is essentially the same as the full-reading non-sensing variant of $5' \to 3'$ WK automata [30, 13], in the sensing version, the computation on an input finishes at latest when the two heads meet. This sensing was taken into account with a rather artificial sensing parameter in [17, 21], while without it in [28, 27, 29, 26]. In [19] specific both-head stepping variants were defined, where both heads move together and read letter by letter the input (till they meet). We should mention here that sensing $5' \to 3'$ WK automata is closely related to other 2-head finite automata models described under various names like linear automata [15], biautomata [9] or simply 2-head automata [20], as their class is capable to accept the class of the linear context-free languages. The specific variant shown in [19] is able to accept the so-called even-linear languages [2, 35]. Other restricted version, namely $5' \to 3'$ WK automata with exactly one state, was investigated in details in [22]. Some extensions of the $5' \to 3'$ WK automata were also developed, e.g., jumping $5' \to 3'$ WK automata [10], combination with automata with translucent letters [24, 25] or $5' \to 3'$ WK transducers [23].

In this paper, the model of $5' \to 3'$ WK automata is used for languages of necklaces, i.e., sets of circular words. As there are circular DNA molecules, it is of particular interest to investigate these automata and analyze their computational power, etc. As usual, we are using linearization of necklaces, i.e., we represent a necklace by the set of (linear) words that are obtained as the conjugate class of any of the words that can represent the necklace. We use two modes of acceptance: a necklace is accepted in the weak mode if any of its conjugates is accepted by the given automaton; and those necklaces are accepted in the strong mode for which each of their conjugates are accepted.

In the next section, we formally define our concepts, and then in Sections 3 and 4 we give a sequence of hierarchy results among the accepted classes of necklace languages including all-final, simple, 1-limited, and stateless $5' \to 3'$ Watson-Crick automata in case of the weak and strong accepting mode for necklaces, respectively. Conclusions and open questions will close the paper.

Here, we recall only one of the main results for each of the acceptance modes:

- a language can be weakly accepted by a $5' \to 3'$ WK automaton if and only if it is the cyclic closure of a linear context-free language.
- if a language is strongly accepted by a $5' \to 3'$ WK automaton, then the language has a kind of locally testable property.

## 2 Preliminaries

We assume that the reader is familiar with basic concepts of formal languages and automata, otherwise she or he is referred to [8, 32]. For any unexplained notions about DNA computing we refer, e.g., to [30]. We denote the empty word by $\lambda$, and the sets of positive and nonnegative integers by $\mathbb{N}$ and $\mathbb{N}_0$, respectively.

Let $T$ be an *alphabet*, then for any word $w \in T^*$ if $w = uv$, then the word $vu$ is a *conjugate* of $w$, and the set of all conjugates of $w$ is called a *necklace* (or cyclic, or circular word) $w_\circ$. The operation by which we can obtain each element of the class is called *cyclic shift*, i.e., the cyclic shift of $aw$ is $wa$ where $a \in T$ and $w \in T^*$. The subsequent application of the cyclic shift operation $cycl$ (at most as many times as the length of the word) obtains each conjugate of the word we start with. Periodic properties of circular words were studied in [6, 7], where a weak period of a circular word was defined as a period of an element of the conjugate class and a period was a strong period if it was a period of each element of the conjugate class. In fact, what we are dealing with is the linearization of the circular words. One can imagine those as words written in a cyclic way joining (i.e., concatenating) the first letter of the word after its last letter, in this way obtaining the word without a starting and without an ending point. Languages of necklaces are also studied in the literature [11]. In this paper, we use necklaces to model (describe) circular DNA molecules. A *language of necklaces* is represented by the union of the necklaces, i.e., conjugate classes. Obviously, this condition can be translated as follows: a language $L \subset T^*$ is a language of necklaces if for any word $w \in L$ each conjugate of $w = uv$ is also in $L$, i.e. $vu \in L$. Consequently, necklace languages are exactly those languages that are closed under cyclic shift operation, when we apply the operation $cycl$ for a language as follows: $cycl(L) = \{uv \mid u, v \in T^*, w = vu \in L\} = \bigcup\limits_{w \in L} w_\circ$. Further, the cyclic closure of a class $\mathscr{L}$ of languages is the class of the cyclic closures of the languages in $\mathscr{L}$.

One class of the Chomsky hierarchy, the class of linear context-free languages, has a strong connection to the automata model we start with, thus we recall it briefly. A generative grammar $G = (N, T, S, P)$ is *linear context-free* if every production is context-free and contains at most one nonterminal on the right hand side, i.e., it is one of the forms $A \to u$, $A \to uBv$ with $A, B \in N$ and $u, v \in T^*$. A language $L$ is linear context-free if it can be generated by a linear context-free grammar. This class of languages is denoted by $\mathscr{L}_{LIN}$. It is known that, on the one hand, the classes of regular and context-free languages (denoted by $\mathscr{L}_{context-free}$) are closed under cyclic shift, on the other hand, the class of linear context-free languages is not [3, 8].

The two strands of the DNA molecule have opposite $5' \to 3'$ orientations. Therefore, Watson-Crick finite automata that parse the two strands of the Watson-Crick tape in opposite directions are investigated. Now, we use them to accept necklace languages. Figure 1 indicates the initial configuration of such an automaton. As there is no specific start and end point of a necklace, the starting point can be chosen arbitrarily (and based on that we will define two types of acceptance conditions).

A $5' \to 3'$ WK automaton is called *sensing* if it senses that its heads are meeting, i.e., they are in the same position. As in these models the full input is already processed at that time (if the heads meet again), we use the model to make the decision of the type of the computation at that point, i.e., if the computation is an accepting computation.

Formally, a *Watson-Crick automaton* is a 6-tuple $M = (V, \rho, Q, q_0, F, \delta)$, where:

- $V$ is the (input or tape) alphabet,
- $\rho \subseteq V \times V$ denotes a complementarity relation,
- $Q$ represents a finite set of states,
- $q_0 \in Q$ is the initial state,
- $F \subseteq Q$ is the set of final (also called accepting) states and
- $\delta$ is called the transition function and it is of the form $\delta : Q \times \begin{pmatrix} V^* \\ V^* \end{pmatrix} \to 2^Q$, such that it is non-empty only for finitely many triplets $(q, u, v), q \in Q, u, v \in V^*$ when these triplets may also be

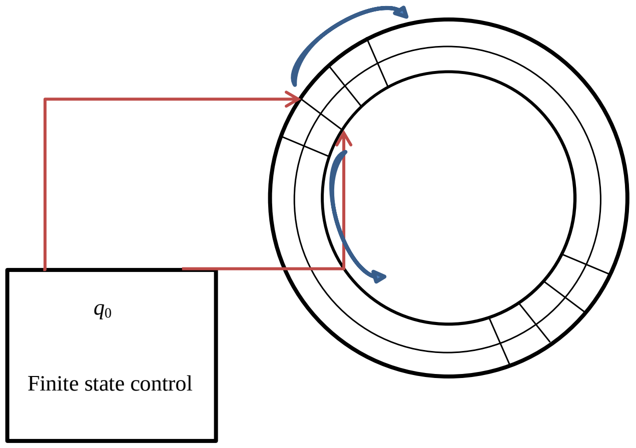


Figure 1: A sensing $5' \to 3'$ WK automaton in the initial configuration.

written either in the form $(q,u,v)$ or $(q,\binom{u}{v})$ indicating which of the strings are read by which of the heads. (The heads can be called upper (left or first) and lower (right or second) heads, respectively.

Based on our definition, in these WK automata every pair of positions in the Watson-Crick tape is read by exactly one of the heads in an accepting computation, thus the complementarity relation cannot play importance; instead, in this paper we always assume that it is the identity relation. We are presenting the sensing $5' \to 3'$ WK automaton in Figures 1 and 2 working on the 2-strand necklace. However, for the above reason, it is more convenient to consider the input as a "normal" necklace and not a double stranded necklace. Actually, this is a usual trick to simplify the notation, as, in some cases, also instead of the nucleotide pairs, e.g., $\begin{bmatrix} C \\ G \end{bmatrix}$ (with $C,G \in V$) one may simply write $a \in T$, by shifting the description to a new alphabet, which can be done always if the complementarity relation is symmetric and bijective (as in the case of real DNA). Thus, we may use alphabet $T$ instead of using $V$ and $\rho$, to *simplify the writing of* $5' \to 3'$ *WK automaton* to a 5-tuple $M = (T,Q,q_0,F,\delta)$, modifying $\delta$ appropriately to use $T$. On the other hand, the complementarity relation can always be replaced by the identity, even in the traditional models, as was proven in [12].

By continuing the formal description, we consider the *computation* of $5' \to 3'$ WK automata on necklaces as finite sequences of configurations. A *configuration* is a pair $(q,w)$ where $q \in Q$ is the current state of the automaton and $w$ is the part of the input necklace which has not been read (processed) yet written as a normal word as we detail it. In the initial configuration, the initial state $q_0$ is used with any element of the conjugate class of the necklace, mimicking the arbitrary (nondeterministic) choice of a position of the cycle from where the computation starts: the conjugate starting at that position will be processed. As the 2 heads are moving in opposite physical directions, the unprocessed part between them will be shorter and shorter until the heads meet (i.e., they are both in the same position again, as this is shown in Figure 2). Formally, let $w',x,y \in T^*$, $q,q' \in Q$. Then, there is a computation step between two configurations: $(q,xw'y) \Rightarrow (q',w')$ if and only if $q' \in \delta(q,x,y)$. The reflexive and transitive closure of the relation $\Rightarrow$ is, as usual, denoted by $\Rightarrow^*$ and called computation. For a given conjugate $w \in T^*$ of the input, an accepting computation is a sequence of transitions $(q_0,w) \Rightarrow^* (q_F,\lambda)$, starting from an initial

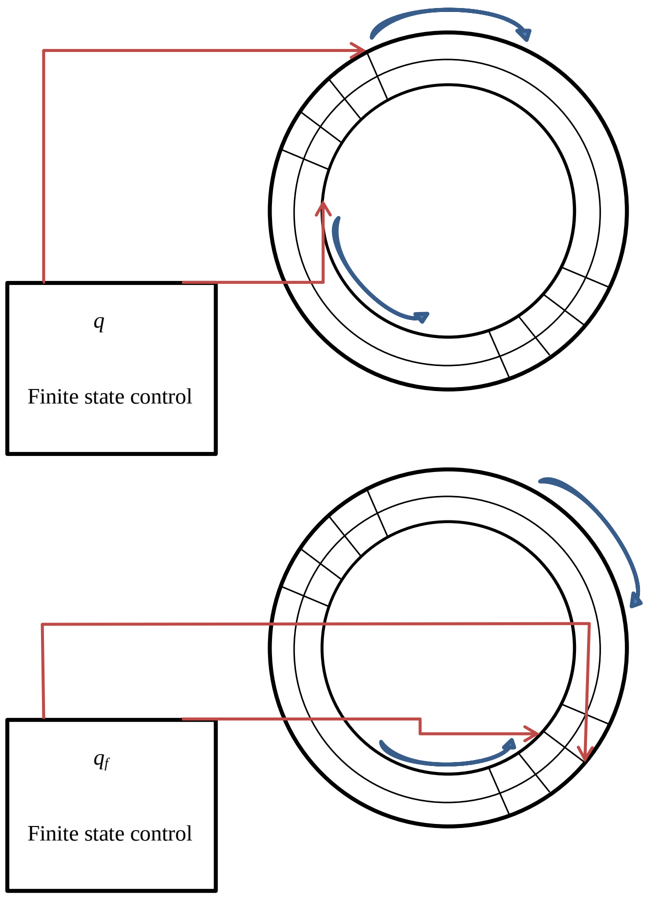


Figure 2: A sensing $5' \rightarrow 3'$ WK automaton in a configuration during a computation and in an accepting configuration with a final state $q_f$ (bottom).

configuration and ending in a configuration consisting of a final state and the empty word. Now, based on the conjugate class of a necklace we define our acceptance conditions:

1. A necklace $w_\circ$ is *weakly accepted* by a WK automaton $M$ if there is a conjugate $vu$ (when $w = uv$, i.e., in this case $vu \in w_\circ$) such that there is an accepting computation on $vu$.
2. A necklace $w_\circ$ is *strongly accepted* by a WK automaton $M$ if there is an accepting computation for every conjugate of $w$ (i.e., each element of $w_\circ$).

The weak and strong distinction comes in a similar manner as it was used for periods in [7]. Now, one may consider the former case, as there is a nondeterministic choice for where to cut the necklace to start the computation, and if this (nondeterministically chosen) starting point leads to an accepting computation, then the necklace is accepted. Contrariwise, in the latter case, there must be an accepting computation for each possible starting point for a necklace to be included in the accepted language.

1. The language $L$ of necklaces is weakly accepted by a WK automaton $M$ if for each word $w \in L$, there is a conjugate $vu$ (when $w = uv$) such that there is an accepting computation on $vu$.

2. The language $L$ of necklaces is strongly accepted by a WK automaton $M$ if for each necklace $w_\circ \subset L$, all conjugates of $w$ (i.e., each element of $w_\circ$) are accepted by $M$ by some computations.

We may also write these conditions more formally and we can also use some special notation for these languages:

1. $L_w(M) = \{w \in T^* \mid \text{ there exist } u,v \in T^*,\ q_f \in F \text{ such that } w = uv, (q_0, vu) \Rightarrow^* (q_f, \lambda)\}$.
2. $L_s(M) = \{w \in T^* \mid \text{for each } u \in w_\circ \text{ there is a } q_f \in F \text{ such that } (q_0, u) \Rightarrow^* (q_f, \lambda)\}$.

The classes of necklace languages weakly and strongly accepted by sensing $5' \to 3'$ WK automata are denoted by $\mathscr{W}_*$ and $\mathscr{S}_*$. There are some restricted variants of WK automata which are usually considered (e.g., from computational and biological motivations):

- **N**: stateless, i.e., with only one state: if $Q = F = \{q_0\}$;
- **F**: all-final, i.e., with only final states: if $Q = F$;
- **S**: simple (at most one head moves in a step) $\delta : (Q \times ((\{\lambda\}, T^*) \cup (T^*, \{\lambda\}))) \to 2^Q$.
- **1**: 1-limited (exactly one letter is being read in each step) $\delta : (Q \times ((\{\lambda\}, T) \cup (T, \{\lambda\}))) \to 2^Q$.

Let $\mathscr{W}_N$, $\mathscr{W}_F$, $\mathscr{W}_S$ and $\mathscr{W}_1$ denote the necklace language classes weakly accepted by sensing **N**, **F**, **S** and **1** $5' \to 3'$ WK automata, respectively. Further variants having multiple constraints can also be defined as sensing **F1**, **N1**, **FS**, **NS** $5' \to 3'$ WK automata. Their weakly accepted language classes are denoted by $\mathscr{W}_{F1}$, $\mathscr{W}_{N1}$, $\mathscr{W}_{FS}$ and $\mathscr{W}_{NS}$, respectively. Similarly, the notation $\mathscr{S}_N, \mathscr{S}_F, \mathscr{S}_S, \mathscr{S}_1$, etc. will be used for the classes that are strongly accepted by the restricted classes of $5' \to 3'$ WK automata, respectively. Further, we may use the traditional way of acceptance for 'ordinary' (i.e., not necessarily necklace) languages and we use the notation for these classes, $\mathscr{L}_*, \mathscr{L}_N$, etc., respectively.

## 3 On weakly accepted necklace language classes

With this section our aim is twofold. On the one hand we would like to present some general result on the class $\mathscr{W}_*$ and, on the other hand, we are presenting hierarchy results among the language classes of necklaces that are weakly accepted by the restricted models.

The next proposition is a direct consequence of the definitions and the fact that exactly the class $\mathscr{L}_{LIN}$ is accepted the class of (unrestricted) sensing $5' \to 3'$ WK finite automata [17, 28, 18, 21, 27].

**Proposition 1** *The cyclic closure $cycl(\mathscr{L}_{LIN})$ is weakly accepted by sensing $5' \to 3'$ WK finite automata, that is, for each linear language $L$, its cyclic closure $cycl(L)$ is in $\mathscr{W}_*$ and for each language $L' \in \mathscr{W}_*$ there is a linear language $L''$ such that $L' = cycl(L'')$.*

*Moreover, for each restricted class* $\mathbf{x} \in \{\mathbf{S},\mathbf{1},\mathbf{F},\mathbf{N},\mathbf{FS},\mathbf{F1},\mathbf{NS},\mathbf{N1}\}$,

$$\mathscr{W}_x = cycl(\mathscr{L}_x),$$

*i.e., the class of weakly accepted necklace languages by a restricted class of sensing $5' \to 3'$ WK finite automata is the same as the cyclic closure of the languages accepted by the class of sensing $5' \to 3'$ WK finite automata with the same restriction.*

The cyclic closure of the class $\mathscr{L}_{LIN}$ was also defined as a kind of class of necklace languages (i.e., languages of cyclic words) among many other classes based on a somewhat similar idea in [11].

As $\mathscr{L}_{LIN}$ is not closed, but $\mathscr{L}_{context-free}$ is closed under cyclic shift [3, 8], we can relate our classes to the Chomsky hierarchy. As, clearly both $\mathscr{L}_{LIN}$ and $\mathscr{L}_{context-free}$ contain some languages that are not necklace languages, (e.g., the singleton language $\{ab\}$), we have:

**Proposition 2** *The inclusion $\mathscr{W}_* \subsetneq \mathscr{L}_{context-free}$ is proper, while the classes $\mathscr{W}_*$ and $\mathscr{L}_{LIN}$ are incomparable under set theoretic inclusion.*

Now we show some equivalences among the classes on the top of the hierarchy. By, e.g., [28, 27], it is known that sensing $5' \to 3'$ WK finite automata accept exactly the linear context-free languages, moreover the same class is accepted by the classes of the following variants: $\mathscr{L}_{LIN} = \mathscr{L}_* = \mathscr{L}_S = \mathscr{L}_1$. This gives the consequence that the weakly accepted classes will also be the same:

**Proposition 3** *The following classes of necklace languages are identical: $cycl(\mathscr{L}_{LIN}) = \mathscr{W}_* = \mathscr{W}_S = \mathscr{W}_1$.*

In the rest of the section we present various hierarchy results of the considered necklace languages.

To show that none of he language classes is empty, we start with the most restricted class, the necklace languages weakly accepted by sensing **N1** $5' \to 3'$ WK automata, to give an example language.

**Proposition 4** *The language $L = \{1^i 0^j 1^k \mid i,j,k \in \mathbb{N}_0\} \cup \{0^i 1^j 0^k \mid i,j,k \in \mathbb{N}_0\}$ is weakly accepted by the sensing **N1** $5' \to 3'$ WK automaton: $M = (\{0,1\},\{q\},q,\{q\},\delta)$ with two allowed transitions $q \in \delta(q,0,\lambda)$ and $q \in \delta(q,\lambda,1)$.*

**Proof** Clearly the automaton has only one state and it reads exactly one letter in each step of the computation, thus it is a sensing **N1** $5' \to 3'$ WK automaton.

Now, considering the accepted language, for each word of the language there is a conjugate in the form $0^n 1^m$ (for $w = 1^i 0^j 1^k$, $n = j$ and $m = i+k$; for $w = 0^i 1^j 0^k$, $n = i+k$ and $m = j$). On the other hand, $M$ is accepting the language $L(M) = \{0^n 1^m \mid n,m \in \mathbb{N}_0\}$ when the first transition is used $n$, the second one $m$ times during the computation. Now, as $cycl(L(M)) = L$, the language $L$ is weakly accepted by $M$. The proof is complete. ●

On the one hand, as all sensing **N1** $5' \to 3'$ WK automata are also sensing **F1** and also sensing **NS** $5' \to 3'$ WK automata, we have obvious inclusion among the (weakly) accepted language classes. On the other hand, we state and shall prove that both of these inclusions are proper.

**Theorem 1** *Each of the classes $\mathscr{W}_{F1}$ and $\mathscr{W}_{NS}$ properly includes the class $\mathscr{W}_{N1}$:*

$$\mathscr{W}_{N1} \subsetneq \mathscr{W}_{F1} \quad \textit{and} \quad \mathscr{W}_{N1} \subsetneq \mathscr{W}_{NS}.$$

**Proof** As the inclusions are obvious by definition, we shall prove only their properness.

Let us consider the first statement and the language $L = \{1^i 0^j 1^k \mid i,j,k \in \mathbb{N}_0,\ j \in \{i+k, i+k+1\}\} \cup \{0^i 1^j 0^k \mid i,j,k \in \mathbb{N}_0,\ i+k \in \{j, j+1\}\}$. It can weakly be accepted by a sensing **F1** $5' \to 3'$ WK automaton: Let $M = (\{0,1\},\{q,p\},q,\{q,p\},\delta)$ with two allowed transitions $p \in \delta(q,0,\lambda)$ and $q \in \delta(p,\lambda,1)$. Notice that for each word in $L$ there is a conjugate in the form $0^n 1^m$ with the condition that either $n = m$ or $n = m+1$. However, $M$ is accepting exactly the language $L' = \{0^n 1^m \mid n \in \{m, m+1\}\}$, and thus weakly accepting $L = cycl(L')$.

To complete the proof of the first part, we should show that $L$ cannot be weakly accepted by any sensing **N1** $5' \to 3'$ WK automata. This part of the proof goes by contradiction: Suppose that $L$ is weakly accepted by a sensing **N1** $5' \to 3'$ WK automaton $M'$ with the sole state $r$ and transition mapping $\delta'$. As $0 = 0^1 1^0 \in L$, $M'$ must have at least one of the loop-transitions $r \in \delta'(r,0,\lambda)$ and $r \in \delta'(r,\lambda,0)$. However, in either case, all words (and thus all necklaces) of $0^*$ would be (weakly) accepted. However, no words (necklaces) of $0^*$ other than $\lambda$ and $0$ are in the language. This contradiction proves that there is no sensing **N1** $5' \to 3'$ WK automaton that weakly accepts $L$, thus, the first statement of the theorem has been proven.

Considering the second statement, let us consider the language $L'' = \{1^i 0^j 1^k \mid i,j,k \in \mathbb{N}_0, j \text{ is even}\} \cup \{0^i 1^j 0^k \mid i,j,k \in \mathbb{N}_0, i+k \text{ is even}\}$. On the one hand, we show that $L''$ is weakly accepted by a sensing

**NS** $5' \to 3'$ WK automaton. Thus, let $M'' = (\{0,1\},\{q\},q,\{q\},\delta'')$ with two allowed transitions $q \in \delta''(q,00,\lambda)$ and $q \in \delta''(q,\lambda,1)$. Then the language accepted by $M''$ is $L(M'') = \{0^{2n}1^m \mid n,m \in \mathbb{N}_0\}$, and its cyclic closure $cycl(L(M'')) = L''$, i.e., there is an even number of 0s such that either they are next to each other, or they form the prefix and suffix of the word in $L''$. Now, on the other hand, we shall prove that $L''$ is not weakly accepted by any sensing **N1** $5' \to 3'$ WK automata. To show this, notice that $00 \in L''$, but $0 \notin L''$. However, a sensing **N1** $5' \to 3'$ WK automaton must read the input letter by letter, and each already read part must also form an accepted word, thus to accept 00, the automaton must read a 0 in the first step of the computation, however, then 0 would also be accepted. In this way the proper inclusion of the second statement has also been proven. •

**Theorem 2** *Each of the classes $\mathscr{W}_N$ and $\mathscr{W}_{FS}$ properly includes the class $\mathscr{W}_{NS}$:*

$$\mathscr{W}_{NS} \subsetneq \mathscr{W}_N \quad \textit{and} \quad \mathscr{W}_{NS} \subsetneq \mathscr{W}_{FS}.$$

**Proof** Let us start with the first statement and consider the necklace language $L = \{0^i1^j0^k \mid$ there exist $n,m \in \mathbb{N}_0$ such that $i+k = 2n+m, j = 2m+n\} \cup \{1^i0^j1^k \mid$ there exist $n,m \in \mathbb{N}_0$ such that $i+k = 2n+m, j = 2m+n\}$. Now, on the one hand, the automaton $M = (\{0,1\},\{q\},q,\{q\},\delta)$ with two transitions $q \in \delta(q,0,11)$ and $q \in \delta(q,00,1)$ is weakly accepting $L$, as each element of $L$ has a conjugate $0^r1^s$ with $n,m \in \mathbb{N}_0$ such that $r = n+2m$ and $s = 2n+m$, where in fact $n$ and $m$ are the numbers of the computation steps made by the two possible transitions, respectively. Further, it is easy to see that $M$ is a sensing **N** $5' \to 3'$ WK automaton. Now, on the other hand, it shall be shown that $L$ is not weakly accepted by any sensing **NS** $5' \to 3'$ WK automaton. This part of the proof is by contradiction, thus let us assume that there is such **NS** automaton $M'$ that weakly accepts $L$. As at least one of the words $011, 101, 110$ is accepted by $M'$ (with sole state $p$ and transition mapping $\delta'$) to include this necklace in the language, the automaton $M'$ must have at least one of the following six transitions: $p \in \delta'(p,011,\lambda)$, $p \in \delta'(p,101,\lambda)$, $p \in \delta'(p,110,\lambda)$, $p \in \delta'(p,\lambda,011)$, $p \in \delta'(p,\lambda,101)$ and $p \in \delta'(p,\lambda,110)$. However, now by applying the same transition in three consecutive computation steps, it leads to accept the following word: 011011011, 101101101, 110110110, 011011011, 101101101 or 110110110 respectively to the six cases. As all these words contain more than two 'blocks' of 0's, clearly none of them is in $L$, thus this contradicts to the fact that $M'$ weakly accepts $L$. Therefore, the language $L$ cannot be weakly accepted by any sensing **NS** $5' \to 3'$ WK automata, completing the proof of the first statement.

Now, let us consider the second statement with the witness language $L = \{1^i0^j1^k \mid i,j,k \in \mathbb{N}_0,\ j \in \{i+k, i+k+1\}\} \cup \{0^i1^j0^k \mid i,j,k \in \mathbb{N}_0,\ i+k \in \{j, j+1\}\}$ used in the proof of the previous theorem. Clearly, the sensing **F1** $5' \to 3'$ WK automaton given there is also a sensing **FS** $5' \to 3'$ WK automaton. On the other hand, as one needs to have one of the transitions to accept the word 0 as it is written in the previous proof, every sensing **N** $5' \to 3'$ WK automaton must also accept words like 00 and 000 which are not in $L$ (and not any of their conjugates are in $L$). This contradiction proves the properness of the inclusion in the second statement. •

**Theorem 3** *The class $\mathscr{W}_{FS}$ properly includes the class $\mathscr{W}_{F1}$:*

$$\mathscr{W}_{F1} \subsetneq \mathscr{W}_{FS}.$$

**Proof** As the inclusion is trivial by definition, we need to show only its properness. Let us consider the witness language $L$ defined by the regular expression $(11)^*$. $L$ contains all words over the unary alphabet with even length. Now, on the one hand, let $M = (\{1\},\{p\},p,\{p\},\delta)$ be a sensing **FS** $5' \to 3'$ WK automaton (in fact also **NS** and **N**) with the only transition $p \in \delta(p,11,\lambda)$. Clearly, $L(M) = L_w(M) = L$.

On the other hand, we need to show that no sensing **F1** $5' \to 3'$ WK automata can (weakly) accept $L$. As all states of **F1** automata are final and they should read the input letter by letter, there must be a configuration when only 1 is read in the accepting computation of, e.g., 11. As the state of this configuration must also be accepting, 1 is also accepted (and weakly accepted) by any **F1** automata that are able to accept 11. As $1 \notin L$, this leads to a contradiction, thus there is no sensing **F1** $5' \to 3'$ WK automata that weakly accept $L$. ●

**Theorem 4** *The class $\mathscr{W}_F$ properly includes the class $\mathscr{W}_N$:*

$$\mathscr{W}_N \subsetneq \mathscr{W}_F.$$

**Proof** On the one hand, the inclusion is trivial by definition. On the other hand, for the properness, let us consider the witness language the regular language $L = 0^* + 0^*10^*$ which is also a necklace language. Let $M = (\{0,1\},\{p,q\},p,\{p,q\},\delta)$ with three transitions $p \in \delta(p,0,\lambda)$, $q \in \delta(p,1,\lambda)$ and $q \in \delta(q,0,\lambda)$, then $L(M) = L_w(M) = L$, moreover $M$ is a sensing **F** $5' \to 3'$ WK automaton (in fact it is also **FS** and **F1**). To show the properness, we need to show that there is no sensing **N** $5' \to 3'$ WK automaton that weakly accepts $L$. The proof goes by contradiction, thus let us assume that $M'$ is an automaton with its sole state $q'$ and transition mapping $\delta'$ such that $L_w(M') = L$. As the word (and also a necklace) 1 is accepted, $M'$ must have at least one of the transitions $q' \in \delta'(q',1,\lambda)$ and $q' \in \delta'(q',\lambda,1)$. However, in either case, the words (and necklaces) 11 and 111 are also accepted. However, as they are not in $L$, we have reached a contradiction. This contradiction shows that $L$ is not weakly accepted by any sensing **N** $5' \to 3'$ WK automata, and the proof is complete. ●

**Theorem 5** *The class $\mathscr{W}_F$ properly includes the class $\mathscr{W}_{FS}$:*

$$\mathscr{W}_{FS} \subsetneq \mathscr{W}_F.$$

**Proof** We need to show only the properness, thus let us have the witness language $L = \{0^i1^j0^k \mid j = i+k\} \cup \{1^i0^j1^k \mid i+k = j\}$. $L$ is the cyclic closure of the linear context-free language $\{0^n1^n \mid n \in \mathbb{N}_0\}$. Clearly, as the automaton $(\{0,1\},\{q\},q,\{q\},\delta)$ with a sole transition $q \in \delta(q,0,1)$ accepts the above mentioned linear context-free language, it also weakly accepts $L$. This automaton is a sensing **N** $5' \to 3'$ WK automaton, and thus, it is also a sensing **F** $5' \to 3'$ WK automaton. Thus, we need to show only that $L$ cannot be weakly accepted by any sensing **FS** $5' \to 3'$ WK automaton. The proof is by contradiction, thus let us assume that $M$ is a sensing **FS** $5' \to 3'$ WK automaton such that $L_w(M) = L$. For each WK automaton, as its transition function gives nonempty sets only for finitely many triplets, there is a maximal length of strings that can be read in a computation step. Let $r$ be this maximal length for automaton $M$. Let us consider the word $w = 0^{3r}1^{3r} \in L$. Since the length of $w$ is large, $M$ needs more than three computation steps to accept one of its conjugates, let us say $u = 0^i1^{3r}0^{3r-i}$ (or symmetrically, $v = 1^j0^{3r}1^{3r-j}$; in this latter case the proof is analogous to the case we present here for $u$). Now, on the one hand, as $M$ is **S** $5' \to 3'$ WK automaton, exactly one of the heads can move in each computation step, thus always a prefix or a suffix of the (remaining) input is processed (and as the input must be processed, there must be computation steps by reading the input). On the other hand, $M$ is also **F** $5' \to 3'$ WK automaton, thus any computation step leads to the acceptance of the word composed by the already read prefix and suffix of the input. Therefore, there are two cases.

If the prefix, let us say $x$ is read in the first step (when input letter is processed), then the prefix $x$ of $u$ must also be in $L$, thus it must also contain at least one occurrence of 0s and also of 1s: $x = 0^i1^i$ $(i < r)$ must hold, and the remaining input is $1^{3r-i}0^{3r-i}$ (where $3r - i > 2r$). Now, in the next step (of the accepting computation of $u$ when some input letters are processed) again a prefix or a suffix of the remaining input is read, however, both the block of 0s and 1s are so large that either only 1s are read (prefix case) or only 0s are read (suffix case). Both lead to the acceptance of some words and necklaces where the number of 0s and 1s mismatch, and thus this leads to a contradiction.

In the second case, if the suffix $y$ of $u$ is read in the first computation step (of the accepting computation of $u$, when at least one letter is processed), then as $y$ is accepted by $M$, $y \in L$ must also hold, and thus $y$ must contain also both 0 and 1: $y = 1^{3r-i}0^{3r-i}$ (and in this case $i > r$). The remaining input after this step is $0^i1^i$. Now, by the second step (of the computation consuming input letter(s)), either the prefix or the suffix of this remaining input is read, but with length at most $r$, meaning that either only 0s or only 1s can be read. But this would lead again to an acceptance of a word (and thus to the weak acceptance of a necklace) that has mismatching numbers of 0s and 1s. This fact contradicts to our assumption, hence $L$ cannot be weakly accepted by any sensing **FS** $5' \to 3'$ WK automata and thus the proof is complete. •

Finally, we present our last hierarchy result of the section by showing that all-final automata are weaker than the unrestricted variants in the term of weakly accepting language classes.

**Theorem 6** *The class $\mathscr{W}_*$ properly includes the class $\mathscr{W}_F$:*

$$\mathscr{W}_F \subsetneq \mathscr{W}_*.$$

**Proof** Again, we need to prove only properness. Consider the witness language $L = \{0^i10^n10^j \mid n \in \mathbb{N}, i, j \in \mathbb{N}_0,\ i + j = n\}$. As $L$ is the cyclic closure of the linear language $\{0^n10^n1 \mid n \in \mathbb{N}\}$, it is in $\mathscr{W}_*$. Now, on the other hand, we show that there is no sensing **F** $5' \to 3'$ WK automaton which weakly accepts $L$. The proof is by contradiction. Thus, let us assume that the language $L$ is weakly accepted by a sensing **F** $5' \to 3'$ WK automaton, say $M$. For each WK automaton, as its transition function gives nonempty sets only for finitely many triplets, there is a maximal length of strings that can be read in a computation step. Let $r$ be this maximal length for automaton $M$. Let us consider the necklace $w_\circ = (0^{2r}10^{2r}1)_\circ \subset L$. In any of the conjugates of $0^{2r}10^{2r}1$, the distance of the two occurrences of 1s is $2r$ implying that at most one of them can be read in the first step of the computation. However, as $M$ is all-final, each computation step leads to an accepted word, and thus, to a weakly accepted necklace. Therefore, as $L(M)$ must contain a word containing at most one 1, $L_w(M)$ has a necklace containing less than two occurrences of 1 which is contradicting to the assumption that $L_w(M) = L$. •

The hierarchy results of this section will be summarized on a Hasse diagram in the concluding section.

## 4 On strongly accepted necklace language classes

In this section we use the strong acceptance mode, i.e., a necklace is in the accepted language if and only if all of its conjugates are accepted by the automaton. By understanding the acceptance mode, and knowing that sensing $5' \to 3'$ WK automata accept exactly the languages of $\mathscr{L}_{LIN}$ ([17, 21]), we can deduce the following fact.

**Proposition 5** *Let $L \in \mathscr{L}_{LIN}$ be a linear context-free language. The maximal necklace language $L' \subset L$ contains exactly those words (necklaces) for which all conjugates (members) are in L. Then there is a sensing $5' \to 3'$ WK automaton M that accepts L, further, for this automaton M, $L_s(M) = L'$.*

*Moreover, the statement hold also in the other direction: Let $M'$ be a $5' \to 3'$ WK automaton. The strongly accepted necklace language $L_s(M')$ is the maximal necklace language L such that $L \subset L(M')$ holds.*

Now we introduce a notion for necklaces. If there is a subword $x$ that occurs in some of the conjugates of $w$, then we say that $x$ is a *pattern* in the necklace $w_\circ$. If this pattern can be written as $x = u'v'$, then we say it fits to the necklace in the (cut) point that defines the conjugate $w'$ in $w_\circ$ such that $u'$ is suffix and $v'$ is a prefix of $w'$. Actually, we can see that one part of $x$ is the prefix and the rest is the suffix of this conjugate. Notice that depending on the length of the pattern there are usually more than one positions where it fits.

We give an example to help the reader to easily catch the concept.

**Example 1** *Let the necklace be defined by the word abcabcaaacb. Then we have a pattern aacba in it, as it is a subword of, e.g., the conjugate bcabcaaacba (especially, it is a suffix here). Now, this pattern fits to the necklace to any points where it occurs, e.g., if we "cut" the necklace to obtain the conjugate cbabcabcaaa, then $u' = aa$ and $v' = cba$, thus our pattern is used as $aa \cdot cba$.*

Because of the special acceptance mode, we have a kind of locally testable property of all these languages. (See [16, 36] for related concepts and language families defined in this way.)

**Proposition 6** *Let $L$ be a necklace language strongly accepted by some $5' \to 3'$ WK automata. Then there is a finite set of patterns such that for each position of the necklace at least one of them must fit.*

**Proof** As, there must be an accepting computation for each conjugate of a word of the language $L$, for every (starting) point, one of the possible transitions from the initial state must match. Let us analyze the case formally. Let $w \in L$ (i.e., $w_\circ \subset L$). Then for each starting point the computation could start, i.e., for each conjugate $w'$ of $w$, there must be a suffix $u'$ and a prefix $v'$ of $w'$ such that there is a transition with them, i.e., $\delta(q_0, v', u') \neq \emptyset$. That means that the pattern $u' \cdot v'$ fits to this cut point of the necklace. On the one hand, there are finitely many possible transitions from the state $q_0$ giving finitely many patterns. On the other hand, for each position at least one of them must match to have an accepting computation for that conjugate. ●

In some special cases, e.g., if the heads read the same length subwords in each transition, the relation with some classes of locally testable languages can be more immediate.

On the other hand, the property stated in the previous proposition must hold for each language in $\mathscr{S}_*$, but for some languages there could be more (meaning more complex) restrictions as we can see later.

Now we turn to present some hierarchy results among the corresponding necklace language classes. As the very first result in this line, we show that even the most restricted class is not empty, i.e., there are languages in $\mathscr{S}_{N1}$. Actually, we show more, we give a full characterization of this class.

**Theorem 7** *A necklace language $L$ is in $\mathscr{S}_{N1}$ if and only if $L = T_1^* \cup T_2^*$ for two alphabets $T_1, T_2$.*

**Proof** The proof goes by two parts. First we show that every language of the form $T_1^* \cup T_2^*$ for two alphabets $T_1$ and $T_2$ is in $\mathscr{S}_{N1}$. By considering $T_1 = \{a_1, \ldots, a_n\} \cup \{c_1, \ldots, c_m\}$ and $T_2 = \{b_1, \ldots, b_k\} \cup \{c_1, \ldots, c_m\}$, let us define the automaton $M = (T, \{q\}, q, \{q\}, \delta)$ with $\delta(q, x, \lambda) = \{q\}$ for each $x \in T_1$ and $\delta(q, \lambda, y) = \{q\}$ for each $y \in T_2$. (For any other triplets let $\delta$ give the empty set.) Clearly, $M$ is a sensing **N1** $5' \to 3'$ WK automaton. Moreover, $M$ accepts $T_1^*$ if only the first head is used during the computation and $T_2^*$ if only the second head is used during the computation. Now, we show that there is no necklace that can be accepted such that both heads must be used. Contrary, let us assume that there is a necklace $w_\circ$ which contains letters from both $T_1 \setminus T_2$ and $T_2 \setminus T_1$, then there is a pattern $a_i b_j$ in $w_\circ$, i.e., it has a conjugate $b_j u a_i$ (with some $u \in (T_1 \cup T_2)^*$). However, there is no transition defined in $M$ to start the computation for this conjugate, thus this necklace cannot be accepted. Finally, as $T_1^* \cup T_2^*$ is a necklace language itself, the maximal necklace language in it is also itself, thus $M$ accepts the necklace language $T_1^* \cup T_2^*$ in strong acceptance mode.

Actually, every sensing **N1** $5' \to 3'$ WK automaton can be described by two (maybe not disjoint) sets $T_1$ and $T_2$ of letters having transitions $\delta(q,a_i,\lambda) = \{q\}$ for each $a_i \in T_1$ and $\delta(q,\lambda,b_j) = \{q\}$ for each $b_j \in T_2$. Then, with a similar argument as we used above, one can see that the language $T_1^* \cup T_2^*$ is accepted, and actually, for each accepted word there is a computation where only one of the heads is used to read the entire input. No input can be accepted that has letters that cannot be read by the same head. ●

Now, we present some hierarchy results among various classes of strictly accepted necklace languages.

**Theorem 8** *The class $\mathscr{S}_{NS}$ properly includes the class $\mathscr{S}_{N1}$:*

$$\mathscr{S}_{N1} \subsetneq \mathscr{S}_{NS}.$$

**Proof** The inclusion holds by definition, as all **N1** automata are also **NS** automata. To show the properness we give an example. Consider $M = (\{a\},\{q\},q,\{q\},\delta)$ with $\delta(q,aa,\lambda) = \{q\}$ and $\delta$ gives the empty set for any other triplets. It is easy to see that both the accepted and the strongly accepted language is $(aa)^*$ which cannot be accepted by any **N1** $5' \to 3'$ WK automaton as we have shown in Theorem 7. ●

**Lemma 1** *Let L be a language strongly accepted by a sensing **NS** $5' \to 3'$ WK automaton. If it contains a nonempty word $a^n$ with some $a \in T$ and $n \in \mathbb{N}$, then it contains all words of $(a^n)^*$.*

**Proof** Any word of the form $a^n$ can be considered as a singleton necklace. Further, as such automaton has only one state, the same computation steps as the ones lead to the acceptance of $a^n$ can be repeated if the input is longer. In this way, each word of $(a^n)^*$ is accepted, thus the language is infinite. ●

**Lemma 2** *Let L be a language of necklaces strongly accepted by a sensing **F1** $5' \to 3'$ WK automaton. If L contains a nonempty word, then it contains one letter long word(s).*

**Proof** In a sensing **F1** $5' \to 3'$ WK automaton all states are accepting, and the automaton can read exactly one letter in the first step of the computation. Thus, if it has any transition from the initial state, it will accept the one letter long word containing the letter of the transition. As every one letter long word itself is a singleton necklace, it is also strongly accepted, thus it appears in the strongly accepted necklace language. W.l.o.g., assume that there is a transition with letter $a \in T$ with the first head in $M$, i.e., $\delta(q_0,a,\lambda) \neq \emptyset$. Then $a \in L_s(M)$. ●

**Theorem 9** *The class $\mathscr{S}_{F1}$ properly includes the class $\mathscr{S}_{N1}$:*

$$\mathscr{S}_{N1} \subsetneq \mathscr{S}_{F1}.$$

**Proof** The inclusion holds by definition, as all **N1** automata are also **F1** automata. To show the properness we give an example. Consider $M = (\{a,b\},\{q,p,r\},q,\{q,p,r\},\delta)$ with $\delta(q,a,\lambda) = \{p\}$ and $\delta(q,b,\lambda) = \{r\}$ (where $\delta$ gives the empty set for any other triplets). It is easy to see that both the accepted and the strongly accepted language is $\{\lambda,a,b\}$ which cannot be accepted by any **N1** $5' \to 3'$ WK automaton. ●

**Theorem 10** *The class $\mathscr{S}_{FS}$ properly includes both of the classes $\mathscr{S}_{F1}$ and $\mathscr{S}_{NS}$:*

$$\mathscr{S}_{F1} \subsetneq \mathscr{S}_{FS} \quad \text{and} \quad \mathscr{S}_{NS} \subsetneq \mathscr{S}_{FS}.$$

**Proof** The inclusions hold by definition, as all **F1** automata and all **NS** automata are also **FS** automata. To show the properness we give an example. Consider $M = (\{a,b\},\{q,p\},q,\{q,p\},\delta)$ with $\delta(q,aa,\lambda) = \{p\}$, $\delta(q,ab,\lambda) = \{p\}$ and $\delta(q,ba,\lambda) = \{p\}$ (where $\delta$ gives the empty set for any other triplets). It is easy to see that both the accepted and the strongly accepted language is $\{\lambda,aa,ab,ba\}$ includes two nonempty necklaces. This language cannot be accepted by any **F1** $5' \to 3'$ WK automaton by Lemma 2 as each of its nonempty words has length 2. Moreover, $L_s(M)$ is a finite language containing the nonempty word $aa$, thus by Lemma 1 it cannot be strongly accepted by any **NS** $5' \to 3'$ WK automaton. ●

The examples we have used so far defined regular languages. To show that the model we are considering here has a larger expressive power, we present the following example, where a non regular (and in fact, not linear context-free) language is defined by an **F1** $5' \to 3'$ WK automaton.

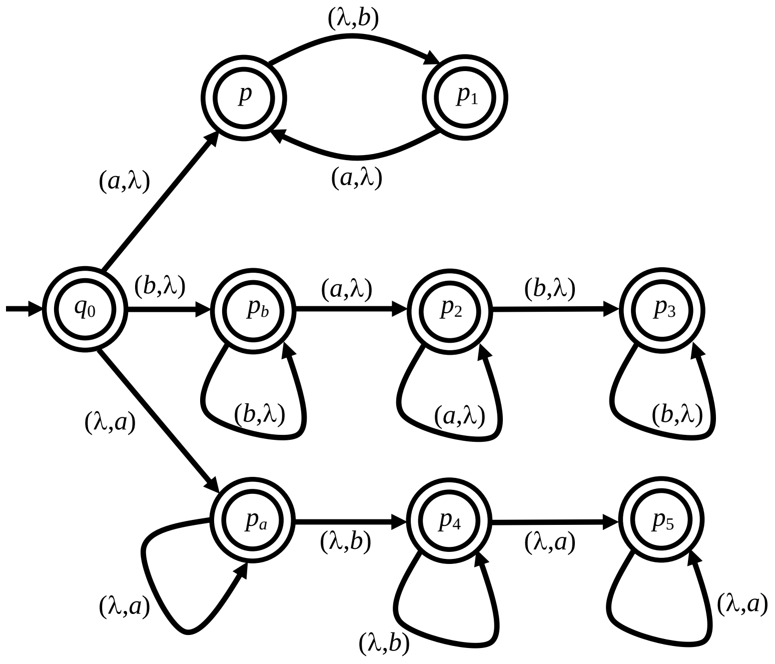


Figure 3: A sensing **F1** $5' \to 3'$ WK automaton that is accepting a non linear context free language of necklaces in the strong mode.

**Example 2** *Consider the sensing* ***F1*** $5' \to 3'$ *WK automaton M shown in Figure 3. Depending on the first letter of the chosen conjugate, the computation follows different ways and also there is computation based on the last letter. If the first letter is b, then state* $p_b$ *is reached, and all continuations belong to* $b^*a^*b^*$ *are accepted. In this way, clearly all words of* $b^*$ *are also strongly accepted, as each of them is a singleton necklace. Whenever, the last letter of the conjugate is an a, there is a computation reaching* $p_a$ *and the computation continues accepting all words of* $a^*b^*a^*$*. Here all necklaces containing only a-s are also accepted, i.e., the elements of* $a^*$ *are in* $L_s(M)$*. If the necklace contains both a and b, then it must also be accepted when conjugate starting with a and finishing with a b (having the pattern* $b \cdot a$ *to fit to this position). However, in this case, the only computation goes from* $q_0$ *to p and continues by using both heads and counting the number of a-s and b-s not to have a larger difference than* 1*. Thus, the strongly* *accepted necklace language is* $\{a^*\} \cup \{b^*\} \cup \{a^n b^n\} \cup \{a^{n+1} b^n\} \cup \{b^k a^n b^m \mid n \in \{k+m, k+m+1\}\}$*. This language is not regular, moreover, it is not linear. On the other hand, it is context-free as a PDA can easily count the number of letters in each of the possible conjugates.*

In [28] it was proven that exactly the class $\mathscr{L}_{LIN}$ of linear context-free languages are accepted by each of the classes of (arbitrary, i.e., unrestricted) sensing $5' \to 3'$ WK automata, of sensing **S** $5' \to 3'$ WK automata and of sensing **1** $5' \to 3'$ WK automata. By considering these automata for necklaces in the strong acceptance mode, we have the following consequence on the top of the hierarchy.

**Proposition 7**

$$\mathscr{S}_* = \mathscr{S}_S = \mathscr{S}_1 \supset \mathscr{S}_F.$$

We leave open whether the hierarchy is proper or not for the pair of classes we did not show proofs. A summary of these results can also be seen in the Hasse diagram in Figure 5 in the next section.

$$\begin{tikzcd}
& \mathscr{L}_{context-free} & \\
& \mathscr{W}_* = \mathscr{W}_S = \mathscr{W}_1 = cycl(\mathscr{L}_{LIN}) \arrow[u] & \\
& & \mathscr{W}_F \arrow[ul] \\
\mathscr{W}_N \arrow[urr] & & \mathscr{W}_{FS} \arrow[u] \\
\mathscr{W}_{NS} \arrow[u] \arrow[urr] & & \mathscr{W}_{F1} \arrow[u] \\
& \mathscr{W}_{N1} \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

Figure 4: Hierarchy of necklace languages weakly accepted by sensing $5' \to 3'$ WK finite automata in a Hasse diagram. Each of the shown inclusions is proper.

## 5 Conclusions

Necklaces (or circular words) may represent various real word objects, e.g., DNA molecules having circular (also called) cyclic structure. In mathematics and computer science they are often modeled by

$$\begin{tikzcd}
 & & \mathscr{S}_* = \mathscr{S}_S = \mathscr{S}_1 \\
 & & \mathscr{S}_F \arrow[u, dash] \\
\mathscr{S}_N \arrow[urr, dash] & & \mathscr{S}_{FS} \arrow[u, dash] \\
\mathscr{S}_{NS} \arrow[u, dash] \arrow[urr] & & \mathscr{S}_{F1} \arrow[u] \\
 & \mathscr{S}_{N1} \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

Figure 5: Hierarchy of necklace languages strongly accepted by sensing $5' \to 3'$ WK finite automata in a Hasse diagram. Arrows represent proper inclusions, while lines represent inclusions where the properness is left open.

the set of conjugates, i.e., linear (ordinary) words that could be the base of the cycle. In this paper, we used WK automata to accept necklaces and necklace languages. Two acceptance modes have been investigated, if at least one of the elements of the conjugate class is accepted, then the corresponding necklace is weakly accepted, while in case all conjugates are accepted, the necklace is strongly accepted. Based on the various restrictions of WK automata, we established hierarchies of the accepted language classes. We summarize these hierarchy results obtained for necklace languages by Hasse diagrams and we also list a few open problems.

On the first hand, a Hasse diagram shows the hierarchy of the weakly accepted classes of necklace languages in Figure 4. On the other hand, Figure 5 shows the Hasse diagram of the language classes of the strongly accepted necklace languages. Here, some of the inclusions are trivial by definition and their properness are left open. More precisely, the relations (equality or proper inclusion) between the following classes is open:

- $\mathscr{S}_{NS} - \mathscr{S}_N$,
- $\mathscr{S}_{FS} - \mathscr{S}_F$,
- $\mathscr{S}_N - \mathscr{S}_F$, and
- $\mathscr{S}_F - \mathscr{S}_*$.

Further open problems are, e.g., the closure properties of the newly defined language classes. Relations to other families of languages, including locally testable families are also planned to be established in the near future.

## References

[1] Leonard M. Adleman (1994): *Molecular computation of solutions to combinatorial problems, Science* 226, pp. 1021–1024, doi:10.1126/science.7973651.

[2] Amar, V., Putzolu, G.R. (1964): *On a family of linear grammars. Inf. Control* 7(3), 283–291, doi:10.1016/S0019-9958(64)90294-3.

[3] Andreas Brandstädt (1981): *Closure Properties of Certain Families of Formal Languages with Respect to a Generalization of Cyclic Closure. RAIRO Theor. Informatics Appl.* 15(3), pp. 233–252, doi:10.1051/ita/1981150302331.

[4] Elena Czeizler & Eugen Czeizler (2006): *A Short Survey on Watson-Crick Automata, Bulletin of the EATCS* 88, pp. 104–119.

[5] Rudolf Freund, Gheorghe Păun, Grzegorz Rozenberg & Arto Salomaa (1997): *Watson-Crick finite automata*. In: Harvey Rubin & David Harlan Wood, editors: *DNA Based Computers, Proceedings of a DIMACS Workshop, Philadelphia, Pennsylvania, USA, June 23-25, 1997, DIMACS Series in Discrete Mathematics and Theoretical Computer Science* 48, DIMACS/AMS, pp. 297–327, doi:10.1090/dimacs/048/22.

[6] László Hegedüs & Benedek Nagy (2013): *Periodicity of circular words. In: WORDS 2013, Turku, Finland, TUCS Lecture Notes No.* 20 (09.2013), pp. 45–56.

[7] László Hegedüs & Benedek Nagy (2016): *On periodic properties of circular words. Discrete Mathematics* 339(3), pp. 1189–1197, doi:10.1016/j.disc.2015.10.043.

[8] John E. Hopcroft & Jeffrey D. Ullman (1979): *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Reading, M.A.

[9] Ondrej Klíma & Libor Polák (2011): *On Biautomata*. In Rudolf Freund, Markus Holzer, Carlo Mereghetti, Friedrich Otto, Beatrice Palano (eds.): *Third Workshop on Non-Classical Models for Automata and Applications – NCMA 2011, Milan, Italy, July 18 - July 19, 2011. Proceedings. books@ocg.at* 282, Austrian Computer Society, pp. 153–164.

[10] Radim Kocman, Zbynek Krivka, Alexander Meduna & Benedek Nagy (2022): *A jumping $5' \to 3'$ Watson-Crick finite automata model. Acta Informatica* 59(5), pp. 557–584, doi:10.1007/s00236-021-00413-x

[11] Manfred Kudlek (2004): *On languages of cyclic words*. In: Natasa Jonoska, Gheorghe Păun, Grzegorz Rozenberg (eds.): *Aspects of Molecular Computing, Essays Dedicated to Tom Head on the Occasion of His 70th Birthday. Lecture Notes in Computer Science, LNCS* 2950, pp. 278–288, doi:10.1007/978-3-540-24635-0_20.

[12] Dietrich Kuske & Peter Weigel (2004): *The role of the complementarity relation in Watson-Crick automata and sticker systems*. In: Cristian S. Calude, Elena Calude & Michael J. Dinneen (editors): *Developments in Language Theory, DLT 2004, Lecture Notes in Computer Science, LNCS* 3340, Springer, Berlin, Heidelberg, pp. 272–283. doi:10.1007/978-3-540-30550-7 23.

[13] Peter Leupold & Benedek Nagy (2009): $5' \to 3'$ Watson-Crick automata with several runs. In: Henning Bordihn, Rudolf Freund, Markus Holzer, Martin Kutrib, Friedrich Otto (eds.): Workshop on Non-Classical Models for Automata and Applications - NCMA 2009, Wroclaw, Poland, August 31 - September 1, 2009. Proceedings. books@ocg.at 256, Austrian Computer Society 2009, pp. 167–180.

[14] Peter Leupold & Benedek Nagy (2010): $5' \to 3'$ *Watson-Crick automata with several runs, Fundamenta Informaticae* 104, pp. 71–91, doi:10.3233/FI-2010-336.

[15] Roussanka Loukanova (2007): *Linear context free languages*. In: Cliff B. Jones, Zhiming Liu, Jim Woodcock (eds.): *Theoretical Aspects of Computing - ICTAC 2007, 4th International Colloquium, Macau, China,*

*September 26-28, 2007, Proceedings. Lecture Notes in Computer Science* 4711, Springer 2007, pp. 351–365, doi:10.1007/978-3-540-75292-9_24.

[16] Robert McNaughton & Seymour Papert (1971): *Counter-Free Automata*. MIT Press.

[17] Benedek Nagy (2008): *On $5' \to 3'$ sensing Watson-Crick finite automata*, In: Garzon M.H. & Yan H. (eds.): *DNA Computing. DNA 2007: Selected revised papers, Lecture Notes in Computer Science, LNCS* 4848, Springer, Berlin, Heidelberg, pp. 256–262. doi:10.1007/978-3-540-77962-9_27.

[18] Benedek Nagy (2009): *On a hierarchy of $5' \to 3'$ sensing WK finite automata languages*, In: *Computaility in Europe, CiE 2009: Mathematical Theory and Computational Practice, Abstract Booklet, Heidelberg*, pp. 266–275.

[19] Benedek Nagy (2010): *$5' \to 3'$ sensing Watson-Crick finite automata*, In: Gabriel Fung (ed.): *Sequence and Genome Analysis II - Methods and Applications*, pp. 39—56, iConcept Press.

[20] Benedek Nagy (2012): *A class of 2-head finite automata for linear languages. Triangle 8: llenguatge, literatura, computació*, 89–99.

[21] Benedek Nagy (2013): *On a hierarchy of $5' \to 3'$ sensing Watson-Crick finite automata languages, Journal of Logic and Computation* 23(4), pp. 855–872, doi:10.1093/logcom/exr049.

[22] Benedek Nagy (2023): *On language classes accepted by stateless $5' \to 3'$ Watson-Crick finite automata. Annales Mathematicae et Informaticae* 58, pp. 110–120, doi:10.33039/ami.2023.08.004.

[23] Benedek Nagy & Zita Kovács (2021): *On deterministic 1-limited $5' \to 3'$ sensing Watson-Crick finite-state transducers. RAIRO Theor. Informatics Appl.* 55(5) (18 pages), doi:10.1051/ita/2021007.

[24] Benedek Nagy & Friedrich Otto (2011): *Finite-State Acceptors with Translucent Letters, ICAART 2011 - 3rd International Conference on Agents and Artificial Intelligence, BILC 2011 - 1st International Workshop on AI Methods for Interdisciplinary Research in Language and Biology*, pp. 3–13, doi:10.5220/0003272500030013.

[25] Benedek Nagy & Friedrich Otto (2020): *Linear automata with translucent letters and linear context-free trace languages. RAIRO Theor. Informatics Appl.* 54, article number 3 (23 pages), doi:10.1051/ita/2020002.

[26] Benedek Nagy& Shaghayegh Parchami (2021): *On deterministic sensing $5' \to 3'$ Watson-Crick finite automata: a full hierarchy in 2detLIN, Acta Informatica* 58(3), pp. 153–175, doi:10.1007/s00236-019-00362-6.

[27] Benedek Nagy & Shaghayegh Parchami (2022): *$5' \to 3'$ Watson-Crick automata languages-without sensing parameter. Nat. Comput.* 21(4), pp. 679–691, doi:10.1007/s11047-021-09869-9.

[28] Benedek Nagy, Shaghayegh Parchami & Hamid-Mir-Mohammed Sadeghi (2017): *A new sensing $5' \to 3'$ Watson-Crick automata concept*. In *AFL 2017: Proceedings 15th International Conference on Automata and Formal Languages, EPTCS* 252, pp. 195–204, doi:10.4204/EPTCS.252.19.

[29] Shaghayegh Parchami, Benedek Nagy (2018): *Deterministic Sensing $5' \to 3'$ Watson-Crick Automata Without Sensing Parameter*, In Susan Stepney & Sergey Verlan (editors): *UCNC 2018: 17th International Conference on Unconventional Computation and Natural Computation, LNCS* 10867, pp. 173–187, doi:10.1007/978-3-319-92435-9_13.

[30] Gheorghe Păun, Grzegorz Rozenberg & Arto Salomaa (2002): *DNA Computing: New Computing Paradigms*. Springer-Verlag, doi:10.1007/978-3-662-03563-4.

[31] Grzegorz Rozenberg, Thomas Bäck & Joost N. Kok (2012): *Handbook of Natural Computing*. Springer, doi:10.1007/978-3-540-92910-9

[32] Grzegorz Rozenberg & Arto Salomaa, eds., (1997): *Handbook of Formal Languages*. Springer, doi:10.1007/978-3-642-59136-5.

[33] José M. Sempere (2004): *A Representation Theorem for Languages Accepted by Watson-Crick Finite Automata. Bulletin of the EATCS* 83, pp. 187–191.

[34] José M. Sempere (2018): *On the application of Watson-Crick finite automata for the resolution of bioinformatic problems*, In Rudolf Freund, Michal Hospodár, Galina Jirásková & Giovanni Pighizzini, editors: *Tenth Workshop on Non-Classical Models of Automata and Applications, NCMA* 2018, Österreichische Computer Gesellschaft, pp. 29–30. Invited talk.

[35] José M. Sempere & P. García (1994): *A characterization of even linear languages and its application to the learning problem*. In: *ICGI 1994, LNCS/LNAI* 862, pp. 38–44, doi:10.1007/3-540-58473-0_135.

[36] Yechezkel Zalcstein (1972): *Locally testable languages, Journal of Computer and System Sciences* 6(2), pp. 151–167, doi:10.1016/S0022-0000(72)80020-5.

# Complexity Aspects of the Extension of Wagner's Hierarchy to $k$-Partitions

Vladimir Podolskii
Tufts University, Medford, MA, USA
podolskii.vv@gmail.com

Victor Selivanov
Department of Mathematics and Computer Science, St.Petersburg University, Saint Petersburg, Russia
A.P. Erhov Institute of Informatics Systems, Novosibirsk, Russia
vseliv@iis.nsk.su

It is known that the Wadge reducibility of regular $\omega$-languages is efficiently decidable (Krishnan et al., 1995), (Wilke, Yoo, 1995).

In this paper we study analogous problem for regular $k$-partitions of $\omega$-languages. In the series of previous papers (Selivanov, 2011), (Alaev, Selivanov, 2021), (Selivanov, 2012) there was a partial progress towards obtaining an efficient algorithm for deciding the Wadge reducibility in this setting as well. In this paper we finalize this line of research providing a quadratic algorithm (in RAM model). For this we construct a quadratic algorithm to decide a preorder relation on iterated posets.

Additionally, we discuss the size of the representation of regular $\omega$-languages and suggest a more compact way to represent them. The algorithm we provide is efficient for the more compact representation as well.

## 1 Introduction

In [23], K. Wagner has shown that the quotient-poset of the preorder $(\mathscr{R};\leq_W)$ of regular $\omega$-languages under the Wadge reducibility (i.e., $m$-reducibility by continuous functions on the Cantor space of $\omega$-words) is semi-well-ordered with order type $\omega^\omega$, and that the related algorithmic problems are decidable. E.g., given Muller acceptors $\mathscr{A}$ and $\mathscr{B}$, one can effectively solve the relation $L(\mathscr{A}) \leq_W L(\mathscr{B})$ between the corresponding regular $\omega$-languages. Later it was shown that there are efficient algorithms solving such problems [11, 24], in particular the problem $L(\mathscr{A}) \leq_W L(\mathscr{B})$ is solvable in cubic time.

In [16] (see also [19] for detailed proofs), the Wagner theory was extended from the regular sets $A \subseteq X^\omega$ (identified in the usual way with functions $A : X^\omega \to \{0,1\}$) to the regular $k$-partitions $A : X^\omega \to \{0,\dots,k-1\}$ of the set $X^\omega$ of $\omega$-words over a finite alphabet $X$. Motivations for this extension come from the fact that similar objects are important e.g. in computability theory [18], descriptive set theory [9], and complexity theory [10].

The extension from sets to $k$-partitions for $k > 2$ is non-trivial in the sense that the corresponding structure $(\mathscr{R}_k;\leq_W)$ becomes much more complex. Nevertheless, it admits a nice combinatorial characterization in terms of iterated $h$-preorders on labeled forests (terminology is briefly recalled in the next section), and the full extension of Wagner's hierarchy to $k$-partitions is possible; the extension is called the fine hierarchy (FH) of $\omega$-regular $k$-partitions). But the existence of the corresponding *efficient* algorithms for the algorithmic problems (like the extension of $L(\mathscr{A}) \leq_W L(\mathscr{B})$ to Muller's $k$-acceptors [19]) is far from obvious.

In this paper, we address the latter problem. A first step in this direction was made in [1] where, with the use of some previous results from [6], efficient algorithms deciding basic problems about the iterated $h$-preorders on labeled forests were established. This is relevant because levels of the FH of $k$-partitions are naturally denoted by the iterated labeled forests, and manipulations with the levels seem inevitable. But unfortunately, this does not immediately yield an efficient algorithm for solving $L(\mathscr{A}) \leq_W L(\mathscr{B})$ because one needs first to find (from given Muller's $k$-acceptors) the levels of the FH (i.e., the corresponding forests $F$ for $\mathscr{A}$ and $G$ for $\mathscr{B}$) where the $k$-partitions $L(\mathscr{A})$ and $L(\mathscr{B})$ are Wadge complete. In computing $F,G$ from $\mathscr{A},\mathscr{B}$, one can first compute iterated $k$-posets $P,R$ (this computation is feasible), and then to unfold $P,R$ to forests $F = u(P), G = u(R)$ using a natural algorithm first described in [15] and then elaborated in [17]. Unfortunately, the size of the unfolded forests grows exponentially, so on this way it is hopeless to find an efficient algorithm.

A possible solution is suggested by the observation in [17] that it is possible to name levels of the FH of $k$-partitions directly by the iterated poset $P$ instead of the unfolded forest $u(P)$, obtaining thus a more succinct notation system for levels. But then we have to work with the preorder $\preceq$ on posets induced by the preorder $\leq_h$ (i.e., $P \preceq R$ iff $u(P) \leq_h u(R)$), and we cannot directly use the algorithms from [1].

As the main result of this paper we reprove the complexity estimates in [1] directly for posets. This result applies not only to the extension of Wagner's hierarchy but might also be useful in other situations where the FH of $k$-partitions naturally appears (for an example from computability theory see [18]). With this at hand, it is not hard to get the desired efficient algorithms for the extended Wagner hierarchy.

For our algorithms we use the RAM model, which is more standard for studying efficient algorithms than the Turing machine model used in [1]. In this model our algorithm is quadratic and we note that the algorithm from [1] is quadratic as well in the RAM model (see Section 2.4 for details).

Since we are studying efficient algorithms, the size of the representation of the input matters. The straightforward way to represent $k$-acceptors is to provide for each subset of states a label in the partition. However, note that Muller acceptors operate with cycles and not all subsets can be cycles. We observe that actually the number of cycles is polynomially smaller than the number of subsets of states. This suggests another way to represent $k$-acceptors that is more compact and might be useful in some settings. We note that the algorithm we provide is efficient for this type of representation of inputs as well.

After recalling some preliminaries in the next section, in Section 3 we prove the mentioned result on iterated labeled posets. In Section 4 we explain how to deduce efficient algorithms working with a straightforward representation of Muller's $k$-acceptors. In Section 5 we discuss a less straightforward representations of acceptors and translate our algorithm for Muller's $k$-acceptors to a more succinct representation of inputs.

## 2 Preliminaries

We use standard notation and facts about finite automata on infinite words which may be found e.g. in [13, 21]. We work with a fixed finite alphabet $X$ containing more than one letter, and only with deterministic finite automata.

### 2.1 Automata and acceptors

By an *automaton* (over $X$) we mean a triple $\mathscr{M} = (Q, f, in)$ consisting of a finite non-empty set $Q$ of states, a transition function $f : Q \times X \to Q$ and an initial state $in \in Q$. The function $f$ is extended to the function $f : Q \times X^* \to Q$ by induction $f(q, \varepsilon) = q$ and $f(q, u \cdot x) = f(f(q,u),x)$, where $u \in X^*$ and $x \in X$.

Similarly, we may define the function $f : Q \times X^{\omega} \to Q^{\omega}$ by $f(q,\xi)(n) = f(q,\xi\restriction_n)$.

Associate with any automaton $\mathscr{M}$ the set of *cycles* (known also as loops) $C_{\mathscr{M}} = \{f_{\mathscr{M}}(\xi) \mid \xi \in X^{\omega}\}$ where $f_{\mathscr{M}}(\xi)$ is the set of states that occur infinitely often in the sequence $f(in,\xi) \in Q^{\omega}$. A *Muller acceptor* is a pair $(\mathscr{M},\mathscr{F})$ where $\mathscr{M}$ is an automaton and $\mathscr{F} \subseteq C_{\mathscr{M}}$; it recognizes the set $L(\mathscr{M},\mathscr{F}) = \{\xi \in X^{\omega} \mid f_{\mathscr{M}}(\xi) \in \mathscr{F}\}$. The Muller acceptors recognize exactly the *regular $\omega$-languages*.

A $k$-partition $A : X^{\omega} \to \{0,\dots,k-1\}$ is *regular*, if any its component $A_i = A^{-1}(i)$, $i < k$, is regular. Regular $k$-partition $A$ may be represented by $k$-tuples of Muller acceptors which recognize the components of $A$, but we will use a slightly different kind of acceptors introduced in [16]. A *Muller $k$-acceptor* is a pair $(\mathscr{M},A)$ where $\mathscr{M}$ is an automaton and $A : C_{\mathscr{M}} \to k$ is a $k$-partition of $C_{\mathscr{M}}$. The Muller $k$-acceptor $(\mathscr{M},A)$ recognizes the $k$-partition $L(\mathscr{M},A) = A \circ f_{\mathscr{M}}$ where $f_{\mathscr{M}} : X^{\omega} \to C_{\mathscr{M}}$ is defined above.

Note that the Muller 2-acceptors are equivalent to Muller acceptors though syntactically they are slightly different beecause, along with the set $\mathscr{F}$ of accepting cicles a Muller 2-acceptor also contains its complement $C_{\mathbf{M}} \setminus \mathscr{F}$. This causes some distinctions of our complexity estimates for $k = 2$ from those in [11, 24].

## 2.2 Iterated $k$-posets

Next we recall some information about the iterated $h$-preorder and its variants; for additional information see e.g. [18, 19]. Let $(P;\leq)$ be a finite poset; if $\leq$ is clear from the context, we simplify the notation of the poset to $P$. Any subset of $P$ may be considered as a poset with the induced partial ordering. By a *forest* we mean a finite poset in which every lower cone $\downarrow x$, $x \in P$, is a chain. A *tree* is a forest with the least element (called the *root* of the tree).

Let $(Q;\leq)$ be a preorder. A *$Q$-poset* is a triple $(P,\leq,c)$ consisting of a finite nonempty poset $(P;\leq)$, $P \subseteq \omega$, and a labeling $c : P \to Q$. Let $\mathscr{P}_Q$, $\mathscr{F}_Q$, and $\mathscr{T}_Q$ denote the sets of all finite $Q$-posets, $Q$-forests, and $Q$-trees, respectively. For the particular case $Q = \overline{k} = \{0,\cdots,k-1\}$ of antichain with $k$ elements we denote the corresponding $h$-preorders by $\mathscr{P}_k$, $\mathscr{F}_k$, and $\mathscr{T}_k$. A *morphism* $f : (P,\leq,c) \to (P',\leq',c')$ between $Q$-posets is a monotone function $f : (P;\leq) \to (P';\leq')$ satisfying $\forall x \in P\,(c(x) \leq c'(f(x)))$. The *$h$-preorder* $\leq_h$ on $\mathscr{P}_Q$ is defined as follows: $P \leq_h P'$, if there is a morphism $f : P \to P'$.

For any $q \in Q$ let $s(q) \in \mathscr{T}_Q$ be the singleton tree labeled by $q$; then $q \leq r$ iff $s(q) \leq_h s(r)$. Identifying $q$ with $s(q)$, we may think that $Q$ is a substructure of $\mathscr{T}_Q$. The quotient-poset of $(\mathscr{F}_Q;\leq_h,\sqcup)$ is a semilattice where the supremum operation is induced by the disjoint union $F \sqcup G$ of $Q$-forests $F,G$. The semilattice is generated by the join-irreducible elements induced by trees. The set $\mathscr{F}_Q$ may be identified with the set $\mathscr{T}_Q^{\sqcup}$ of finite disjoint unions of trees.

For any finite $Q$-poset $(P,c)$ there exist a finite $Q$-forest $(F,d)$ and a morphism $f$ from $F$ onto $P$ such that $F$ is a largest element in $(\{G \in \mathscr{F}_Q \mid G \leq_h P\};\leq_h)$. The forest $F = u(P)$ is constructed by a natural bottom-up unfolding of $P$ (for additional details see [15] and sections 7,8 of [19]). The unfolding operator $u : \mathscr{P}_Q \to \mathscr{F}_Q$ gives rise to a preorder $\preceq$ on $\mathscr{P}_Q$ (already mentioned in the introduction) defined by $P \preceq R \leftrightarrow u(P) \leq_h u(R)$. Note that $P \leq_h R$ implies $P \preceq R$ (but the converse fails in general), and that both relations coincide on $\mathscr{F}_Q$.

Define the sequence $\{\mathscr{T}_k(n)\}_{n<\omega}$ of preorders by induction on $n$ as follows: $\mathscr{T}_k(0) = \overline{k}$ and $\mathscr{T}_k(n+1) = \mathscr{T}_{\mathscr{T}_k(n)}$. The sets $\mathscr{T}_k(n)$, $n < \omega$, are pairwise disjoint but, identifying the elements $i$ of $\overline{k}$ with the corresponding singleton trees $s(i)$ labeled by $i$ (which are precisely the minimal elements of $\mathscr{T}_k(1)$), we may think that $\mathscr{T}_k(0) \sqsubseteq \mathscr{T}_k(1)$, i.e. the quotient-poset of the first preorder is an initial segment of the quotient-poset of the second. This also induces an embedding of $\mathscr{T}_k(n)$ into $\mathscr{T}_k(n+1)$ as an initial segment, so (abusing notation) we may think that $\mathscr{T}_k(0) \sqsubseteq \mathscr{T}_k(1) \sqsubseteq \cdots$.

Let $\mathscr{T}_k(\omega) = \bigcup_{n<\omega} \mathscr{T}_k(n)$; the induced preorder on this set is again denoted by $\leq_h$. We often simplify $\mathscr{T}_k(n)^{\sqcup}$ to $\mathscr{F}_k(n)$; in particular, $\mathscr{F}_k(2) = \mathscr{F}_{\mathscr{T}_k}$. The embedding $s$ is extended to $\mathscr{T}_k(\omega)$ by defining $s(T)$ as the singleton tree labeled by $T$.

Similar iterations are possible for other aforementioned constructions. E.g., we can define iterations of the construction $Q \mapsto \mathscr{V}_Q$ where $\mathscr{V}_Q$ is the set of pointed posets from $\mathscr{P}_Q$ (i.e., posets with a smallest element) ordered by the relation $\preceq$ (rather than by $\leq_h$). In this way, we obtain the sequence $\{\mathscr{V}_k(n)\}_{n\leq\omega}$.

The aforementioned unfolding operator $u : \mathscr{P}_Q \to \mathscr{F}_Q$ is naturally extended and modified to operators $u : \mathscr{V}_k(n) \to \mathscr{T}_k(n)$ and $u : \mathscr{V}_k(n)^{\sqcup} \to \mathscr{T}_k(n)^{\sqcup}$ for each $n \leq \omega$ which have properties similar to those of the basic operator $u : \mathscr{P}_Q \to \mathscr{F}_Q$ (cf. Lemma 8.7 in [19]). Especially relevant to this paper is the unfolding $u : \mathscr{P}_{\mathscr{V}_k} \to \mathscr{F}_{\mathscr{T}_k}$ which first unfolds the poset to a forest, and then unfolds the labels (which are pointed $k$-posets) to $k$-trees.

## 2.3 Bases and fine hierarchies

Next we recall some notation and notions relevant to the fine hierarchies. A *1-base* in a set $S$ is just a subalgebra $\mathscr{L}$ of $(P(S); \cup, \cap, \emptyset, S)$, i.e. a subset of of the Boolean $P(S)$ closed under finite unions and intersections. A *2-base* in $S$ is a pair $\mathscr{L} = (\mathscr{L}_0, \mathscr{L}_1)$ of 1-bases in $S$ such that $\mathscr{L}_0 \cup \check{\mathscr{L}}_0 \subseteq \mathscr{L}_1$, where $\check{\mathscr{L}}_1$ is the set of complements of sets in $\mathscr{L}_1$.

By an *$\omega$-base in a set $S$* we mean a sequence $\mathscr{L} = \mathscr{L}(S) = \{\mathscr{L}_n\}_{n<\omega}$ of 1-bases such that $\mathscr{L}_n \cup \check{\mathscr{L}}_n \subseteq \mathscr{L}_{n+1}$ for each $n$. Note that the $\omega$-bases subsume the 1-bases $\mathscr{L}$ (by taking $\mathscr{L}_0 = \mathscr{L}$ and $\mathscr{L}_{k+1} = (\mathscr{L})$ where $(\mathscr{L})$ is the Boolean closure of $\mathscr{L}$) and the 2-bases $(\mathscr{L}_0, \mathscr{L}_1)$ (by taking $\mathscr{L}_0 = \mathscr{L}_0$, $\mathscr{L}_1 = \mathscr{L}_1$ and $\mathscr{L}_{k+2} = (\mathscr{L}_1)$).

The $\omega$-base $\mathscr{L}$ is *reducible* if every its level $\mathscr{L}_n$ has the reduction property, i.e. for any $A, B \in \mathscr{L}_n$ there exist $A', B' \in \mathscr{L}_n$ such that $A' \subseteq A$, $B' \subseteq B$, $A' \cap B' = \emptyset$, and $A' \cup B' = A \cup B$.

We give two examples of bases. Let $\{\mathbf{\Sigma}^0_{1+n}\}_{n<\omega}$ be the $\omega$-base of finite $\mathbf{\Sigma}$-levels of Borel hierarchy in the Cantor space $X^\omega$. This base is well known to be reducible. The class $\mathscr{R}$ of regular $\omega$-languages over $X$ induces the $\omega$-base $\{\mathscr{R} \cap \mathbf{\Sigma}^0_{1+n}\}_{n<\omega}$ in $\mathscr{R}$. Since all regular $\omega$-languages sit in the Boolean closure of $\mathbf{\Sigma}^0_2$, the latter $\omega$-base coincides with the 2-base $\mathscr{R}\mathbf{\Sigma} = (\mathscr{R} \cap \mathbf{\Sigma}^0_1, \mathscr{R} \cap \mathbf{\Sigma}^0_2)$. As shown in [14], this base in also reducible.

With any $\omega$-base $\mathscr{L}$ in $S$ one can associate the *FH of $k$-partitions over $\mathscr{L}$* which is a family $\{\mathscr{L}(F)\}_{F \in \mathscr{T}_k(\omega)^{\sqcup}}$ of subsets of $k^S$. The notation system $\mathscr{T}_k(\omega)^{\sqcup}$ for levels of the FH based on iterated trees and forests is convenient for establishing properties of the FH in a series of papers of the second author (see e.g. [19] and references therein).

We do not reproduce here all (rather technical) details concerning the FH but we note that, instead of the notation system $\mathscr{T}_k(\omega)^{\sqcup}$ for its levels and the relation $\leq_h$ on the forests, we can equivalently take the larger system $\mathscr{V}_k(\omega)^{\sqcup}$ and the relation $\preceq$ on labeled posets, as explained in the introduction. In the second approach (first described in sections 7 and 8 of [17]) the FH over $\omega$-base $\mathscr{L}$ takes the form $\{\mathscr{L}(P)\}_{P \in \mathscr{V}_k(\omega)^{\sqcup}}$, where the levels have the property: $\mathscr{L}(P) = \mathscr{L}(u(P))$ for each $P \in \mathscr{V}_k(\omega)^{\sqcup}$ (see Lemma 8.16(5) in [17]). This property is crucial for the results in this paper, as explained in the introduction.

We give some details for the particular cases of FHs of $k$-partitions over 1-bases and 2-bases (which are in fact sufficient for this paper). The FH over a 1-base $\mathscr{L}$ in $S$ looks as $\{\mathscr{L}(F)\}_{F \in \mathscr{F}_k}$ (in the forest notation system) or as $\{\mathscr{L}(P)\}_{P \in \mathscr{P}_k}$ (in the poset notation system). The level $\mathscr{L}(P)$ of the latter hierarchy consists of all $k$-partitions $A : S \to \bar{k}$ such that for some family $\{B_p\}_{p \in P}$ of $\mathscr{L}$-sets we have: $A_i = \bigcup\{\tilde{B}_p \mid p \in P_i\}$ for each $i < k$, where $\tilde{B}_p = B_p \setminus \bigcup\{B_q \mid p < q\}$ and $P_i = c^{-1}(i)$, $c : P \to \bar{k}$. According to section 7 of [17], $\mathscr{L}(P) = \mathscr{L}(u(P))$, the family $\{B_p\}_{p \in P}$ may be assumed monotone (i.e., $B_p \supseteq B_q$ for $p < q$), and, if $P \in \mathscr{F}_k$ and $\mathscr{L}$ is reducible, the family $\{B_p\}_{p\in P}$ may be assumed reduced (i.e., $B_p \cap B_q = \emptyset$ for all incomparable $p, q \in P$).

The FH over a 2-base $(\mathscr{L}_0, \mathscr{L}_1)$ in $S$ looks as $\{\mathscr{L}(F)\}_{F\in\mathscr{F}_{\mathscr{T}_k}}$ (in the forest notation system) or as $\{\mathscr{L}(P)\}_{P\in\mathscr{P}_{\mathscr{V}_k}}$ (in the poset notation system). The level $\mathscr{L}(P)$ of the latter hierarchy consists of all $k$-partitions $A : S \to \bar{k}$ such that for some family $\{B_p\}_{p\in P}$ of $\mathscr{L}_0$-sets and for some families $\{B_{p_0p_1}\}_{p_1\in c(p_0)}$ of $\mathscr{L}_1$-sets, $p_0 \in P$, we have: $A_i = \bigcup\{\tilde{B}_{p_0p_1} \mid p_0 \in P, p_1 \in c(p_0)_i\}$ for each $i < k$, where $\tilde{B}_{p_0p_1} = B_{p_0p_1} \setminus \bigcup\{B_{p_0q_1} \mid p_1 < q_1 \in c(p_0)\}$, $\tilde{B}_{p_0} = \bigcup\{\tilde{B}_{p_0p_1} \mid p_1 \in c(p_0)\}$, and $c(p_0)_i = d^{-1}(i)$, $d : c(p_0) \to \bar{k}$. According to section 8 of [17], $\mathscr{L}(P) = \mathscr{L}(u(P))$, the families above may be assumed monotone, and, if $P \in \mathscr{F}_{\mathscr{T}_k}$ and the base $(\mathscr{L}_0, \mathscr{L}_1)$ is reducible, the families above may be assumed reduced.

For the 2-base $\mathscr{L} = (\mathscr{R} \cap \mathbf{\Sigma}^0_1, \mathscr{R} \cap \mathbf{\Sigma}^0_2)$ above, the FH $\{\mathscr{L}(F)\}_{F\in\mathscr{T}_k(2)^\sqcup}$ is the extension of Wagner's hierarchy to $k$-partitions introduced in [16]. For $k = 2$ we get back to the classical Wagner hierarchy in a set-theoretical presentation from [14]. As shown in [16, 19], every $\omega$-regular $k$-partition is Wadge complete in some level $F \in \mathscr{T}_k(2)^\sqcup$, all the possibilities are realized and the structure of such degrees is isomorphic to $\mathscr{T}_k(2)^\sqcup$.

## 2.4 Computational model

We conclude this section with comparing the computational model used in this paper with that from [1]. The paper [1] used Turing machines to analyze the complexity of their algorithm. However, for efficient algorithms the model that is closer to practical computations and that is widely considered to be standard is the Random Access Machine (RAM) model. Each memory entry in this model contains a string (typically bounded in length by $O(\log n)$, where $n$ is the size of input) and standard arithmetic operations on memory entries can be performed in constant time. Operations with memory (store, copy, load) as well as control operations (branching, subroutine calls) can also be done in constant time. See e.g. [3, Section 2.2] or [5, Section 1.1.2] for more details.

The paper [1] had constructed a cubic algorithm for the case of trees in the model of multitape Turing machines. We note that this algorithm is quadratic in RAM model. Indeed, the main recurrence relations on the complexity $t(n,k)$ of the algorithm given two structures of size $n$ and $k$ respectively are

$$t(n,k) \le \sum_{i=1}^{l} t(n_i,k) + O(n+k),$$

where $n_1, \ldots, n_l$ are any natural numbers such that $\sum_i n_i = n$, and the symmetric relation for $k$ instead of $n$. The solution to this recurrence is $t(n,k) = O(n^2k + nk^2)$.

The term $O(n+k)$ is needed on the step of the recursive construction to scan the inputs to find labels of specific nodes. This requires linear complexity on multitape Turing machines, but can be done with constant number operations in our model. Thus in our model the recurrence changes to

$$t(n,k) \le \sum_{i=1}^{l} t(n_i,k) + O(1)$$

and the complexity drops to $t(n,k) = O(nk)$.

# 3 Algorithms on labeled posets

Suppose we are given a finite poset $(P,c)$ labeled by a poset $Q$, i.e. $(P,c) \in \mathscr{P}_Q$ (see Section 2). In this section it is convenient to represent the bottom-up unfolding of $P$ by $F(P)$ instead of $u(P)$. It is convenient to represent elements of the unfolding $F(P)$ as paths $v = v_1 \dots v_t$, where $v_i \in P$, $v_1$ is a minimal element in $P$, $v_i < v_{i+1}$, and there are no other elements of $P$ between $v_i$ and $v_{i+1}$. The ordering on $F(P)$ can be naturally described as $v \leq w$, if $v$ is a prefix of $w$.

According to Section 2, on $\mathscr{P}_Q$ we have two preorders: $\leq_h$ and $\preceq$, where $P_1 \preceq P_2$ means that $u(P_1) \leq_h u(P_2)$. Deciding the relation $\leq_h$ on $\mathscr{P}_k$ is NP-complete for every $k \geq 2$ [12]. It becomes polynomial time decidable if we restrict posets to forests. However, as we show in this section, the order $\preceq$ is polynomial time decidable for arbitrary posets.

For $v \in P$ denote by $P{\uparrow_v} \subseteq P$ an upper cone of $v$ in $P$. It is easy to see that posets $F(P{\uparrow_v})$ and $F(P){\uparrow_u}$, where $u$ is an arbitrary element of unfolding with an endpoint in $v$, are isomorphic.

From this the following observation follows easily.

**Lemma 1.** *For any element $v \in P$ all orders $F(P){\uparrow_u}$, such that $u$ has an endpoint $v$, are isomorphic.*

Now we are ready to prove the main result of this section.

**Theorem 2.** *Assume that there is an algorithm $A$ that checks the relation $q_1 \leq q_2$ on $Q$ in time $C \cdot |q_1| \cdot |q_2|$, where $|q_i|$ is the size of the description of $q_i$ and $C$ is a positive constant. Then, there is an algorithm that checks the relation $(P_1, c_1) \preceq (P_2, c_2)$ on $\mathscr{P}_Q$ in time $C \cdot |P_1| \cdot |P_2|$.*

*Proof.* Next we describe the algorithm.

We will compute for all pairs of elements $v_1 \in P_1$ and $v_2 \in P_2$ one bit of information: whether there is a morphism from $F(P_1{\uparrow_{v_1}})$ to $F(P_2{\uparrow_{v_2}})$. We denote this bit by $M(v_1, v_2)$.

We assume that on the input we are given graphs corresponding to posets in which directed edges connect an element $p$ to each element $q$ such that $p < q$ and there are no other elements between $p$ and $q$. We call $q$ a *successor* of $p$. The graphs are given as adjacency lists.

We run a depth-first search (DFS) on $P_1$. For each vertex $v_1$ we fill-in the corresponding row of $M$ after visiting all its successors (this is our post-processing of the vertex $v_1$ in the DFS [4, Section 3.2.1]). For this we run a depth-first search on $P_2$ (that is, we have a loop over all vertices of $P_1$ and inside of it another loop over all vertices of $P_2$).

After visiting all the neighbors of a vertex $v_2$ we can use the following to compute $M(v_1, v_2)$.

**Claim 1.** *$M(v_1, v_2) = 1$ iff there is a successor $u_2$ of $v_2$, such that $M(v_1, u_2) = 1$, or $c_1(v_1) \leq c_2(v_2)$ and for each successor $u_1$ of $v_1$ we have $M(u_1, v_2) = 1$.*

Observe that once all values of $M$ are computed we can check if $P_1 \preceq P_2$ just by checking if for all $v_1$ there is $v_2$ such that $M(v_1, v_2) = 1$.

*Proof of the claim.* $M(v_1, v_2) = 1$ iff there is a morphism from $F(P_1{\uparrow_{v_1}})$ to $F(P_2{\uparrow_{v_2}})$. In this morphism $F(v_1)$ is mapped either to $F(v_2)$, or to some other element of the tree $F(P_2{\uparrow_{v_2}})$. The second case means that there is a successor $u_2$ of $v_2$, such that there is a morphism from $F(P_1{\uparrow_{v_1}})$ to $F(P_2{\uparrow_{u_2}})$. The first case means that the label of $c_1(v_1)$ is less or equal to $c_2(v_2)$, and that for any successor $u_1$ of $v_1$ there is a morphism of $F(P_1{\uparrow_{u_1}})$ to $F(P_2{\uparrow_{v_2}})$. $\square$

We can bound the running time of the algorithm (up to a multiplicative constant) by the following expression:

$$\sum_{v_1} \left( O(1) + d(v_1) + \sum_{v_2} \left( O(1) + d(v_2) + |A(c_1(v_1), c_2(v_2))| + d(v_1) + d(v_2) \right) \right),$$

where $d(v)$ is the out-degree of $v$ and $|A(c_1(v_1), c_2(v_2))|$ is the running time of the algorithm $A(c_1(v_1), c_2(v_2))$ to compare the labels. In this expression the terms $O(1) + d(v)$ correspond to the time needed to explore vertex $v$ in depth-first search and the terms $d(v_1)$ and $d(v_2)$ in the end of the formula is the time needed to check the conditions of the claim (we need to go over all neighbors of $v_1$ and $v_2$ for this).

Using the fact that the sum of the out-degrees of vertices in the graph is equal to the number of edges we can simplify the expression to the following (up to a multiplicative factor):

$$|V_1|+|E_1|+\sum_{v_1,v_2}(O(1)+d(v_2)+|A(c_1(v_1),c_2(v_2))|+d(v_1)+d(v_2)) =$$
$$|V_1|+|E_1|+|V_1||V_2|+|V_1||E_2|+\sum_{v_1,v_2}A(c_1(v_1),c_2(v_2))+|E_1||V_2|+|V_1||E_2| =$$
$$O\left(|V_1||V_2|+|V_1||E_2|+|E_1||V_2|+\sum_{v_1,v_2}A(c_1(v_1),c_2(v_2))\right)$$

Denote by $a_0$ and $b_0$ the sizes of descriptions of $P_1$ and $P_2$ respectively without the descriptions of labels (the size of $P_i$ is $O(|V_i|+|E_i|)$). The sizes of descriptions of labels in $P_1$ we denote by $a_1,\dots,a_k$, and in $P_2$ by $b_1,\dots,b_l$. Note that by the statement of the theorem $A(c_1(v_1),c_2(v_2)) = O(a_ib_j))$, where $a_i$ is the size of $c_1(v_1)$ and $b_j$ is the size of $c_2(v_2)$. Then we can upper bound the running time of the algorithm (up to a multiplicative factor) by

$$a_0b_0+\sum_{i=1}^{k}\sum_{j=1}^{l}a_ib_j \le \left(\sum_{i=0}^{k}a_i\right)\cdot\left(\sum_{j=0}^{l}b_j\right)$$

as needed. □

## 4 Algorithms on Muller's $k$-acceptors

Here we describe a feasible algorithm deciding the relation $L \le_W M$ (meaning that $L = M \circ f$ for some continuous function $f$ on $X^\omega$), where the $\omega$-regular $k$-partitions $L,M : X^\omega \to \bar{k}$ are given by Muller $k$-acceptors recognizing them.

The standard way to represent a Muller $k$-acceptor $(\mathscr{M},A)$ is to describe the graph of $\mathscr{M}$ with $n$ vertices (corresponding to the states $q_1,\dots,q_n$), $2^n$ bit vectors representing the sets of states, and labels $l < k$ of each vector representing the $k$-partition $A$. Because of the $2^n$ bit vector the size of this representation is $O(2^n)$ (we assume that $k$ is constant).

**Theorem 3.** *The relation $L(\mathscr{M}_1,A_1) \le_W L(\mathscr{M}_2,A_2)$ may be decided in quadratic time in the size of the inputs.*

*Proof.* According to the idea sketched in Sections 1 and 2, we first compute the levels $P_1,P_2 \in \mathscr{V}_k(2)^\sqcup$ in which the given $k$-partitions are Wadge complete, and then apply the algorithm of Theorem 2 to check whether $P_1 \preceq P_2$. We explain how to compute $P_1$. Let $\le_0$ and $\le_1$ be preorders on $C_{\mathscr{M}_1}$ defined in [23] as follows: $c \le_0 d$, if some (equivalently, every) state in $d$ is reachable from some (equivalently, every) state in $c$; $c \le_1 d$, if $c \supseteq d$. It follows that $\le_1$ implies $\equiv_0$ (the equivalence relation induced by $\le_0$), and that any $\equiv_0$-equivalence class has a largest cycle under inclusion. The preorders are easily computable from the presentation of $(\mathscr{M}_1,A_1)$ by using reachability in the graph of $\mathscr{M}_1$. Indeed, for preorder $\le_0$ it is enough to precompute reachability relation in the graph of $\mathscr{M}_1$ (this can be done in time $O(n^2)$ for graphs of constant degree, for example, by running bredth-first search from each vertex of the graph) and then for each pair of subsets of vertices to check, if a vertex in one subset is reachable from a vertex in the other one. For this we need constant time for each pair of subsets, which gives us $O(2^{2n})$ overall.

For relation $\leq_1$ we need to compute for each pair of subsets if one is included in the other one (we compute this relation for all subsets and then consider it for cycles only). We can think of subsets as enumerated by their characteristic vectors interpreted as binary representation of integers. We can compute the relation by splitting the vertices into two parts depending on the first coordinate of the characteristic vector. In each part we compute the relation recursively, and to compute the relation between the parts, observe that the relation holds for $0x$ and $1y$ iff it holds for $1x$ and $1y$, and the latter is already computed. To bound the running time of this recursive procedure, note that for any pair of subsets we need constant time of computation. Thus, the total time of this step is $O(2^{2n})$ as well.

As explained in the proof of Lemma 15 in [19], we can take $P_1 = (C_{\mathscr{M}_1}/_{\equiv_0}, \leq_0, d)$ where $d : C_{\mathscr{M}_1}/_{\equiv_0} \to \mathscr{V}_k$ is defined by $d([c]_0) = ([c]_0, \leq_1, A_1|_{[c]_0})$. Note that the equivalence classes in $C_{\mathscr{M}_1}/_{\equiv_0}$ bijectively correspond to the reachable strongly connected components (SCCs) of the graph of $\mathscr{M}_1$ (i.e., the largest cycles), hence $C_{\mathscr{M}_1}/_{\equiv_0}$ may be replaced by the set of all SCCs (canonical representatives in the equivalence classes). The latter set is computable in linear time from the graph of automaton by Tarjan's algorithm [20]. Altogether, $P_1, P_2$ are easily computable, and deciding the relation $P_1 \preceq P_2$ is quadratic in the size of input by Theorem 2. □

## 5 Representation of $k$-acceptors

As we discussed, the standard representation of a Muller $k$-acceptor is of size $\Theta(2^n)$. In this section we observe that actually, the number of possible cycles in the acceptor is at most $O(C^n)$, where $C < 2$ is a positive constant independent of the size of the acceptor (but dependent on the size of the alphabet). As a result, it is possible to have a more compact representation for $k$-acceptors.

Denote the number of vertices in the acceptor by $n$ and the size of the alphabet by $d$. We prove the following statement.

**Lemma 4.** *The number of cycles in an acceptor is at most*

$$\max(2^d, C^n + n),$$

*where* $C = 2\left(1 - \frac{1}{2^{d+1}}\right)^{\frac{1}{d+1}} < 2$.

Note that the first term $2^d$ in the maximum is constant in terms of $n$ and is needed only to cover the case of small constant $n$.

In the proof of the lemma we will use the following lemma proved in [2] (we state only the special case of their lemma that will be enough for us). This lemma is a simple consequence of the classic Product Theorem [8, Theorem 22.10].

**Lemma 5** ([2])**.** *Let $V$ be a finite set with $n$ elements and with subsets $A_1, \ldots, A_n$, such that every $v \in V$ is contained in exactly $\delta$ subsets. Let $\mathscr{F}$ be a family of subsets of $V$ and assume that there is a log-concave function $f \geq 1$ such that the projections $\mathscr{F}_i = \{F \cap A_i \mid F \in \mathscr{F}\}$ satisfy $|\mathscr{F}_i| \leq f(|A_i|)$ for each $i = 1, \ldots, n$. Then,*

$$|\mathscr{F}| \leq f(\delta)^{n/\delta}.$$

Next we proceed to the proof of Lemma 4. The proof follows the same strategy as the proof of Lemma 6 in [2].

*Proof of Lemma 4.* Denote the states of the automata by $Q = \{q_1, \dots, q_n\}$ and consider the automata as a directed graph on $Q$, in which for every vertex $q_i$ and for every letter in the alphabet $x \in X$ there is an edge leaving $q_i$ labeled by $x$ (some edges might be parallel and some edges might be loops). In particular, the out-degree of every vertex in the graph is $|X| = d$.

If $n \leq d$, then the number of cycles is less or equal to the number of subsets in $Q$, which is $2^d$. In this case we are done. Thus, from now on we assume that $n \geq d+1$.

Denote by $\mathrm{N}_-(q)$ for $q \in Q$ the set of vertices that have an outgoing edge to $q$. Analogously denote by $\mathrm{N}_+(q)$ the set of vertices that have incoming edges from $q$. Let $V = Q$ and $A_i = \{q_i\} \cup \{\mathrm{N}_-(q_i)\}$. If for $q_j$ we have $|\mathrm{N}_+(q_j)| = l$, then $q_j$ is contained in at most $l+1$ sets $A_i$. Note that $l$ might be smaller than $d$, since there might be parallel edges and loops. If $l < d$, we add $q_j$ to arbitrary sets $A_i$, that do not contain it yet, until $q_j$ is in $d+1$ sets. This is possible, since the total number of sets is $n$ and $n \geq d+1$. Denote the resulting sets by $A'_i$. Now each vertex $q_j$ is contained in exactly $d+1$ sets $A'_i$.

Denote by $\mathscr{C}$ the set of all cycles in the acceptor. Let $\mathscr{C}' = \mathscr{C} \setminus \{\{q\} \mid q \in Q\}$, that is $\mathscr{C}'$ consists of cycles of size at least 2.

Note, that for any $i$ the size of the set of projections $|\mathscr{C}'_i| = \{C \cap A_i \mid C \in \mathscr{C}'\}$ is at most $2^{|A_i|} - 1$, since $\{q_i\}$ cannot be a projection (any cycle of size at least 2 containing $q_i$ must contain one of its neighbors in $\mathrm{N}_-(q_i)$).

Consider a log-concave function $f(a) = 2^a - 1$ and apply Lemma 5 to $f$, $\mathscr{C}'$, sets $A'_1, \dots, A'_n$ and $\delta = d+1$. We get

$$|\mathscr{C}'| \leq f(d+1)^{n/d+1} = \left(2^{d+1} - 1\right)^{\frac{n}{d+1}}.$$

Since $|\mathscr{C}| \leq |\mathscr{C}'| + n$, the lemma follows. □

Thus, the number of cycles can be substantially smaller than the number of subsets on $Q$. As a result, if in the representation of an acceptor we provide the binary vector that contains a bit for each cycle, instead of each subset of states, the representation becomes more compact. However, for an algorithm to interpret this input, it needs first to compute the list of all cycles. In the next lemma we show that this can be done efficiently.

**Lemma 6.** *Given an acceptor $\mathscr{M}$ the set $\mathscr{C}$ of all cycles can be computed in time $O(n \cdot |\mathscr{C}|^2)$.*

*Proof.* Let an *elementary cycle* in a directed graph be a sequence of distinct vertices $v_1, \dots, v_t$, such that from each vertex $v_i$ there is a directed edge to $v_{i+1}$ and there is a directed edge from $v_t$ to $v_1$. The paper [7] provides an algorithm that constructs all elementary cycles in a directed graph $G = (V, E)$ in time $O((|V| + |E|)p)$, where $p$ is the number of elementary cycles. We first apply this algorithm to the acceptor and denote the resulting list by $\mathscr{C}'$.

From this list we can construct the list $\mathscr{C}$. For this we first consider an undirected graph $G_C$ with $\mathscr{C}'$ as a set of vertices, such that $C_i$ and $C_j$ are connected by an undirected edge if they share at least one vertex. It is easy to see that cycles in $\mathscr{C}$ correspond to induced connected subgraphs of $G_C$.

The graph $G_C$ can be constructed in time $O(n|\mathscr{C}'|^2)$: it is enough for each pair of vertices $C_i$, $C_j$ to check if the cycles share a vertex. This can be done in $O(n)$.

To store all elements of $\mathscr{C}$ we maintain a red-black tree data structure on them. In the beginning the data structure is empty. To add elements of $\mathscr{C}$ to the red-black tree we iterate through the induced connected subgraphs of $G_C$ and check if they correspond to a new element of $\mathscr{C}$. More precisely, first we create a queue of size 1 subgraphs of $G_C$ (they correspond to just elements of $\mathscr{C}'$). With each element $S \subseteq \mathscr{C}'$ in the queue we will store the characteristic vector $N(S)$ of the set of its neighbors in $G_C$ and the characteristic vector $Cyc(S)$ of the cycle in $\mathscr{M}$ that it corresponds to. Computing these vectors for the each of the first elements of the queue naively takes time $O(|\mathscr{C}'|)$ and $O(n)$ respectively.

Next we repeat the following step. We extract the current first element $S \subseteq \mathscr{C}'$ of the queue. For each of its neighbors $C$ (there are at most $|\mathscr{C}'|$ of them), we add $C$ to $S$ and compute $Cyc(S \cup C)$. This can be done in time $O(n)$ by scanning through the corresponding vectors for $S$ and $C$. We check if the cycle $Cyc(S \cup C)$ was already computed before. If it was, we move on to the next neighbor of $S$. If this is a new cycle we add it to the red-black tree and add $S \cup C$ to the end of the queue. These operations with the red-black tree can be performed in time logarithmic in the size of the tree, that is in time $O(\log|\mathscr{C}|) = O(n)$ (since $|\mathscr{C}| \leq 2^n$). When adding new element to the queue we compute $N(S \cup C)$, this takes time $O(|\mathscr{C}'|)$. We are done once the queue is empty.

For the correctness of this procedure, note that if two induced connected subgraphs $S, S' \subseteq \mathscr{C}'$ correspond to the same cycle, they have the same set of neighbors in $G_C$. As a result, our queue will scan through subgraphs corresponding to all cycles.

Moreover, for each cycle $C \in \mathscr{C}$ we will have exactly one subgraph corresponding to it. For the running time this means that the total number of subgraphs that are added to the queue is $|\mathscr{C}|$. When we add each element to the queue, we compute $N(S)$ for it. For each $S$ in the queue we consider at most $|\mathscr{C}'|$ neighbors, compute $Cyc$ for the union and perform queue operations. In total, the running time is $O(|\mathscr{C}| \cdot (|\mathscr{C}'| + |\mathscr{C}'| \cdot n)) = O(|\mathscr{C}| \cdot |\mathscr{C}'| \cdot n) = O(|\mathscr{C}|^2 \cdot n)$. □

Next we prove a version of Theorem 3 for the more compact representation of $k$-acceptors.

**Theorem 7.** *The relation $L(\mathscr{M}_1,A_1) \leq_W L(\mathscr{M}_2,A_2)$ may be decided in time $O\left(n \cdot C^2 + n^2\right)$, where n is the size of the representations of graphs of $\mathscr{M}_1$ and $\mathscr{M}_2$ and C is the size of the representations of cycles in $\mathscr{M}_1$ and $\mathscr{M}_2$.*

*Proof.* The proof is analogous to the proof of Theorem 3.

As before, we compute preorders $\leq_0$ and $\leq_1$. For $\leq_0$, as before, we can precompute reachability relation on the graph of the automata (this takes $O(n^2)$ time) and then for each pair of cycles check reachability between a couple of vertices in them (this requires $C^2$ time). For the inclusion relation we can check inclusion of each pair of cycles in $O(n)$ straightforwardly. This results in time $O(n \cdot C^2)$

The remaining part of the proof remains completely the same and requires $O(C^2)$ time. □

**Acknowledgments**

V. Selivanov's research was supported by the Russian Science Foundation, project 23-11-00133.

## References

[1] Pavel Alaev & Victor L. Selivanov (2021): *Complexity Issues for the Iterated h-Preorders*. In Yo-Sub Han & Sang-Ki Ko, editors: *Descriptional Complexity of Formal Systems - 23rd IFIP WG 1.02 International Conference, DCFS 2021, Virtual Event, September 5, 2021, Proceedings*, *Lecture Notes in Computer Science* 13037, Springer, pp. 1–12. Available at `https://doi.org/10.1007/978-3-030-93489-7_1`.

[2] Andreas Björklund, Thore Husfeldt, Petteri Kaski & Mikko Koivisto (2008): *The Travelling Salesman Problem in Bounded Degree Graphs*. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir & Igor Walukiewicz, editors: *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part I: Tack A: Algorithms, Automata, Complexity, and Games*, *Lecture Notes in Computer Science* 5125, Springer, pp. 198–209. Available at `https://doi.org/10.1007/978-3-540-70575-8_17`.

[3] T.H. Cormen, C.E. Leiserson, R.L. Rivest & C. Stein (2022): *Introduction to Algorithms, fourth edition*. MIT Press. Available at `https://books.google.com/books?id=HOJyzgEACAAJ`.

[4] Sanjoy Dasgupta, Christos H. Papadimitriou & Umesh V. Vazirani (2008): *Algorithms*. McGraw-Hill.

[5] M.T. Goodrich & R. Tamassia (2014): *Algorithm Design and Applications*. Wiley. Available at `https://books.google.com/books?id=tQBFBQAAQBAJ`.

[6] Peter Hertling & Victor L. Selivanov (2014): *Complexity issues for Preorders on finite labeled forests*. In Vasco Brattka, Hannes Diener & Dieter Spreen, editors: *Logic, Computation, Hierarchies, Ontos Mathematical Logic* 4, De Gruyter, pp. 165–190. Available at `https://doi.org/10.1515/9781614518044.165`.

[7] Donald B. Johnson (1975): *Finding All the Elementary Circuits of a Directed Graph*. *SIAM J. Comput.* 4(1), pp. 77–84. Available at `https://doi.org/10.1137/0204007`.

[8] Stasys Jukna (2011): *Extremal Combinatorics - With Applications in Computer Science*. Texts in Theoretical Computer Science. An EATCS Series, Springer. Available at `https://doi.org/10.1007/978-3-642-17364-6`.

[9] T. Kihara & A. Montalbán (2019): *On the structure of the Wadge degrees of bqo-valued Borel functions*. *Trans. Amer. Math. Soc.* 371, pp. 7885–7923. Available at `https://doi.org/10.1090/tran/7621`.

[10] S. Kosub (2000): *Complexity and Partitions*. Phd thesis, Universität Würzburg.

[11] Sriram C. Krishnan, Anuj Puri & Robert K. Brayton (1995): *Structural Complexity of Omega-Automata*. In Ernst W. Mayr & Claude Puech, editors: *STACS 95, 12th Annual Symposium on Theoretical Aspects of Computer Science, Munich, Germany, March 2-4, 1995, Proceedings*, *Lecture Notes in Computer Science* 900, Springer, pp. 143–156. Available at `https://doi.org/10.1007/3-540-59042-0_69`.

[12] Léonard Kwuida & Erkko Lehtonen (2011): *On the Homomorphism Order of Labeled Posets*. *Order* 28(2), pp. 251–265. Available at `https://doi.org/10.1007/s11083-010-9169-x`.

[13] D. Perrin & J.-E. Pin (2004): *Infinite Words*. Elsevier.

[14] Victor L. Selivanov (1998): *Fine Hierarchy of Regular Omega-Languages*. *Theor. Comput. Sci.* 191(1-2), pp. 37–59. Available at `https://doi.org/10.1016/S0304-3975(97)00301-0`.

[15] Victor L. Selivanov (2004): *Boolean hierarchies of partitions over a reducible base*. *Algebra and Logic* 43(1), pp. 44–61. Available at `https://doi.org/10.1023/B:ALLO.0000015130.31054.b3`.

[16] Victor L. Selivanov (2011): *A Fine Hierarchy of $\omega$-Regular $k$-Partitions*. In Benedikt Löwe, Dag Normann, Ivan N. Soskov & Alexandra A. Soskova, editors: *Models of Computation in Context - 7th Conference on Computability in Europe, CiE 2011, Sofia, Bulgaria, June 27 - July 2, 2011. Proceedings*, *Lecture Notes in Computer Science* 6735, Springer, pp. 260–269. Available at `https://doi.org/10.1007/978-3-642-21875-0_28`.

[17] Victor L. Selivanov (2012): *Fine hierarchies via Priestley duality*. *Ann. Pure Appl. Log.* 163(8), pp. 1075–1107. Available at `https://doi.org/10.1016/j.apal.2011.12.029`.

[18] Victor L. Selivanov (2022): *Non-collapse of the effective Wadge hierarchy*. *Comput.* 11(3-4), pp. 335–358. Available at `https://doi.org/10.3233/COM-210376`.

[19] Victor L. Selivanov (2023): *Wadge Degrees of Classes of $\omega$-Regular* k*-Partitions*. *J. Autom. Lang. Comb.* 28(1-3), pp. 167–199. Available at `https://doi.org/10.25596/jalc-2023-167`.

[20] Robert Endre Tarjan (1972): *Depth-First Search and Linear Graph Algorithms*. *SIAM J. Comput.* 1(2), pp. 146–160. Available at `https://doi.org/10.1137/0201010`.

[21] Wolfgang Thomas (1990): *Automata on Infinite Objects*. In Jan van Leeuwen, editor: *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, Elsevier and MIT Press, pp. 133–191. Available at `https://doi.org/10.1016/b978-0-444-88074-1.50009-3`.

[22] W. Wadge (1984): *Reducibility and determinateness in the Baire space*. Phd thesis, University of California, Berkely.

[23] K. Wagner (1979): *On ω-regular sets*. *Information and Control* 43, pp. 123–177. Available at `https://doi.org/10.1016/S0019-9958(79)90653-3`.

[24] T. Wilke & H. Yoo (1995): *Computing the Wadge degree, the Lipschitz degree, and the Rabin index of a regular language of infinite words in polynomial time*. In Peter D. Mosses, Mogens Nielsen & Michael I. Schwartzbach, editors: *TAPSOFT '95: Theory and Practice of Software Development*, *Lecture Notes in Computer Science* 915, Springer, Berlin, Heidelberg, pp. 288–302. Available at `https://doi.org/10.1007/3-540-59293-8_202`.

# Large Language Models and the Extended Church-Turing Thesis[*]

Jiří Wiedermann
Institute of Computer Science
Czech Academy of Sciences
Prague, Czech Republic
jiri.wiedermann@cs.cas.cz

Jan van Leeuwen
Department of Information and Computing Sciences,
Utrecht University, the Netherlands
J.vanLeeuwen1@uu.nl

The Extended Church-Turing Thesis (ECTT) posits that all effective information processing, including unbounded and non-uniform interactive computations, can be described in terms of interactive Turing machines with advice. Does this assertion also apply to the abilities of contemporary large language models (LLMs)? From a broader perspective, this question calls for an investigation of the computational power of LLMs by the classical means of computability and computational complexity theory, especially the theory of automata. Along these lines, we establish a number of fundamental results. Firstly, we argue that any fixed (non-adaptive) LLM is computationally equivalent to a, possibly very large, deterministic finite-state transducer. This characterizes the base level of LLMs. We extend this to a key result concerning the simulation of space-bounded Turing machines by LLMs. Secondly, we show that lineages of evolving LLMs are computationally equivalent to interactive Turing machines with advice. The latter finding confirms the validity of the ECTT for lineages of LLMs. From a computability viewpoint, it also suggests that lineages of LLMs possess super-Turing computational power. Consequently, in our computational model knowledge generation is in general a non-algorithmic process realized by lineages of LLMs. Finally, we discuss the merits of our findings in the broader context of several related disciplines and philosophies.

## 1 Introduction

**Historical context** Back in 2001, at the occasion of entering a new millennium, a notable book entitled *"Mathematics Unlimited — 2001 and Beyond"* appeared, giving a unique overview of the state of mathematics at the end of the twentieth century and offering remarkable insights into its future development and that of its related fields. In one of the book chapters, we investigated the role of the classical Turing machine paradigm in contemporary computing [15]. Especially, we were interested in whether the *Church-Turing Thesis* (CTT), claiming that every algorithm can be described in terms of a Turing machine, has withstood the 'test of time'. Did this thesis still apply to all modern computations as we witness them today? We characterized the latter as computations that fulfill three conditions that appeared with the advent of modern computing technologies: *non-uniformity of programs, interactivity* (or *reactivity*), and *infinity of operation.* Based on this observation, we brought evidence and argumentation in favor of what we called the *Extended Church-Turing Thesis* (ECTT), which can be seen as a strengthening of the classical Church-Turing Thesis from this newer perspective:

> **Extended Church-Turing Thesis:** *All effective information processing, including unbounded and non-uniform interactive computations, can be modeled in terms of interactive Turing machines with advice.*

[*]The research of the first author was partially supported by Grant No. CK04000150 EBAVEL of the Czech Technology Agency, programme Strategy AV21 "Philosophy and Artificial Intelligence", and the Karel Čapek Center for Values in Science and Technology.

Interactive Turing machines with advice (ITM/As) are variants of Turing machines that accommodate the above-mentioned, most general conditions: non-uniformity of programs (covered by the advice mechanism), interactivity, and potentially unbounded operation [15]. The extended thesis still perceives Turing machines as etalons of all underlying computations.

Examples of effective processes covered by the ECTT, aside from those covered by the classical CTT, include artificial computational systems like the Internet, sensory nets, ad-hoc nets, cognitive systems, relativistic computing, and natural data processing in DNA, cells, brains, and the Universe, and similar systems.

Since the formulation of the ECTT, more than 20 years have passed. Today, new pretenders to the most complex software systems have emerged: large language models (LLMs). Namely, as the pinnacles of contemporary information technology, LLMs mark a paradigm shift in the AI landscape, offering so-far unseen properties: emerging autonomy, and amazing abilities of natural language processing and understanding. Some even see LLMs as messengers of artificial general intelligence (AGI) [1].

**Motivation** From this perspective, the following questions suggest themselves. Does the ECTT apply to the effective behavior of LLMs, i.e., can ITM/As simulate LLMs? And if so, can LLMs simulate any ITM/As, or is it beyond their power?

These questions seem to be of purely academic interest, but answering them is significant for several reasons. For one, a positive answer to the first question would confirm the thesis's validity for LLMs and the ITM/A would keep its position as a lighthouse pointing the pathways in modern computing. This in turn would confirm the fundamental position of Turing machines as a model of computation.

The second question is challenging as well. Namely, it calls for mutual simulations between both kinds of devices. Alas, it is hard to imagine more opposing models. An ITM/A is an elegant mathematical abstraction of purposeful information processing aiming at maximal computational efficiency, whereas an LLM is a pinnacle of contemporary real information processing technology.

Yet both devices share a common inspiration: the human brain and its information-processing abilities. From this perspective, both devices are complementary. On the one hand, ITM/As seem to be suitable mainly for modeling the brain body-controlling abilities by performing mechanistic, non-uniform logic-inspired processing of non-verbal information, using potentially unbounded additional memory. On the other hand, LLMs epitomize a biologically inspired approach mimicking the ability of the human brain to learn, produce, and understand natural languages (i.e., verbal information) by finite-state machines. Can one imagine a conceptually more distant pair of devices?

**Contributions and results** Along the previous lines, the paper contributes to the present state of knowledge about the computational power of LLMs, by answering the questions as implied by the ECTT.

First, we present mutual simulations between interactive finite-state transducers and fixed (trained and non-adaptive) LLMs. This characterizes the computational power of both kinds of devices by proving their computational equivalence, within the limit of the feasible context windows of the LLMs.

Second, we present a fundamental simulation of a space-restricted TM by a "standard", sufficiently large LLM. The vocabulary and word-to-vector embedding mechanisms of the simulated LLM must be adjusted for processing the "language" represented by the computations of the simulated TM. Nevertheless, the necessary adjustments do not intervene in the standard architecture and internal workings of LLMs. The simulation also reveals the interdependence between the space complexity of the simulated TM and the size of the word-to-vector representation used by the simulating LLM.

Third, the insights gained from the previous simulations enable the design of mutual simulations between ITM/As and *lineages* of evolving LLMs, i.e. of sequences of ever more capable LLMs. It leads to a characterization of the computational power of the latter: lineages of LLMs possess super-

Turing computational power. In the recent investigations of the computational complexity of LLMs (e.g. [11, 13, 14]), the results characterizing the computational power of evolving LLMs seem to be the first to deal with the general complexity-theoretic aspects of this kind of device. If we see the LLMs as knowledge generating devices, then their super-Turing abilities indicate that in this computational model, knowledge is non-computational: it cannot be generated in general by computers satisfying the classical Church-Turing Thesis.

The paper concludes with a discussion of the amazing knowledge-generating capabilities of (very) large finite state systems, their limits, and the influence of the paradigm change in AI brought forth by the technologies used by LLMs.

Closing this introductory section, a word of warning concerning the paper's presentation style is in place. We see our paper as a pioneering, pre-formal paper making a preliminary inquiry into the newly born field of LLMs from the viewpoint of the computability theory. Necessarily, it is less formal since the basic concepts related to LLMs and their understanding are still in statu nascendi. Priority is given to the investigation of the scope of the new research field, the potential of automata theory to solve the related problems, and their merits for associated disciplines and philosophies. Therefore, the paper presents only the outlines of the basic concepts, proofs and contemplations on the respective achievements.

## 2 Simulations between LLMs and finite-state transducers

When it comes to characterizing the computational power of LLMs, note that a single LLM cannot in principle simulate an arbitrary Turing machine, simply because LLMs are finite-state devices and no such device can simulate an infinite-state machine. Thus, to characterize the computational power of LLMs, we must look for weaker models than Turing machines. Finite-state transducers (FSTs) offer themselves as a natural choice. We will argue first that deterministic FSTs and fixed (non-adaptive) LLMs are computationally equivalent, within the input limits of the latter.

As far as the computational model of an LLM is concerned, we will consider a standard, or nowadays one might even say, 'classical' elementary ChatGPT model as described, e.g., in [24]. These models are able to learn probability distributions of words (tokens) during a training phase, and make use of it during their inference phase. Probabilistic mechanisms are not included in the LLM architecture. During inference, their behavior is influenced by their initial setting and a learned probabilistic distribution captured from the training data.

From a computability viewpoint, trained LLMs can be viewed as deterministic interactive finite-state systems producing knowledge in response to the initial prompts. They work with finite precision weights and learned fixed statistical behavior, devoid of any memory or functional augmentations. They read their inputs in a sequential manner. LLMs require substantial computational resources for their deployment, although the development of their transformer decoder-based technology leads to ever more efficient implementations [14].

A *finite-state transducer* is a finite-state machine with two tapes, following the terminology for Turing machines: a two-way read-only input tape and a write-only output tape. An FST reads strings of a given set on the input tape and generates strings of a related set on the output tape. An FST is allowed to make steps that do not consume an input symbol ($\varepsilon$-moves), to reflect 'internal processing' of state information. A general FST can be thought of as a generator of strings, a deterministic FST as a mapping from input to output strings.

The input-to-output properties of FSTs relate their abilities to those of LLMs. However, the models are still very different. For example, LLMs have an intrinsic ability to learn and adapt based on vast amounts of data and using statistical methods, whereas FSTs are designed with static rule-sets or transition tables. Also, LLMs normally process inputs from 'context windows' of some bounded size, to allow for the effect of various mechanisms to generate adequate output, whereas FSTs have no such limitation on their input sequences. Techniques for 'long text processing' of LLMs are rapidly advancing [3, 7] but not yet standard.

For a basic comparison we restrict attention to the capabilities of *fixed* (or, static) LLMs, i.e., LLMs that are trained and ready but do not change or adapt during their operation. We may assume that these LLMs are deterministic and of bounded size (in bits). To enable a mutual simulation between the two kinds of devices, we assume that FSTs and LLMs work over the same alphabet and that an LLM's parameters are represented as finite strings of bits. In order to keep the modeling discrete and finite-state we do not assume the use of analog neural nets working with real or rational weights (cf. [12]).

**Theorem 1.** *Let $\mathscr{L}$ be a fixed LLM. Then, there is a deterministic FST $\mathscr{T}$ simulating $\mathscr{L}$.*

*Proof.* (Outline.) Let $\mathscr{L}$ be a fixed LLM. We argued that $\mathscr{L}$ is deterministic and of bounded size. The Church-Turing thesis guarantees that the deterministic system $\mathscr{L}$ can, in principle, be simulated by a Turing machine with the same input convention. Since $\mathscr{L}$ is of bounded size and hence a finite-state system, the simulating Turing machine is of constant space complexity for all inputs and thus, computationally equivalent to some deterministic FST $\mathscr{T}$. This proves that theoretically, a sufficiently large and complex FST could simulate the behavior of $\mathscr{L}$. □

The following remarks are in order. The simulation in Theorem 1 can also be seen as a philosophic thought experiment, revealing strong and weak aspects of the involved devices. On the one hand, it leads to the illuminating finding that, from a computability-theoretic point of view, fixed LLMs are essentially deterministic finite-state transducers, which are fundamental devices of automata theory. On the other hand, from a computational complexity point of view, a fixed LLM with millions or billions of parameters would translate into a finite-state transducer with an astronomical number of states and transitions among them. This 'state explosion' makes the resulting FST computationally intractable, but efficiency has not been our current concern. Nor has it been our concern that the resulting FST would be incredibly complex and opaque and would not offer the same level of insight into how the LLM arrives at its outputs, compared to analyzing the LLM's internal architecture directly.

The proof of the reverse to Theorem 1 is more involved since, to simulate a deterministic FST on an LLM, one can only use the standard LLM mechanisms that are quite incompatible with an FST's computational mechanisms. We will argue that the processing of an input string by a deterministic FST can be simulated by the 'answering' of the same input as a prompt by a suitable (fixed) LLM.

**Theorem 2.** *Let $\mathscr{T}$ be a deterministic FST. Then, for any $n \geq 1$, there is a fixed LLM $\mathscr{L}$ simulating $\mathscr{T}$ on all words of length at most $n$.*

*Proof.* (Outline.) Given $n$, we indicate how to construct an LLM $\mathscr{L}$ as required. Note that the given FST $\mathscr{T}$ is a deterministic two-way transducer with a finite number of states, a finite number of input and output symbols or phrases, and a finite number of transition rules. For an LLM, this makes it a relatively simple system to learn: it can just represent the single-valued transition function of $\mathscr{T}$ with rules $(current\_state, symbol\_read) \rightarrow (head\_move, new\_state, output\_symbol)$, for every pair $(current\_state, symbol\_read)$.

The transition rules of $\mathscr{T}$ can be directly embedded into the vector representations used by an LLM. The rules can be seen as the 'words' of the language that is to be processed by $\mathscr{L}$, the sentences are the sequences of 'words' as they are applied in deterministic order when $\mathscr{T}$ is processing an input string. The LLM can efficiently learn the behavior of $\mathscr{T}$ because the LLM's attention mechanism can effectively capture the single-valued relationship between a tuple $(current\_state, symbol\_read)$ and the corresponding tuple $(head\_move, new\_state, output\_symbol)$.

When presented with a prompt (input) of size at most $n$, $\mathscr{L}$ stores it for reference and begins the generation of the 'answer' (output string) like $\mathscr{T}$ would. The attention mechanism 'predicts' (here, identifies) the unique transition rule based on the current state and input symbol, $\mathscr{L}$ outputs it as the 'next' word of the output, updates its state and $O(\log n)$-size positional information, and repeats, thus generating the output string word after word (until $\mathscr{T}$ would stop or some limit is exceeded). In stead of outputting the words, $\mathscr{L}$ can 'decode' them and produce the output symbol that is contained in them. This effectively makes $\mathscr{L}$ into a fixed LLM that mimics the deterministic transducer within the limit of the context windows that it can handle. ☐

There are several implicit facts one may observe in the proof. First, the proof illustrates that in general learning the prediction-of-the-next-word idea is enough. The learning of the FST makes use of exactly this idea. Second, the language of a (deterministic) FST to be learned is composed of fragments of sequences of state transitions covering the state diagram of the transducer. Such a state diagram consists of a finite number of cycles covering this diagram, and each computation presents a concatenation of finite paths along these cycles. Hence what has to be learned is a finite number of fragments covering the state diagram between the "crossroads" where the cycles meet.

We return to Theorem 2 in Section 3.1. The two theorems lead to the following conclusion, respecting the input limitations of the LLMs.

**Theorem 3.** *The computational power of fixed LLMs equals that of deterministic FSTs.*

Theorem 3 is the ground level result for our purposes. More powerful simulations can be proved for LLMs that allow for enhanced capabilities of the transformer decoders. For an overview of this recent research area, see e.g. [5, 13].

## 3 Simulations between LLMs and interactive Turing machines with advice

We now turn to our main question: the position of LLMs with respect to the ECTT. How to deal with the development towards more and more powerful LLMs in this respect? In answering it, an important role will be played by the simulation of (deterministic) Turing machines by LLMs.

### 3.1 Simulating Turing machines 'inside' of LLMs

When contemplating the simulation of Turing machines (TMs) by LLMs, a first solution that comes to mind is the one that led A.M. Turing to the design of his 'Turing machine'. In this solution, the LLM at hand is augmented with an external, potentially unbounded memory that will take the role of the tape of the simulated TM, and the LLM itself will merely serve as the finite-state control of that machine. Essentially, this is the solution presented by Schuurmans [11], who showed how to operate an external read-write memory using specific prompts to simulate computations of a universal TM. In doing so, it was not necessary to modify the LLM's weights, which the author sees as a key aspect of his proposal.

In our present approach, we strive for a different exploitation of the computational potential of LLMs, without augmenting them with any external memories – by scrutinizing the resource limits of their computational mechanisms. This is achieved by specializing an LLM to the desired 'degree' in its only task, the simulation of the given TM as far as the finite-state nature of the model allows it. Accepting the fact that LLMs are finite-state devices it is clear that, if the space complexity of the simulated TM grows with the size of its inputs, we cannot hope for its simulation by an LLM on all inputs. That is, we must accept that our simulation by an LLM of a given size can only work for a TM up to a certain fixed space complexity bound for its work tape(s). This is what we call a TM simulation 'inside' of an LLM.

Now a crucial question arises that has to be answered first: where can we gain the space needed for representing a TM and its computations inside an LLM? How can we exploit the existing LLM's architecture and mechanisms, without introducing new ones? To find a solution to this problem, we look for an analogy between natural language processing (NLP) by LLMs and TM computations. Can we interpret a TM computation as a language-processing task? We consider the generic case of deterministic multi-tape TM acceptors.

Before explaining the analogy, let's look at a common representation of a successful computation by the given TM. On processing a given input, the TM generates a 'sequence of configurations'. The input is written on the separate read-only input tape – which will be presented as a 'prompt' to the simulating LLM. Each configuration of the TM consists of a 'listing' of its current work tape configurations. Each work tape configuration is represented by the contents of that work tape and the position of the read/write head on that tape. The input symbol currently scanned by the TM's reading head on the input tape and the current state are appended to the end of the current list of work tape configurations. The 'sequence of configurations' of the TM starts with the list of initial configurations for each work tape. It ends with an accepting configuration or a configuration that exceeds the allowable space limit on the work tapes as derived from the size of the LLM. (We assume that looping is prevented, e.g. by timing constraints.) The transition function of the simulated TM orders the configurations in a 'valid' TM computation.

To see the analogy with natural language processing, one may view the configurations as the 'words' of a fictive 'Turing machine language'. The syntax of these words is given by the prescription for correct configuration representations. Sequences of such words represent sentences, or their fragments, of the underlying 'Turing machine language'. The semantics of the language is given by the orderings of such words following the TM transition function. As a result, the words in a sentence are ordered according to the cause-and-effect principle, because the simulated TM is assumed to be deterministic. Any semantically correct sequence of such words represents a valid fragment of TM computations, and its processing gives it its meaning.

**Theorem 4.** *Let $\mathscr{M}$ be a deterministic multi-tape TM of space complexity $S(n)$. Then, for any $n$ and $k$ such that $S(n) \leq k$, there is an LLM $\mathscr{L}$ using word-to-vector embeddings of size $O(k)$ simulating $\mathscr{M}$ on any input of length $n$.*

*Proof.* Given $n$ and $k$ such that $S(n) \leq k$, we construct an LLM $\mathscr{L}$ that simulates $\mathscr{M}$ on inputs of size $n$. Without loss of generality, $\mathscr{M}$ may be assumed to be always halting. (Note that halting computations cannot be longer than the number of different configurations of $\mathscr{M}$, which is bounded by $c^{\log n + S(n)}$ for some constant $c$. This can be checked by keeping a count of the number of steps. If $S(n) \geq \log n$, this can be done within the space bound by $\mathscr{M}$ itself, otherwise it can be done by $\mathscr{L}$ itself.) $\mathscr{L}$ will be a 'standard' LLM with word-to-vector embeddings of size $O(k)$ with a special attentional mechanism to be described later in this proof.

In the training phase, our initially "empty" LLM must be trained on various valid fragments of the TM computation for inputs of size $n$, i.e. with configurations of size at most $S(n)$, where $S(n) \leq k$. Doing so, the set of all work tape configurations up to size $k$ will become the basic 'vocabulary' (set of words) of the language of our LLM. Each configuration will serve as the word-to-vector embedding of some word from the underlying 'Turing machine language'. The result of the training phase is the representation of the complete transition table for tape configurations of $\mathscr{M}$ in the LLM's memory for all inputs of size $n$ with $S(n) \leq k$.

After the training phase, the simulation of $\mathscr{M}$ on any given input of length $n$ such that $S(n) \leq k$ (given to the LLM in the form of a prompt) can start. The simulation starts from $\mathscr{M}$'s initial configuration: the input word is supplied at $\mathscr{L}$'s interface and stored for reference, and the initial tape configurations of $\mathscr{M}$ are given in the respective word-to-vector embedding, i.e., as a word in the "Turing machine language".

If the training phase was long enough (including valid fragments for all possible transitions) and the desired transition table of tape configurations 'fits' into the LLM, the model will generate the correct 'sentence' of consecutive words that corresponds to $\mathscr{M}$'s computation on the given input, as for each configuration there is exactly one successor configuration (because $\mathscr{M}$ is deterministic). As we assumed that $\mathscr{M}$ always halts, the generated sentence will be finite. When complete, $\mathscr{L}$ can answer by outputting 'accept' or 'reject' depending on the final word.

If the training phase was not long enough, which may happen if $k$ is large, then configurations may arise for which the proper transition rule is missing and is yet to be 'learned'. Eventually a 100% correctness of the simulating LLM can be achieved by tuning its attentional mechanism (cf. [16]) to follow the transition function among successive configurations of the simulated TM, as in the proof of Theorem 2. There is no need to track the relations between the words (configurations) across long sequences.

Thus, $\mathscr{L}$ eventually simulates $\mathscr{M}$ on all inputs within the bounds it can handle. $\square$

Theorem 4 can be seen as a generalization of Theorem 2 although, strictly speaking, the simulating LLM need not be 'fixed' (non-adapting). The LLM is likely to be very large. As the entire transition table for $\mathscr{M}$'s work tape configurations for inputs of size $n$ must be represented in the word-to-vector embeddings of $\mathscr{L}$, the simulating LLM is likely to have a space complexity of at least order $O(c^k)$, where $c \geq 2$ is a bound on the size of the alphabets and the state set of $\mathscr{M}$ and $S(n) \leq k$.

A further remark can be made. The proof shows that an LLM $\mathscr{L}$ can be designed to generate the chain of configurations corresponding to $\mathscr{M}$'s computation on an input, regardless of what the purpose of the computation actually is. If, for example, $\mathscr{M}$ was meant to compute a more general recursive function of the input instead, then $\mathscr{L}$ could be used equally well to obtain (an encoding of) the resulting function value that is represented in the final configuration of $\mathscr{M}$. This opens the way to the use of LLMs for 'computing' arbitrary *recursive functions,* although it is uncommon that the LLM may well have to go through many 'internal' word generations before it can actually output an answer. In several recent studies, the possible extension of LLMs to allow for precisely these extended chains of inferences are explored [5, 9].

We now argue that Theorem 4 even holds for Turing machines with advice, a very powerful variant of the TM model that we will employ below. A *Turing machine with advice* (TM/A) is a (deterministic multi-tape) Turing machine with an oracular input facility which, when the machine is given any input $w$, provides the TM with an extra read-only input in the form of a finite string ('advice') that depends only on the length of $w$, i.e. that is the same for all inputs of the given length. Advice models the possibility that TM programs can get adjusted or modified over time, especially as input sizes increase. The advice string is placed on a separate read-only advice tape. Similar to the original input, the length of the advice is not counted into the space complexity of the respective machine.

**Corollary 1.** *Let $\mathscr{M}$ be a TM/A of space complexity $S(n)$. Then, for any $n$ and $k$ with $S(n) \leq k$, there is an LLM $\mathscr{L}$ using word-to-vector embeddings of size $O(k)$ simulating $\mathscr{M}$ on any input of length $n$.*

*Proof.* (Outline.) Referring to the proof of Theorem 4, one can clearly add $\mathscr{M}$'s advice as an extra input without altering the argument. The advice can be stored in the embeddings used by $\mathscr{L}$ at the cost of adding only a single advice symbol to each embedding. This is the symbol read by $\mathscr{M}$ from its advice tape at the time when $\mathscr{M}$ enters the configuration represented by the respective embedding. Since in each step $\mathscr{M}$ reads at most one advice symbol, all advice symbols read during the computation of $\mathscr{M}$ will fit into embeddings that are available in $\mathscr{L}$ for simulation of $\mathscr{M}$.

The simulation proceeds similar to the proof of Theorem 4. The extension of the embeddings by advice symbols will prolong the size of each embedding of the resulting LLM by one symbol. □

From the proofs it is clear that the same LLM $\mathscr{L}$ will ultimately correctly simulate $\mathscr{M}$ on inputs of every length $n$ as long as $S(n) \leq k$.

It is important to note that the adjustments of any LLM specialized to simulating TMs with or without advice did not put the resulting LLM outside the family of standard LLMs. When compared to LLMs that process a natural language, the necessary adjustments affect just the form of the word embeddings and the working of the attention mechanism. But the main ideas of the LLM architecture, its structure and working, have remained intact. Note that the simulating LLM in the inference phase is fixed and deterministic when fully trained.

The results clearly demonstrate that no single LLM can compute every function that a TM can. *No LLM is Turing complete.* This is because the size of the vector embeddings of the words in the simulating LLM must go hand in hand with the space complexity of the simulated TM, and this is not possible for fixed size embeddings. In fact, it is the consequence of the fact that any LLM is a finite-state machine and a TM generally isn't, and trivially keeps LLMs within the scope of the ECTT.

On the other hand, Theorem 4 and Corollary 1 give evidence of the fact that by specifying more and more 'advanced' TMs, even with advice, and by increasing $n$ and $k$, more and more powerful LLMs can be constructed. It suggests that LLMs can be 'scaled' to match any computational challenge they are up against. This is a possibility that must be anticipated in our further investigation.

It is an open problem whether the simulations from Theorem 4 and Corollary 1 can be improved. For instance, does the model need to explicitly represent the full dictionary of the necessary "Turing machine words"? It seems to depend on the possibilities of the internal model of an LLM. Still, one thing is sure: any simulation of an infinite-state Turing machine by a finite-state machine (like an LLM) is necessarily limited by the lack of computational resources of the latter machine, and therefore is confined to initial segments of computations of the former machines. Luckily, the simulations from Theorem 4 and Corollary 1 are fully sufficient for our further purposes.

## 3.2 Non-uniform computation and lineages of LLMs

By their very definition, the LLMs are interactive computational devices. During their operation, future prompts can react to the answers to the previous prompts. Also, by the results from the previous section, it is conceivable that LLMs are adjusted or modified over time, or even do so themselves when needed or desired. What could, ultimately, be the computational 'reach' of this conception of LLMs?

**Lineages** We model this very general notion of an evolving LLM by a sequence $\mathfrak{L} = \mathscr{L}_1, \mathscr{L}_2, \cdots$ of consecutive LLMs called a *lineage* (after [17]). Each member of such a sequence is specialized in performing computations that require specific 'technical' parameters, such as a specific size of word embeddings, a specific input sizes and so on (like the values of $n$ and $k$ in Theorem 4). So far, this is the standard approach as known in non-uniform complexity theory, e.g. in the study of Boolean circuits or neural networks.

We assume that a lineage of evolving LLMs $\mathfrak{L} = \mathscr{L}_1, \mathscr{L}_2, \cdots$ can process finite but otherwise unbounded streams of inputs as follows. Processing is initiated by $\mathscr{L}_1$. Suppose the processing of the current stream has progressed to LLM $\mathscr{L}_i$, for some $i \geq 1$, and that a trigger of some sort is generated that the lineage must 'switch' to the next 'evolution' $\mathscr{L}_{i+1}$ of the evolving LLM. (The trigger could be a technology update, reaching a memory limit, and so on.) Then the processing is continued by $\mathscr{L}_{i+1}$ *after* it is conditioned with the 'knowledge' built up by $\mathscr{L}_i$, possibly after being 'pre-trained' ahead of time on the input stream that was processed so far. $\mathscr{L}_{i+1}$ only produces answers and responses to the new inputs in the stream as it receives them.

We assume that for every lineage of LLMs $\mathfrak{L} = \mathscr{L}_1, \mathscr{L}_2, \cdots$, the constituent LLMs $\mathscr{L}_i$ are essentially pre-trained and non-adaptive (fixed). Any change or update that is not the result of 'internal' computation is assumed, in principle, to lead to a next LLM in the lineage. The action of constructing and activating a next member of $\mathfrak{L}$ is generally called *model reconstruction*.

Our question about the position of LLMs with respect to the ECTT can now be concretized as follows: what is the position of lineages of evolving LLMs with respect to the ECTT?

**Interactive Turing Machines with Advice** Before answering this question, we need more details about ITM/As. An *interactive TM* (ITM) is a (deterministic multi-tape) Turing machine that operates on a 'stream' of input symbols, supplied at an input port. In this mode inputs are not fixed before the computation starts but new, unforeseen inputs may appear at the input port as the computation proceeds. Inputs may depend on outputs that were produced earlier. Moreover, the processing of a new stream may start from the working tape configuration in which the processing of the previous stream has terminated, if it was indeed finite. For a more detailed description, cf. [15].

Similar to TM/As, *interactive Turing machines with advice* (ITM/As) [15] are ITMs that are extended with an advice facility. In this model, a new advice string may be supplied and appended to the existing advice tape, every time a new input is read and input length is increased by 1. ITM/As arguably are the most general machine model for non-uniform interactive information processing. (Cf. the discussion of the ECTT in Section 1.)

## 3.3 Simulation of ITM/As by lineages of LLMs and Vice Versa

Will simulations as in Theorem 4 and Corollary 1 work also in the extended setting of lineages and ITMs? Of course, as long as the input streams are confined to single members of a lineage and satisfy the fixed assumptions of the theorem and the corollary, the simulations will work. But what happens when the streams fail to satisfy these assumptions, e.g. when streams are not bounded ahead of time? We first focus on the simulation of ITM/As.

To simulate an ITM/A by a lineage of LLMs, we must solve two problems. First, the simulation must consider the fact that the space complexity of the simulated machine may grow with the size of the input, and second, the use of advice (which depends only on the input size) must be taken into account as well. (We consider the simulation on finite but unbounded inputs only, as infinite inputs are not realistic as prompts for LLMs.)

The general idea of the simulation is to simulate computations of the given ITM/A $\mathscr{M}$ per partes by members of a lineage of LLMs, as the individual sequences of configurations of $\mathscr{M}$ unfold, having increasing requirements on the computational resources of the simulating LLMs. Each sequence of configurations is simulated following Corollary 1 by a dedicated member of $\mathscr{L} \in \mathfrak{L}$ as long as the configurations "fit" into the word embeddings of $\mathscr{L}$ and the advice of $\mathscr{M}$ remains unchanged.

**Theorem 5.** *Let $\mathscr{M}$ be an ITM/A. Then, there exists an lineage of evolving LLMs $\mathfrak{L}$ simulating $\mathscr{M}$ on all input streams.*

*Proof.* (Outline) Our starting point is Corollary 1. Initially, assume that $\mathscr{M}$ has space complexity $S(n)$ and that we have chosen a 'trigger' $k_1$ such that $S(n) \leq k_1$ holds for some initial values of $n$, the number of symbols in the input stream so far. Then, by Corollary 1, there exists an LLM $\mathscr{L}_1$ using word-to-vector embeddings of size $O(k_1)$ simulating $\mathscr{M}$ on the input stream for $n$ inputs with $n = 1, 2, \ldots$. However, this simulation may have to come to a halt from two reasons.

First, for some value of $n$, it may appear for the first time that $S(n)$ exceeds $k_1$. This means that a configuration $\rho$ of $\mathscr{M}$ has been reached that no longer fits into the word-to-vector embeddings of size $O(k_1)$. To remedy this situation, we construct a new member $\mathscr{L}_2 \in \mathfrak{L}$ with embeddings of size $k_2 > k_1$. The respective embeddings will contain all configurations of $\mathscr{M}$ of size $k_2$ which are descendants of configuration $\rho$, augmented, of course, with all possible inputs and advice symbols as required in the proof of Corollary 1.

Second, it might happen that for some value of $n$, it still holds that $S(n) \leq k_1$, but that $\mathscr{M}$ gets a new advice as it reaches configuration $\rho$. As before, this calls for a model reconstruction, this time constructing $\mathscr{L}_2 \in \mathfrak{L}$, with word embeddings of a size $k_2$ with $k_2 > k_1$ for all descendants of $\rho$ of size $O(k_2)$ and a new advice string. This also handles the case when both reasons occur simultaneously.

Proceeding inductively in the same way as indicated above, an evolving lineage $\mathfrak{L} = \mathscr{L}_1, \mathscr{L}_2, \cdots$ is obtained that simulates the ITM/A on all finite but unbounded streams. $\square$

We now consider the reverse simulation, of lineages of evolving LLMs by ITM/As. It is the simulation required for the ECTT argument.

Let $\mathfrak{L} = \mathscr{L}_1, \mathscr{L}_2, \cdots$ be a lineage of evolving LLMs. Considering any LLM $\mathscr{L}_i$ in the lineage, it is useful to distinguish between its software and data on the one hand, and the environment in which it runs on the other. Before it 'evolves', we view $\mathscr{L}_i$ as essentially fixed, but its 'environment' can provide external sources that the LLM might use during its computation e.g. for probabilistic purposes. We assume that this provision is independent of the particular input that is processed but part of the 'generic' operation of $\mathscr{L}_i$. The LLM's action is then fully determined by its program and data (and history), if a full description of this interaction of the LLM with its environment over time is given as well. This is exactly what advice does, the rest can be simulated by an interactive Turing machine.

**Theorem 6.** *Let $\mathfrak{L} = \mathscr{L}_1, \mathscr{L}_2, \cdots$ be a lineage of evolving LLMs. Then, there exists an ITM/A $\mathscr{M}$ simulating $\mathfrak{L}$ on all input streams.*

*Proof.* (Outline.) The result follows from the description of how lineages work, provided an ITM/A $\mathscr{M}$ can be designed that, for every $i \geq 1$, will simulate the $i$-th LLM of the lineage whenever this LLM's turn has come, i.e. when the $i$-th switching point is passed in the processing of the input stream.

For $i \geq 1$, let the $i$-th advice of $\mathscr{M}$ be defined to be $D(\mathscr{L}_i)$, a full description of $\mathscr{L}_i$ (including any provision from its environment that applies). On any input stream, if $i$ inputs have been processed, $\mathscr{M}$ calls its advice function to get access to $D(\mathscr{L}_i)$ on its advice tape, enabling it to simulate $\mathscr{L}_i$ when its time in the simulation of the lineage has come. Thanks to the classical Church-Turing thesis this is possible, as $D(\mathscr{L}_i)$ is an algorithmic description of a real digital 'machine'. Hence, $\mathscr{M}$ computes the same transduction as $\mathscr{L}_i$ on its part of the input stream. $\square$

Theorems 5 and 6 can be combined into a single statement as follows.

**Theorem 7.** *For each lineage $\mathfrak{L}$ of evolving LLMs there is an ITM/A $\mathscr{M}$ such that $\mathscr{M}$ simulates $\mathfrak{L}$ on all input streams, and vice versa.*

From the point of view of computational complexity theory, Theorem 7 is significant because it characterizes the computational power of 'evolving LLMs'. Namely, it is known that Turing machines with advice are more powerful than classical TMs, due to the effect of advice (cf. [15]). Therefore, Theorem 7 can be said to express that lineages of LLMs have *'super-Turing' computational power.* By this we do not mean that such lineages can solve undecidable problems. We merely claim that such lineages cannot be simulated by 'classical' ITMs (i.e., ITMs without advice). For a more comprehensive discussion of the computational power of ITM/As, we refer to [15].

## 4 Discussion of the amazing knowledge generation ability of very large finite-state transducers

We now review the results we obtained from a more detached viewpoint, in the broader perspective of related fields like computability, computational complexity theory, AI theory, robotics, and cognitive sciences. The common denominator in our discussion will be to point to the potential of our findings for a better understanding of the essential qualities and limitations of the new emerging information processing technology represented by evolving LLMs. The discussion aims to bring thought-provoking insights, provide novel perspectives to the ongoing debates of LLMs, and challenge future AI research.

**Computability and complexity aspects** In these domains, the main message has been the confirmation that the Extended Church-Turing Thesis is valid also for the latest achievement in the field of IT technology, the development of evolving LLMs. This result could be obtained, thanks to the design of a novel simulation of "small" (resource-bounded) Turing machines entirely within LLMs as in Theorem 4 or Corollary 1. Subsequently, in the end, this has led to the proof of the super-Turing computational power of these AI systems, as a consequence of Theorem 7. Related results comprised a complete characterization of the computational power of single LLMs in Theorem 3, and that of lineages of LLMs in Theorem 7. These results seem to be the first results dealing with the complexity of LLMs from an automata theoretic point of view.

The results are a bit paradoxical – mankind's most complex computational devices turn out to be computationally equivalent to one of the simplest fundamental models of computation, finite-state transducers. LLMs are, in fact, instances of highly resourceful large scalable finite-state transducers.

**The illusory language-processing power of LLMs** In Theorem 3 we saw that the computational power of LLMs is on par with that of FSTs. This raises questions concerning the natural language-processing power of LLMs.

Namely, the languages generated or accepted by FSTs are regular. How it is then possible that, in the practice of LLMs, these devices seem to correctly recognize long sequences of natural languages which in general are known to be more complex than words in a regular language (cf. [10])? This conundrum could be explained by the fact that a finite swath of a language of whatever complexity, captured in the training set, can always be seen as part of a regular language. Beyond this swath, for sufficiently long inputs, the language behaves as a regular language. On the one hand, this explains the apparent inherent efficiency of giant LLMs in processing natural language texts of a reasonable length we see in practice. On the other hand, it also explains the limited abilities of LLMs to deal with tasks not sufficiently represented in the training set, such as simple arithmetic, logical operations like abduction, planning, etcetera.

Nevertheless, prolonging the context window (hence the input length) indefinitely will reveal the true recognition power—that of FSTs—of LLMs, which from a certain internal configuration will start to cycle (or halt). This seems to contradict the recently appearing articles about efficient methods to scale LLMs to infinitely long inputs (cf. [7]). The theoretical catch here is that such methods cannot work in fixed memory spaces like classical LLMs have. They need additional space to enable the long-span attention mechanisms to work. This space may grow with the growing input size. To overcome this difficulty either the full power of ITM/As is required or that of an infinite lineages of evolving LLMs (cf. Theorem 5).

Another problem with viewing LLMs as FSTs is that, in FSTs, the semantics of transitions is encoded entirely in the "relationships" between the states in the state diagram, not in their "names", because the states can be arbitrarily renamed (this seems to be an important observation). On the other hand, in LLMs almost all of the semantics is encaged in the information 'inside' a state. The analogy between relationships among automata states and the semantic processing of language data is hard to see. When thinking about the semantics of words of a natural language, what is of importance is the relationships among the meanings of words, not their "names". Here may also lay the roots of the easiness with which LLMs cross the boundaries between various existing natural languages.

**The problem of understanding** In the domain of AI, the amazingly versatile abilities of LLMs put these systems into the position of the harbingers that announce a paradigm shift afflicting the entire AI ecosystem. Our results contribute to a better apprehension of the nature of the information processing in LLMs. The key observation in this respect is the analogy between natural language processing in LLMs and general computation realization by Turing machines. While natural language processing in LLMs works with finitely many words of a natural language, within a general TM computation each configuration is seen as a word of the "Turing machine language". Such a language has potentially an infinite number of words. Syntax and semantics of this language are described by the underlying TM "program". It describes the relationship between the words generated by the program and, in the end, between the input and the output of the program. In this way, it explains how the program's execution transforms the input to the output. This can be seen as a correctness proof of the program, or as a formal proof of (the machine's) understanding (of what it had done). Of course, this form of understanding is different from what we, humans, understand as understanding.

It is here where the study of programming language theory can inspire, e.g., the ongoing debates on understanding by LLMs (cf. [6]). The analogy between natural language processing by LLMs and the processing of the TM language by a TM may shift the debate to a firm mathematical ground.

**Understanding understanding** To illustrate the difference in understanding in LLMs and TMs, let us compare the "mechanism of understanding" in an LLM processing a natural language, and in an LLM simulating a TM according to Theorem 4. In the former case, the decision to generate the next word of the underlying natural language is based on the limited semantic context based on the vector embeddings of several meticulously chosen words, and the gigantic linguistic background knowledge stored in the form of neural nets. In the latter case, the decision to generate the next word of the Turing machine language (i.e., the next TM configuration) is based on the entire history of computation represented by the sequence of configurations entered by the machine until that time. Moreover, any TM computation makes implicit use of the designer's background knowledge that is already embedded in the design of the underlying TM program. This kind of knowledge is tailored to the intended purpose of the computation.

Which of the two decisions concerning the prolongation of both computations being compared, is based on a more profound knowledge of the situation? The winner is the TM, because its decision is based on the maximal available information it could have directly and indirectly at its disposal.

**Competence without linguistic understanding?** From an evolutionary point of view, it seems that *the key to the notion of understanding is understanding in non-linguistic systems.* "Human-like understanding" adds a layer to the understanding in the AI systems of the latter type. Contemporary wisdom is that human-like understanding is based on concepts – internal mental models of external categories, situations, events, and one's internal state and "self" [6]. LLMs can build internal representations of external categories, situations, and to some extent, one's internal state mediated to the system via textual information. Neural networks are good at building such kinds of representations and excel in verbally expressing them. However, they fail to adequately represent the events and the "self" concept. Representation of events calls for representing the sequences of situations and actions, and the LLMs lack adequate means for doing that. Speaking about the "self" concept is difficult in the case of disembodied entities.

Except for linguistic expressions, non-linguistic embodied AI models can deal with all the concepts mentioned before. Moreover, in the form of multi-modal cyber-physical systems, equipped with memory, sensory, motor, and feedback units, they can do more since their artificial senses are grounded in the real world. Such systems can aspire to represent, and deal with, events and realization of the concept of *"what is it like, for the system, to understand"*. Memory augmenting of AI systems may help to remember, recognize, and recall important events while multi-modality in the form of complementary external and internal sensations allows to represent the last mentioned concept that is considered to be the hallmark of consciousness [8]. Such systems will find themselves on the verge of artificial phenomenal experience (cf. [21, 22]).

Note that the external view of non-linguistic understanding we are discussing above is "competence without comprehension", while the internal view, from the inner perspective of the system, is "what it is like to understand". It may well be that the latter concept presents the missing link even in our understanding to human understanding.

**The inner life of LLMs** There is more to the previous comparison between LLMs and TMs. At each computational step, an embodied TM governing a cognitive cyber-physical system has complete information available not only about its own "current state", from all its external and internal sensors, motors, and the respective feedback from those devices that the system possesses, but also about all its past states.

Note that each configuration of such a TM contains a complete representation of the machine's "phenomenal experience" at that time (cf. [22]). This gives the LLM simulating a TM as in Theorem 4 (that by its very definition remembers all possible "states of mind" of such a machine, one is tempted to say) an opportunity to "time travel" backward and forwards over these states, and thus explore its past decisions or consider its possible "futures" and adjust its behavior accordingly. What never-thought-of possibilities for classical LLMs! This observation is of interest, especially in the context of the recent announcement on a new consensus: there is *"a realistic possibility" for elements of consciousness in reptiles, insects, and mollusks* [4]. If so, why can't it occur in the AI systems whose complexity competes with such simple creatures? Is the feeling "how it is like to understand?" the missing element in our understanding of understanding? In this context, considerations about minimal machine consciousness are the first signs of similar general trends in AI (cf. [21, 22]).

In general, it might be possible to consider various high-level non-linguistic cognitive abilities of LLMs via formal counterparts in the TM environment. For, how else could these abilities be ascribed to LLMs without having their mirror in the language of TMs? In this way, variants of representations of a Turing machine's computations could serve as *drosophila* for ideas about LLMs.

**Building a bridge between rule-based and biological computation** The paradigm shift in our apprehension of computations mentioned above is pregnantly expressed precisely by Theorem 7, which builds a bridge between biologically-inspired and logico-mathematical ways of information processing. Although equipollent from a computability point of view, i.e. expressing the same computational power by different means, the two ways do not have equivalent significance and reach. ITM/As epitomize the classical view of computations, which are seen merely as data transformation tools. The interpretation of the results for ITM/A computations is left to its user. The view of computations through the lens of LLMs puts stress on their semantic contents—it liberates the users from the burden of data interpretation by "automatizing" that task through answers in a natural language. Another view of the respective computations might be that ITM/As mostly capture the processing of non-verbal information, while LLMs capture that of linguistic, verbal information. As Browning and LeCun [2] remind us, *"abandoning the view that all knowledge is linguistic permits us to realize how much of our knowledge is non-linguistic"*. Nevertheless, in both cases, these are but different forms of knowledge that are produced. Viewing computations as knowledge generators has been coined and used by us since the last decade [19, 20, 23].

**Semantics is all we need** Although it may seem that LLMs let us forget about Turing Tests and Chinese Room experiments, the opposite is true. These experiments focus our attention on semantics and understanding, their importance, their representation, processing and interpretation of information that results from computation. The above-mentioned tests and experiments expose the problem of expressing the semantics of computations in their syntax and flow. Perhaps they open the problem of what are the semantic resources of computation, and how are they best represented, utilized and shared. In the case of LLMs, these seem to unequivocally be the word-to-vector embeddings. For ITM/As, the machine configurations. Is there some general theory behind this? In any case, these speculations support the view of computations as knowledge generators (cf. [19, 20]). This view puts stress on the meaning of what is computed, rather than on how a computation is performed.

**Is knowledge computable?** The answer to this question depends on how we define "knowledge" and "computable". There is no one, universally agreed-upon definition of what is knowledge. Within our quest of understanding computation (cf. [15, 18, 19, 20], we see computations as knowledge generators. LLMs are typical examples of computations generating knowledge [23], and especially, wisdom as the agentic form of knowledge. This is a type of knowledge that can be inferred from human-produced texts, programs, pictures, videos, various multimodal sources, and the likes.

If we accept that knowledge is what is generated by computations, and that Turing machines are recognized as the general model of computation in computability theory, then from Theorem 6 it follows that knowledge generation is a non-algorithmic process that cannot be performed by the classical Turing machines. Or, to make knowledge generation computable, shouldn't one redefine the notion of computability using interactive Turing machines with advice?

## 5 Conclusion

The era of interactive non-uniform information processing at scale is here. The Extended Church-Turing Thesis formulated as a vision more than 20 years ago, has appeared to hold for LLMs, too. The information processing by evolving LLMs heralds the advent of the new understanding of computation, and especially of AI. Despite their known deficiencies, the LLMs are wonderful, exciting, and so far quite mysterious information processing devices possessing a maximal computational power like we can expect from massive classical computations. It remains to be seen where and what the new development of LLMs-like devices will bring us in the future. Undoubtedly, the Extended Church-Turing Thesis will cover our steps in this endeavor.

## References

[1] Agüera y Arcas, B., Norvig, P.: Artificial General Intelligence Is Already Here. *NOĒMA*, October 13, 2023, https://www.noemamag.com/artificial-general-intelligence-is-already-here/

[2] Browning, J., LeCun, Y.: AI and the Limits of Language. *NOĒMA*, August 23, 2022, https://www.noemamag.com/ai-and-the-limits-of-language/

[3] Dong, Z., Tang, T., Li, L., Zhao, W.X.: A Survey on Long Text Modeling with Transformers. *arXiv preprint*, arXiv:2302.14502 (2023), https://doi.org/10.48550/arXiv.2302.14502

[4] Lenharo, M.: Do insects have an inner life? Animal consciousness needs a rethink. *Nature*, April 19, 2024, https://www.nature.com/articles/d41586-024-01144-y

[5] Merrill, W., Sabharwal, A.: The Expressive Power of Transformers with Chain of Thought. *arXiv preprint*, arXiv:2310.07923 (2023), https://doi.org/10.48550/arXiv.2310.07923

[6] Mitchell, M., Krakauer, D.C.: The debate over understanding in AI's large language models. *Proceedings of the National Academy of Sciences (PNAS)*, 120 (13) e2215907120, March 23, 2023, https://www.pnas.org/doi/full/10.1073/pnas.2215907120

[7] Munkhdalai, T., Faruqui, M., Gopal, S.: Leave No Context Behind: Efficient Infinite Context Transformers with Infini-attention. *arXiv preprint*, arXiv:2404.07143 (2024), https://doi.org/10.48550/arXiv.2404.07143

[8] Nagel, T.: What Is It Like to Be a Bat? *The Philosophical Review* 83:4 (1974), 435-450, https://doi.org/10.2307/2183914

[9] Nye, M., *et al.*: Show Your Work: Scratchpads for Intermediate Computation with Language Models. *arXiv preprint*, arXiv:2112.00114 (2021), https://doi.org/10.48550/arXiv.2112.00114

[10] Pullum, G.K., Gazdar, G.: Natural languages and context-free languages. *Linguistics and Philosophy* 4 (1982) 471-504, https://doi.org/10.1007/BF00360802

[11] Schuurmans, D.: Memory Augmented Large Language Models are Computationally Universal. *arXiv preprint*, arXiv:2301.04589 (2023), https://doi.org/10.48550/arXiv.2301.04589

[12] Siegelmann, H.T.: *Neural Networks and Analog Computation: Beyond the Turing Limit*. Birkhäuser (1999), Springer Science & Business Media, 2012

[13] Strobl, L., Merrill, W., Weiss, G., Chiang, D., Angluin, D.: What Formal Languages Can Transformers Express? A Survey. *Trans. Assoc. Comput. Ling.* 12 (2024) 543-561, https://doi.org/10.1162/tacl_a_00663

[14] Tai, Y., Dehghani, M., Bahri, D., Metzler, D.: Efficient Transformers: A Survey. ACM Comp. Surv. 55:6 (2022) 1-28, https://doi.org/10.1145/3530811

[15] van Leeuwen, J., Wiedermann, J.: The Turing Machine Paradigm in Contemporary Computing. In: Engquist, B., Schmid, W. (eds), *Mathematics Unlimited - 2001 and Beyond*. Springer, Berlin, Heidelberg (2001), pp. 1139-1155, https://doi.org/10.1007/978-3-642-56478-9_59

[16] Vaswani, A., *et al.*: Attention is All you Need. In: Guyon, I., *et al.* (eds), *Advances in Neural Information Processing Systems* 30 (NIPS 2017), https://doi.org/10.48550/arXiv.1706.03762

[17] Verbaan, P., van Leeuwen, J., Wiedermann, J.: Complexity of Evolving Interactive Systems. In: Karhumäki, J., Maurer, H., Păun, G., Rozenberg, G. (eds), *Theory Is Forever*, Lecture Notes in Computer Science, vol 3113. Springer, Berlin, pp. 268-281 (2004) https://doi.org/10.1007/978-3-540-27812-2_24

[18] Wiedermann, J., van Leeuwen, J. (2008). How We Think of Computing Today. In: Beckmann, A., Dimitracopoulos, C., Löwe, B. (eds) *Logic and Theory of Algorithms. CiE 2008.* Lecture Notes in Computer Science, vol 5028. Springer, Berlin, Heidelberg, pp. 579-593 (2008) https://doi.org/10.1007/978-3-540-69407-6_61

[19] Wiedermann, J., van Leeuwen, J.: Rethinking computation. In: Brown, M., Erden, Y. (Eds), 6th AISB Symp. on Computing and Philosophy: *The Scandal of Computation – What is Computation?*, Proceedings, AISB Convention 2013, University of Exeter, pp. 6-10 (2013), `https://gordana.se/work/PUBLICATIONS-files/2013-PROCEEDINGS-AISB.pdf`

[20] Wiedermann, J., van Leeuwen, J.: What is Computation: An Epistemic Approach. In: G.F. Italiano et al. (eds), *SOFSEM 2015: Theory and Practice of Computer Science*, Lecture Notes in Computer Science, vol. 8939, Springer, Berlin, pp. 1-13 (2015), `https://doi.org/10.1007/978-3-662-46078-8_1`

[21] Wiedermann, J., van Leeuwen, J.: Finite State Machines with Feedback: An Architecture Supporting Minimal Machine Consciousness. In: Manea, F., *et al.* (eds), *Computing with Foresight and Industry*: 15th Conference on Computability in Europe (CiE 2019), Proceedings, Lecture Notes in Computer Science, Vol. 11558, pp. 286-297. Springer, Cham (2019), `https://doi.org/10.1007/978-3-030-22996-2_25`

[22] Wiedermann, J., van Leeuwen, J.: Towards Minimally Conscious Cyber-Physical Systems: A Manifesto. In: Bureš, T., *et al.* (eds), *SOFSEM 2021: Theory and Practice of Computer Science*, Lecture Notes in Computer Science, Vol. 12607, pp. 43-55. Springer, Cham (2021), `https://doi.org/10.1007/978-3-030-67731-2_4`

[23] Wiedermann, J., van Leeuwen, J.: From Knowledge to Wisdom: The Power of Large Language Models in AI, Technical Report UU-PCS-2023-01, Dept. of Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands, 2023, `https://webspace.science.uu.nl/~leeuw112/techreps/UU-PCS-2023-01.pdf`

[24] Wolfram, S.: *What Is ChatGPT Doing... and Why Does It Work?* Wolfram Media, Inc. (2023), `https://doi.org/10.1007/978-3-030-67731-2_4`