EPTCS 151

Proceedings of the

14th International Conference on Automata and Formal Languages

Szeged, Hungary, May 27-29, 2014

Edited by: Zoltán Ésik and Zoltán Fülöp

Published: 21st May 2014
DOI: 10.4204/EPTCS.151
ISSN: 2075-2180
Open Publishing Association

Table of Contents

Preface

The 14th International Conference Automata and Formal Languages (AFL 2014) was held in Szeged, Hungary, from the 27th to the 29th of May, 2014. The conference was organized by the Department of Foundations of Computer Science of the University of Szeged. Topics of interest covered the theory and applications of automata and formal languages and related areas.

The scientific program consisted of invited lectures by

Arnaud Carayol (Marne-la-Vallée),
Markus Holzer (Giessen),
Ondřej Klíma (Brno),
Christof Löding (Aachen),
Sebastian Maneth (Edinburgh)

and 21 short presentations.

This volume contains the texts of the invited lecturers and the 21 papers selected by the International Program Committee from a total of 33 submissions. We would like to thank everybody who submitted a paper to the conference.

The members of the International Program Committee were

Marie-Pierre Béal (Marne-la-Vallée),	Kamal Lodaya (Chennai),
Symeon Bozapalidis (Thessaloniki),	Markus Lohrey (Siegen),
Erzsébet Csuhaj-Varjú (Budapest),	Andreas Maletti (Stuttgart),
Jürgen Dassow (Magdeburg),	Alexander Okhotin (Turku),
Volker Diekert (Stuttgart),	Friedrich Otto (Kassel),
Pál Dömösi (Debrecen, Nyíregyháza),	Giovanni Pighizzini (Milano),
Frank Drewes (Umea),	Libor Polák (Brno),
Zoltán Ésik (Szeged, chair),	Antonio Restivo (Palermo),
Zoltán Fülöp (Szeged, co-chair),	Kai Salomaa (Kingston, ON, Canada),
Viliam Geffert (Kosice),	Pedro V. Silva (Porto),
Oscar H. Ibarra (Santa Barbara, CA, USA),	György Vaszil (Debrecen),
Masami Ito (Kyoto),	Pascal Weil (Bordeaux),
Martin Kutrib (Giessen),	Hsu-Chun Yen (Taipei).

We thank all members of the Program Committee and their subreferees who assisted in the selection of the papers. Special thanks go to the Szent-Györgyi Albert Agora for providing us with the conference facilities, and to our our sponsors, the John von Neumann Computer Society, the Szeged Software Zrt, and the Institute of Informatics of the University of Szeged.

Szeged, May 2014.

Zoltán Ésik and Zoltán Fülöp

Local organization

Zoltán Ésik
Zoltán Fülöp
Éva Gombás
Szabolcs Iván
Zoltán L. Németh
Sándor Vágvölgyi

AFL Steering Committee

A. Ádám (Budapest, chair)
I. Babcsányi (Budapest)
E. Csuhaj-Varjú (Budapest)
P. Dömösi (Debrecen, Nyíregyháza)
Z. Ésik (Szeged)
Z. Fülöp (Szeged)
F. Gécseg (Szeged)
S. Horváth (Budapest)
L. Hunyadvári (Budapest)
L. Kászonyi (Szombathely)
A. Nagy (Budapest)
A. Pukler (Győr)
M. Szijártó (Győr, Székesfehérvár)

List of subreviewers

Dragana Bajic	Szabolcs Iván	Gwenaël Richomme
A. Baskar	Sebastian Jakobi	Chloé Rispal
Beatrice Berard	Artur Jeż	Benoît Rittaud
Mikhail Berlinkov	Antonios Kalampakas	Aleksi Saarela
Johanna Björklund	Stefan Kiefer	Moshe Schwartz
Benedikt Bollig	Manfred Kufleitner	Juraj Sebej
Péter Burcsi	Peter Leupold	Shinnosuke Seki
Michaël Cadilhac	Andreas Malcher	Simoni Shah
Maxime Crochemore	Eleni Mandrali	Magnus Steinby
Judit Csima	Giovanni Manzini	Vaishnavi Sundararajan
Flavio D'Alessandro	Katja Meckel	Louis-Marie Traonouez
Attila Egri-Nagy	Victor Mitrana	Mikhail Volkov
Szilárd Zsolt Fazekas	Frantisek Mraz	Igor Walukiewicz
Gabriele Fici	Benedek Nagy	Qichao Wang
Anna Frid	Laurent Noé	Yuan-Fang Wang
Yo-Sub Han	Beatrice Palano	Abuzer Yakaryilmaz
Markus Holzer	Svetlana Puzynina	Niklas Zechner
Norbert Hundeshagen	George Rahonis	Georg Zetzsche

Saturation algorithms for model-checking pushdown systems[*]

Arnaud Carayol
LIGM
Université Paris-Est & CNRS
arnaud.carayol@univ-mlv.fr

Matthew Hague
Department of Computer Science
Royal Holloway University of London
matthew.hague@rhul.ac.uk

We present a survey of the saturation method for model-checking pushdown systems.

1 Introduction

Pushdown systems have, over the past 15 years, been popular with the software verification community. Their stack can be used to model the call stack of a first-order recursive program, with the control state holding valuations of the program's global variables, and stack characters encoding the local variable valuations. As such the control flow of first-order recursive programs (such as C and Java programs) can be accurately modelled [29]. Pushdown systems have played a key role in the automata-theoretic approach to software model checking and considerable progress has been made in the implementation of scalable model checkers of pushdown systems. These tools (e.g. Bebop [3] and Moped [21, 39, 52, 50]) are an essential back-end components of high-profile model checkers such as SLAM [2].

A fundamental result for the model-checking of pushdown systems was established by Büchi in [12]. He showed that the set of stack contents reachable from the initial configuration of a pushdown system form a regular language and hence can be represented by a finite state automaton. The procedure provided by Büchi to compute this automaton from the pushdown system is exponential. In [15], Caucal gave the first polynomial time algorithm to solve this problem. This efficient computation is obtained by a saturation process where transitions are incrementally added to the finite automaton. This technique, which is the topic of this survey, was simplified and adapted to the model-checking setting by Bouajjani *et al.* in [7] and independently by Finkel *et al.* in [22].

The saturation technique allows global model checking of pushdown systems. For example, one may construct a regular representation of all configurations reachable from a given set of initial configurations, or, dually, the set of all configurations that may reach a given set of target configurations. As well as providing direct solutions to simple reachability properties (e.g. can an error state be reached from a designated initial configuration), the representations constructed by global analyses may be reused in a variety of settings. For example, once may perform multiple (and dynamic) queries on the set of reachable states without having to re-run the model checking routine. Additionally, these representations may be combined as part of a larger algorithm or proof. For example, Bouajjani *et al.* provided solutions to the model checking problem for the alternation free μ-calculus by combining the results obtained through multiple global reachability analyses [7].

In this survey, we present the saturation method under its different forms for reachability problems in Section 3. The saturation technique also generalises to the analysis of two-players games played over the configuration graph of a pushdown systems. This extension based on the work of Cachat [13] and Hague and Ong [28] is presented in Section 4. In Section 5, we review the various model-checking tools that

[*]We thank Didier Caucal and Olivier Serre for helpful discussions. This survey was supported by the Engineering and Physical Sciences Research Council [EP/K009907/1].

implement the saturation technique. We conclude in Section 6 by giving an overview of the extensions of the basic model of pushdown system for which the saturation technique has been applied.

2 Preliminaries

2.1 Finite automata

We denote by Σ^* the set of words over the finite alphabet Σ. For $n \geq 0$, we denote by $\Gamma^{\leq n}$ the set of words of length at most n.

A finite automaton $\mathscr{A}$ over the alphabet Σ is a tuple $(\mathbb{S}, \mathscr{I}, \mathscr{F}, \delta)$ where $\mathbb{S}$ is a finite set of states, $\mathscr{I} \subseteq \mathbb{S}$ is the set of initial states, $\mathscr{F} \subseteq \mathbb{S}$ is the set of final states and $\delta \subseteq \mathbb{S} \times \Sigma \times \mathbb{S}$ is the set of transitions. We write $s \xrightarrow[\mathscr{A}]{a} t$ to denote that (s,a,t) is a transition of $\mathscr{A}$. For a word $w \in \Sigma^*$, we write $s \xRightarrow[\mathscr{A}]{w} t$ to denote the fact that $\mathscr{A}$ can reach the state t while reading the word w starting from the state s. The language accepted by $\mathscr{A}$ from a state s is

$$\mathscr{L}_s(\mathscr{A}) = \left\{ w \in \Sigma^* \;\middle|\; \exists s_f \in \mathscr{F} . s \xRightarrow[\mathscr{A}]{w} s_f \right\}$$

and the language accepted by $\mathscr{A}$ is

$$\mathscr{L}(\mathscr{A}) = \bigcup_{s \in \mathscr{I}} \mathscr{L}_s(\mathscr{A}) \ .$$

2.2 Pushdown system

A pushdown system P is a given by a tuple $(Q, \Gamma, \perp, \Delta)$ where Q is a finite set of control states, Γ is the finite stack alphabet, $\perp \in \Gamma$ is a special bottom of stack symbol and $\Delta \subseteq (Q \times \Gamma) \times (Q \times \Gamma^{\leq 2})$ is the set of transitions. We write $(q,A) \to (p,w)$ for the transition $((q,A),(p,w))$. A configuration is a tuple (q,w) where q is a state in Q and w is a stack content in $(\Gamma \setminus \{\perp\})^* \perp$. In the configuration $c = (q,Aw)$, the pushdown system can apply the transition $(q,A) \to (p,u)$ to go to the configuration $c' = (p,uw)$. As is usual, we assume that transitions of the pushdown system does not pop the bottom of stack symbol or does not push it on the stack (*i.e.* all transitions involving the symbol $\perp$ are of the form $q\perp \to p\perp$ or $q\perp \to p\perp A$ for some $A \in \Gamma \setminus \{\perp\}$). We denote by $\xrightarrow[P]{}$ (or simply $\to$ if P is clear from the context) the relation on configurations defined by the transitions of P. We denote by $\xRightarrow[P]{}$ the reflexive and transitive closure of $\xrightarrow[P]{}$.

3 Reachability problems for pushdown systems

A fundamental result for the model-checking of pushdown systems is the fact that the set of stack contents:

$$\{w \in \Gamma^* \mid \exists q \in Q, (q_0, \perp) \Rightarrow (q,w)\}$$

that are reachable from an arbitrary initial configuration of the system, form a regular set of words over the stack alphabet Γ.

A more elegant formulation of this result can be obtained by extending the notion of regularity to sets of configurations. A set of configurations C is *regular* if for every state $p \in Q$, the set of associated stack contents $\{w \in \Gamma^* \mid (p,w) \in C\}$ is regular. A P-automaton is a slight extension of the standard notion of finite automaton to accept configurations. The only extra assumption is that the set of states of the P-automaton contains the set of states of the pushdown system. Formally, a P-automaton is of the form $(\mathbb{S}, Q, \mathscr{F}, \delta)$ where Q is the set of states of the pushdown system P. A configuration is (p, w) is accepted by $\mathscr{A}$ if w is accepted by $\mathscr{A}$ starting from the state p (*i.e.* $w \in L_p(\mathscr{A})$).

Theorem 1 *[12] The set of configurations of a pushdown system reachable from the initial configuration (*i.e. *the configuration* $(q_0, \perp)$ *for some arbitrary state* q_0*) is regular. Moreover a P-automaton accepting it can be effectively constructed from the pushdown system.*

To the authors knowledge, the first proof of this result is due to Büchi in [12]. The formalism used by Büchi is not that of pushdown automata but that of prefix word-rewriting systems (which he calls *regular canonical systems*). These systems syntactically include pushdown automata and conversely can be simulated by pushdown automata. In [23], Greibach formalises the correspondence between the two models and gives a simple proof based on a result on context-free languages proved by Bar-Hillel *et al.* in [4]. Greibach also says that the result (for pushdown automata) was part of the folklore at the time but never appeared in print. Even though effective, these proofs do not provide a polynomial time algorithm[1]. The first polynomial time algorithm is due to Caucal [15, 16] which is based on a saturation procedure of a finite state automaton. The idea behind the saturation method can be traced back to [5]. This method was independently rediscovered and used for model-checking purposes by Bouajjani *et al.* in [7] and Finkel *et al.* in [22].

A more general problem is, given a regular set of configurations C, to compute the set:

$$Post^*_P(C) = \{c' \mid \exists c \in C, c \underset{P}{\Longrightarrow} c'\}$$

of configurations that can be reached from a configuration in C.

The regularity of $Post^*(C)$, for any regular set C, can be derived from Theorem 1. Indeed starting from a pushdown system P and a regular set of configurations C, we can create a new pushdown system P' which using new states builds any configuration in C and afterwards behaves like P. Clearly the set of configurations reachable from the initial configuration of P' coincide with $Post^*_P(C)$ when restricted to the states of P.

As mentioned in the introduction, for model-checking purposes it is often interesting to compute the set of configurations that can reach a given set of *bad* configurations. This leads to consider the set

$$Pre^*_P(C) = \{c' \mid \exists c \in C, c' \Rightarrow c\}$$

of configurations that can reach a configuration in C.

The regularity of $Pre^*(C)$ for any regular set C can be deduced from the regularity of $Post^*(C)$. The intuitive idea is to construct, from P, a new pushdown system P' whose derivation relation is the inverse of that of P. For a transition of the form $qA \to p$ of P, we add the transitions $pX \to qAX$ for all symbols $X \in \Gamma$. For a transition $qA \to pBC$ of P, we add two transition $pB \to r_{(C,q,A)}$ and $r_{(C,q,A)}C \to qA$ where $r_{(C,q,A)}$ is a new intermediary control state. For any two configurations c and c' of P, it holds that $c \Rightarrow_P c'$ if and only if $c' \Rightarrow_{P'} c$. Hence $Pre^*_P(C)$ is equal to the restriction of $Post^*_{P'}(C)$ to the states of P and is therefore regular.

The section is structured as follows. We present Büchi's original proof in Section 3.1. In Section 3.2, we present the saturation algorithm to compute $Pre^*(C)$ introduced in [7]. Finally in Section 3.3, we characterise the derivation relation of the pushdown automata using the saturation technique following [15].

[1]We will see Section 3.1 that it can easily be adapted to provide a polynomial time algorithm.

3.1 Büchi's proof

We present a proof of Theorem 1 adapted from [12]. In the original proof, Büchi first reduced the problem to a very simple form of pushdown system where transitions are either of the form $pA \to q$ or $p \to qA$. This model (called *reduced regular systems* by Büchi) is completely symmetric and therefore computing Pre^* or $Post^*$ is essentially the same thing. However to adapt the proof to the formalism used in this article (recall that our formalism does not allow rules of the form $p \to qA$), it is more convenient to work with Pre^* than with $Post^*$.

Given a pushdown system $P = (Q, F, \perp, \Delta)$, we construct a P-automaton accepting $Pre^*_P(\{(q_f, \perp)\})$ where q_f is an arbitrary *final* state of the pushdown system.

The construction is based on the following remark: to reach the configuration $(q_f, \perp)$ from a configuration $(p, Aw\perp)$ it is necessary, at some point, to reach a configuration of the form $(q, w\perp)$ for some state $q \in Q$. Moreover the first time such a configuration is reached, the actions taken by P cannot depend on w since at no point was w exposed at the top of the stack. Hence it must be the case that $pA \Rightarrow q$.

The P-automaton when accepting a stack content $A_1 \ldots A_n\perp$ from the state p will guess the states $p_1, \ldots, p_n$ such that $pA_1 \underset{P}{\Longrightarrow} p_1$ and $p_iA_{i+1} \underset{P}{\Longrightarrow} p_{i+1}$ for $i \in [0, n-1]$ and will enter a final state upon reading the symbol $\perp$ if $p_n\perp \underset{P}{\Longrightarrow} q_f\perp$.

Consider the P-automaton $\mathscr{A}$ with set of states $Q \cup \{s_\perp\}$ where $s_\perp$ is a new state and the only final state of the automaton. The transitions of the automaton $\mathscr{A}$ are defined as follows:

- $p \xrightarrow{A} q$ if and only if $pA \underset{P}{\Longrightarrow} q$ for all $p, q \in Q$ and $A \in \Gamma \setminus \{\perp\}$,
- $p \xrightarrow{\perp} s_\perp$ if and only if $p\perp \underset{P}{\Longrightarrow} q_f\perp$ for all $p \in Q$.

A simple induction on the length of the stack content shows that $\mathscr{A}$ accepts a stack content $w\perp$ from the state $q \in Q$ if and only if $(q, w\perp)$ belongs to $Pre^*(\{(q_f, \perp)\})$.

To make the construction effective, it remains to compute the relations $pA \Rightarrow q$ and $p\perp \Rightarrow q\perp$ for all states p and $q \in Q$ and stack symbol $A \in \Gamma$. The procedure provided by Büchi is exponential[2]. He first establishes a bound on the height of the stack necessary to build a derivation path witnessing these relations. As the bound is polynomial in the size of the pushdown system, the problem is reduced to a simple reachability problem in a finite graph of exponential size with respect to the size of the pushdown system.

To obtain a polynomial algorithm, it is enough to efficiently compute the relation $Rew = \{(pA, qB) \mid pA \underset{P}{\Longrightarrow} qB\}$. Indeed $pA \underset{P}{\Longrightarrow} q$ if and only if there exists $r \in Q$ and $B \in \Gamma$ such that $pA \underset{P}{\Longrightarrow} rB$ (*i.e.* $(p, A, r, B) \in \text{Rew}$) and $rB \to q$ is a transition of P.

The key idea which is at the heart[3] of the saturation algorithm presented in Section 3.2 is to express Rew as a smallest fixed-point.

The relation Rew is the smallest relation (for the inclusion) in $Q\Gamma \times Q\Gamma$ such that:

- $(pA, pA) \in Rew$ for all $p \in Q$ and $A \in \Gamma$,

[2]In [12], the P-automaton constructed is deterministic (essentially the automaton obtained by applying the power-set construction to the automaton presented here). With the added constraint of determinism, it not possible to obtain a polynomial algorithm as the smallest deterministic automaton is in general exponential in the size of the pushdown system. To convince oneself, it is enough to consider a pushdown system that simulates a non-deterministic finite state automaton (NFA) by popping its stack until the bottom of the stack is reached and when the bottom of the stack is reached goes to the state q_f if the NFA has reached a final state.

[3]We will see that the algorithm presented in Section 3.2 performs a fixed-point computation for the relation $\{(pA, q) \mid pA \underset{P}{\Longrightarrow} q\}$.

- $(pA,qB) \in \text{Rew}$ if $pA \to qB$ is a transition of P,
- $(pA,qC) \in \text{Rew}$ if $(pA,rB) \in \text{Rew}$ and $(rB,qC) \in \text{Rew}$,
- $(pA,qC) \in \text{Rew}$ if $pA \to rBC$ is a transition of P and there exists $t \in Q$ and $D \in \Gamma$ such that $(rB,tD) \in \text{Rew}$ and $tD \to q$ is a transition of P.

The property (1) expresses that Rew is reflexive and (3) that it is transitive. Property (2) ensures that Rew contains the relevant transitions of P. Property (4) describes the case when $pA \underset{P}{\Longrightarrow} qC$ is obtained by a sequence of the form $pA \underset{P}{\to} rBC \underset{P}{\Longrightarrow} qC$ where $rB \underset{P}{\Longrightarrow} q$.

Using the Knaster-Tarski theorem, we can compute Rew as the limit of an increasing sequence of relations $(\text{Rew}_i)_{i\geq 0}$ over $Q \times \Gamma$. The relation Rew_0 contains the elements satisfying property (1) and (2). The relation Rew_{i+1} is obtained from Rew_i by adding all the elements that can be derived by property (3) or (4) in Rew_i. The sequence $(\text{Rew}_i)_{i\geq 0)}$ is increasing for the inclusion and its limit (*i.e.* the first set such that $\text{Rew}_{i+1} = \text{Rew}_i$) is equal to Rew. As at least one element is added at each step before the limit is reached, the limit is reached in at most $|Q|^2|\Gamma|^2$ steps. Furthermore as the computation of Rew_{i+1} from Rew_i can be done in polynomial time with respect to the size of P, the resulting algorithm is polynomial. However the exact complexity is not as good as the algorithm presented in Section 3.2.

3.2 Saturation algorithm of [7]

In [7], Bouajjani *et al.* present an algorithm that given a pushdown system $P = (Q,\Gamma,\bot,\Delta)$ and a P-automaton $\mathscr{A} = (\mathbb{S},Q,\delta,\mathscr{F})$, constructs a new P-automaton $\mathscr{B}$ accepting $Pre^*_P(\mathscr{L}(\mathscr{A}))$. The only requirement on $\mathscr{A}$ is that no transition in δ goes back to a state in Q[4]. This restriction also implies that none of the states in Q are final.

The algorithm proceeds by adding transitions to $\mathscr{A}$ following a unique rule until no new transition can be added. The resulting P-automaton $\mathscr{B}$ accepts the set of configurations $Pre^*_P(\mathscr{L}(\mathscr{A}))$.
More precisely, the algorithm constructs a finite sequence $(\mathscr{A}_i)_{i\in[0,N]}$ of P-automata. The P-automaton $\mathscr{A}_0$ is the automaton $\mathscr{A}$. All the P-automata $\mathscr{A}_i$ are of the form $(\mathbb{S},Q,\mathscr{F},\delta_i)$, meaning that they only differ by their set of transitions. The construction guaranties that for all $i \in [0,N-1]$, $\delta_i \subseteq \delta_{i+1}$ and terminates when $\delta_{i+1} = \delta_i$. As at least one transition is added at each step, the algorithm terminates in at most $|Q|^2|\Gamma|$ steps.

The set of transitions δ_{i+1} is obtained by adding to δ_i, the transition:

$$p \xrightarrow{A} s \text{ if } q \underset{\mathscr{A}_i}{\overset{w}{\Longrightarrow}} s \text{ and } pA \to qw \text{ is a transition of } P.$$

Note that only transitions starting with a state of Q are added by the algorithm. In particular, the language accepted the automaton $\mathscr{A}_i$ from a state in $\mathbb{S} \setminus Q$ never changes.[5].

The construction of δ_{i+1} from δ_i ensures that the configurations that can reach in one step a configuration in $\mathscr{L}(\mathscr{A}_i)$ belong to $\mathscr{L}(\mathscr{A}_{i+1})$. Consider two configurations $c = (p,Au)$ and $c' = (q,wu)$ such that $pA \to qw$ is a transition of P (and hence $c \underset{P}{\to} c'$). Now assume that c' belongs to $\mathscr{L}(\mathscr{A}_i)$. This means that for some state $s \in \mathbb{S}$ and some final state $s_f \in \mathscr{F}$, $q \underset{\mathscr{A}_i}{\overset{w}{\Longrightarrow}} s \underset{\mathscr{A}_i}{\overset{u}{\Longrightarrow}} s_f$. The rule of construction of δ_{i+1} ensures that $p \xrightarrow{A} s$ is a transition of $\mathscr{A}_{i+1}$. Hence $p \underset{\mathscr{A}_{i+1}}{\overset{A}{\Longrightarrow}} s \underset{\mathscr{A}_{i+1}}{\overset{u}{\Longrightarrow}} s_f$ and the configuration $c = (p,Au)$ is accepted by $\mathscr{A}_{i+1}$. As $\mathscr{B}$ is the limit of the saturation process (*i.e.* $\mathscr{B} = \mathscr{A}_{N-1} = \mathscr{A}_N$), $\mathscr{L}(\mathscr{B})$ is closed under taking

[4]This requirement is easily met by adding a copy of each state in Q if necessary. This restriction is required to ensure that the first invariant maintained by the algorithm holds initially.

[5]Recall that initially the states in Q are not the target of any transition

the immediate predecessor for the relation $\underset{P}{\rightarrow}$ (*i.e.* if $c' \in \mathscr{L}(\mathscr{B})$ and $c \underset{P}{\rightarrow} c'$ then $c \in \mathscr{L}(\mathscr{B})$). As $\mathscr{L}(\mathscr{B})$ includes $\mathscr{L}(\mathscr{A})$, it follows that $Pre^*_P(\mathscr{L}(\mathscr{A})) \subseteq \mathscr{L}(\mathscr{B})$.

The proof of the converse inclusion requires a more careful analysis. The algorithm maintains two invariants on the transitions in δ_i. For all $i \in [0,N]$, the presence of a transition $p \xrightarrow{A} s$ in δ_i guaranties that:

1. $pA \underset{P}{\Longrightarrow} s$ if s belongs to Q.
2. the configuration (p, Au) belongs to $Pre^*(\mathscr{L}(\mathscr{A}))$ for any $u \in \mathscr{L}_s(\mathscr{A}_i) = \mathscr{L}_s(\mathscr{A})$ if s belongs to $\mathbb{S} \setminus Q$.

From these invariants, it follows that for all $i \geq 0$, $\mathscr{L}(\mathscr{A}_i) \subseteq Pre^*_P(\mathscr{L}(\mathscr{A}))$. In particular, $\mathscr{L}(\mathscr{B}) \subseteq Pre^*_P(\mathscr{L}(\mathscr{A}))$.

Remark 1 *As indicated by the first invariant, if we restrict our attention to transitions with both source and target in Q, this algorithm is performing a fixed-point computation for the relation* $\underset{P}{\Longrightarrow}$ *restricted to* $(Q \times \Gamma) \times (Q \times \{\varepsilon\})$. *Indeed this relation can be characterised as the smallest relation (for the inclusion)* $\mathscr{R}$ *such that:*

1. *$pA \mathscr{R} q$ if $pA \rightarrow q$ belongs to Δ,*
2. *$pA \mathscr{R} q$ if $rB \mathscr{R} q$ and $pA \rightarrow rB$ belongs to Δ,*
3. *$pA \mathscr{R} q$ if $pA \rightarrow rBC$ belongs to Δ and for some state $s \in Q$, $rB \mathscr{R} s$ and $sC \mathscr{R} q$.*

In fact, the algorithm performs the computation of the smallest such relation following the procedure given by Knaster-Tarski theorem.

A naive implementation of this algorithm yields a complexity in $\mathscr{O}(|P|^2|\mathscr{A}|^3)$. However a more efficient implementation presented in [20] lowers the complexity to $\mathscr{O}(|Q|^2|\Delta|)$.

In [20], an adaptation of the algorithm for computing Pre^* is given to compute $Post^*$. The algorithm is slightly less elegant as it requires the addition of new states before the saturation process. In fact, it is very similar to first applying the transformation to invert the pushdown system presented at the beginning of this section and then applying the algorithm to compute Pre^*.

In [39], Schwoon shows how to use the saturation algorithm to construct for any configuration c accepted by $\mathscr{B}$ a derivation path to some configuration in $\mathscr{L}(\mathscr{A})$.

3.3 Derivation relation of a pushdown system

In this section, we will see that the saturation method can be adapted to characterise the derivation relation of a pushdown system. Let us fix a pushdown system[6] $P = (Q, \Gamma, \Delta)$, an initial state q_0 and a final state q_f. We aim at giving an effective characterisation of the following relation between stacks:

$$\mathrm{Deriv}_P = \{(u,v) \in \Gamma^* \mid (q_0, u) \underset{P}{\Longrightarrow} (q_f, v)\}.$$

In [15], Caucal showed that $\mathrm{Deriv}_P \subseteq \Gamma^* \times \Gamma^*$ is a rational relation, *i.e.* it is accepted by a finite state automaton with output (also called a transducer).

The proof presented here is based on [17] but similar ideas can be found in [38, 22]. The idea of the proof is to use symbols to represent the actions of the pushdown system on the stack: one symbol for

[6]To simplify the presentation, we do not take the bottom of stack symbol into account.

pushing a given symbol and one symbol for popping it. The pushdown system is transformed into a finite state automaton that instead of performing the actions on the stack outputs the symbols that represent these actions (see Section 3.3.1). This finite state automaton is then transformed using a saturation algorithm so that it erases sequences of actions corresponding to pushing a symbol and then immediately popping it (see Section 3.3.2). From this *reduced* language, the relation Deriv_P is easily characterised (see Section 3.3.3).

3.3.1 Sequences of stacks actions

For every symbol $A \in \Gamma$, we introduce two symbols:

- A_+ which represents the action of pushing the symbol A on top of the stack,
- and A_- which represents the action of popping the symbol A from the top of the stack.

We denote by Γ_+ the set $\{A_+ \mid A \in \Gamma\}$ of *push* actions, by Γ_- the set $\{A_- \mid A \in \Gamma\}$ of *pop* actions and by $\overline{\Gamma}$ the set $\Gamma_+ \cup \Gamma_-$ of all action symbols.

Intuitively a sequence $\alpha = \alpha_1 \dots \alpha_n \in \overline{\Gamma}^*$ is interpreted as performing the action α_1, followed by the action α_2 and so on. For instance, the effect on the stack of the transition $pA \to qBC$ is represented by the word $A_-C_+B_+$. First the automaton removes the A from the top of the stack and then pushes C and then B.

For two stacks u and $v \in \Gamma^*$, we write $u \overset{\alpha}{\rightsquigarrow} v$ if u can be transformed into v by the sequence of actions α. For instance, we have $ABB \overset{\alpha}{\rightsquigarrow} DCB$ for the α sequence $A_-B_-C_+D_+$. Note that some sequences of actions such as B_+C_- cannot be applied to any stack. We say that such sequences α are *non-productive*, *i.e.* there are no u and $v \in \Gamma^*$ such that $u \overset{\alpha}{\rightsquigarrow} v$.

From the pushdown system P, we can construct a regular set of action sequences denoted $\mathrm{Behaviour}_P$ which contains all the sequences (even the non-productive ones) that can be performed by P when starting in state q_0 and ending in state q_f. Consider for instance the finite state automaton[7] $(Q, \{q_0\}, \{q_f\}, \delta)$ where the set of transitions δ is given by:

$$\begin{cases} p \xrightarrow{A_-C_+B_+} q \in \delta \quad \text{if} \quad pA \to qBC \in \Delta \\ p \xrightarrow{A_-B_+} q \in \delta \quad \text{if} \quad pA \to qB \in \Delta \\ p \xrightarrow{A_-} q \in \delta \quad \text{if} \quad pA \to q \in \Delta \end{cases}$$

It is clear that $\mathrm{Behaviour}_P$ characterises Deriv_P in the following sense:

$$(u,v) \in \mathrm{Deriv}_P \quad \text{if and only if} \quad u \overset{\alpha}{\rightsquigarrow} v \ \text{ for some } \alpha \in \mathrm{Behaviour}_P.$$

However this representation of Deriv_P is not yet very helpful. For instance, $\mathrm{Behaviour}_P$ can contain non-productive sequences or sequences such as $A_-B_+A_+A_-C_+C_-$ which is equivalent to the more informative sequence A_-B_+.

[7]The finite state automaton does not strictly conform to the definition we gave in Section 2 as its transitions are labelled by words and not single letters. This can be easily avoided at the cost of adding intermediate states.

3.3.2 Reducing sequences of actions

To simplify Behaviour_P, we first erase all factors of the form A_+A_- for $A \in \Gamma$. These factors can safely be omitted as they do not affect the stack: the symbol is pushed then immediately popped. A sequence that does not contain any such factors is called *reduced*.

To perform this erasure, we introduce the relation $\mapsto$ which relates a stack $u \in \Gamma^*$ and a stack $v \in \Gamma^*$ if v can be obtained by erasing a factor A_+A_- from u (*i.e.* $u = u_1A_+A_-u_2$ and $v = u_1u_2$). Clearly, if $\alpha \mapsto \beta$ then the sequences α and β are equivalent with respect to their actions on the stack :

$$\text{for } u, v \in \Gamma^*,\ u \overset{\alpha}{\rightsquigarrow} v \text{ if and only if } u \overset{\beta}{\rightsquigarrow} v.$$

As the rewriting relation $\mapsto$ is confluent and decreases the length of the sequence, every sequence α can be iteratively rewritten by $\mapsto$ into a reduced sequence denoted $\text{Red}(\alpha)$. For instance the reduced sequence associated to $B_-A_+A_+A_-A_-C_+$ is B_-C_+ as $B_-A_+A_+A_-A_-C_+ \mapsto B_-A_+A_-C_+ \mapsto B_-C_+$.

In [5], Benois showed[8] that the set of reduced sequences corresponding to a regular set of sequences is again regular.

Theorem 2 *[5, 6] For any regular set R of action sequences, the corresponding set of reduced action sequences:*

$$\text{Red}(R) = \{\text{Red}(\alpha) \mid \alpha \in R\}$$

is regular. Moreover given a finite automaton $\mathscr{A}$ accepting R, an automaton accepting $\text{Red}(R)$ *can be constructed in $\mathcal{O}(|\mathscr{A}|^3)$.*

The proof of this theorem is the essence of the saturation method. Starting with the automaton $\mathscr{A}$, ε-transitions are added until no new ε-transition can be added. The ε-transitions are added according to the following rule. We add an ε-transition from a state p to a state q if it is possible to reach q from p reading a word of form $A_+\varepsilon^*A_-$. It can be shown that the resulting saturated automaton accepts the language:

$$\{\beta \in \overline{\Gamma}^* \mid \alpha \mapsto^* \beta \quad \text{for some } \alpha \in R\}.$$

The construction is concluded by taking the ε-closure of the saturated automaton and restricting the language to the set of reduced sequences (which is a regular language as it is the complement of the language $\cup_{A\in\Gamma}\overline{\Gamma}^*A_+A_-\overline{\Gamma}^*$). A careful implementation of the procedure presented in [6] gives an algorithm in $\mathcal{O}(|\mathscr{A}|^3)$.

3.3.3 Characterisation of Deriv_P

One of the advantages of working with $\text{Red}(\text{Behaviour}_P)$ is that we can easily remove non-productive sequences. Indeed a reduced sequence is non-productive if and only if it contains a factor of the form A_+B_- for $A \neq B \in \Gamma$.

We can hence compute the regular language:

$$RP_P = \text{Red}(\text{Behaviour}_P) \cap \left(\overline{\Gamma}^* \setminus \bigcup_{A\neq B\in\Gamma} \overline{\Gamma}^* A_+B_-\overline{\Gamma}^* \right)$$

which is composed of the reduced and productive action sequences characterising Deriv_P.

[8] Benois consider the erasure of all factor of the form A_-A_+ as well as A_+A_- but the proof is identical.

The language RP_P does not contain any factor in $\Gamma_+\Gamma_-$ and is hence included in $\Gamma_-^*\Gamma_+^*$. We can express it as a finite union:

$$\bigcup_{i\in[1,N]} X_i Y_i$$

where for all $i \in [1,N]$, X_i is a regular language in Γ_-^* and Y_i is a regular language in Γ_+^*.

Let us denote by U_i the regular set $\{A^1\cdots A^n \in \Gamma^* \mid \mid A_-^1 \cdots A_-^n \in X_i\}$ of words in Γ^* that can be popped by a sequence in X_i and by V_i the regular set $\{A^1\cdots A^n \in \Gamma^* \mid \mid A_+^n \cdots A_+^1 \in Y_i\}$ of words in Γ^* that can be pushed by a sequence in Y_i.

The relation Deriv_P can be characterised as follows: a pair (w_1,w_2) belongs to Deriv_P, if for some $i \in [1,N]$, w_1 can be written as uw with $u \in U_i$ and w_2 can be written as vw for some $v \in V_i$. In other terms, the relation Deriv_P can be written as a finite union of relations that remove a prefix of the stack belonging to a certain regular language and replace any word in another regular language. As these relations are easily accepted by finite transducer, so is Deriv_P. Combining all the steps, we obtain a polynomial time algorithm for computing a transducer accepting Deriv_P from P.

4 Winning regions of pushdown games

The saturation technique also generalises to the analysis of pushdown games with two players: Éloise and Abelard. The two players may, for example, model a program (Éloise) interacting with the environment (Abelard). While the program can control its next move based on its internal state, it cannot control the results of requesting external input. Hence, the external input is decided by the second player.

A pushdown game may be used to analyse various types of properties. We will consider three, increasingly expressive, types of properties here: reachability, Büchi and parity. We will begin by defining games with generic winning conditions and then consider the instantiations of this generic framework for each winning condition in turn. We will simultaneously discuss the saturation algorithm for each of these properties and show how they build upon each other.

The saturation algorithm was first extended to pushdown reachability games by Bouajjani *et al.* [7]. Their algorithm was extended to the case of Büchi games by Cachat [13] and then to parity games by Hague and Ong [28]. Our presentation will follow that of Hague and Ong since it provides the most general algorithm, though we remark that all the essential ideas of the algorithm were in place by the introduction of the Büchi algorithm. The main contribution of Hague and Ong was a proof framework that simplified the technical arguments by Bouajjani *et al.* and Cachat and allowed the full parity case to go through.

4.1 Preliminaries

4.1.1 Pushdown games

We can obtain a two-player game from a pushdown system P by the addition of two components: a partition of the configurations of P into positions controlled by Éloise and positions controlled by Abelard; and the definition of a winning condition that determines the winner of any given play of the game.

In the following, for technical convenience, we will assume for each $q \in Q$ and $A \in \Gamma$ there exists some $(q,A) \rightarrow (p,w) \in \Delta$. Together with the bottom-of-stack symbol, this condition ensures that from a configuration $(q,w\perp)$ it is not possible for the system to become stuck; that is, reach a configuration with no successor.

A two-player pushdown game is a tuple $P = (Q, \Gamma, \bot, \Delta, W)$ such that $(Q, \Gamma, \bot, \Delta)$ defines a pushdown system, Q is partitioned $Q = Q_E \uplus Q_A$ into Éloise and Abelard positions respectively, and W is a set of infinite sequences of configurations of P.

A play of a pushdown game is an infinite sequence $(q_0, w_0), (q_1, w_1), \ldots$ where (q_0, w_0) is some starting configuration and (q_{i+1}, w_{i+1}) is obtained from (q_i, w_i) via some transition $(q_i, A) \to (q_{i+1}, w) \in \Delta$. In the case where $q_i \in Q_E$ it is Éloise who chooses the transition to apply, otherwise Abelard chooses the transition.

The winner of an infinite play $(q_0, w_0), (q_1, w_1), \ldots$ is Éloise if $(q_0, w_0), (q_1, w_1), \ldots \in W$; otherwise, Abelard wins the play. The winning region $\mathscr{W}$ of a pushdown game is the set of all configurations from which Éloise can always win all plays, regardless of the transitions chosen by Abelard.

4.1.2 Alternating automata

To extend the saturation algorithm to compute the winning region of a pushdown game, we augment the automata used to recognise sets of configurations with alternation. Bouajjani *et al.* first used alternating automata to analyse pushdown reachability games via saturation [7], however, they used the equivalent formalism of *alternating pushdown systems* rather than pushdown games. An alternating automaton is a tuple $\mathscr{A} = (\mathbb{S}, \Gamma, \mathscr{F}, \delta)$ where $\mathbb{S}$ is a finite set of states, Γ is a finite alphabet, $\mathscr{F} \subseteq \mathbb{S}$ is the set of accepting states, and $\delta \subseteq \mathbb{S} \times \Gamma \times 2^{\mathbb{S}}$ is a transition relation. Note that we do not specify a set of initial states. This is because it is more convenient to present the following results in terms of the stacks accepted from particular states, rather than fixing a set of initial states.

Whereas a transition $s \xrightarrow{A} t$ of a non-deterministic automaton requires the remainder of the word to be accepted from t, a transition $s \xrightarrow{A} S$ of an alternating automaton requires that the remainder of the word is accepted from all states $s' \in S$. It is this "for all" condition that captures the fact that Éloise must be able to win for all moves Abelard may make.

More formally, a run over a word $A_1 \ldots A_n \in \Gamma^*$ from a state s_0 is a sequence

$$S_1 \xrightarrow{A_1} \cdots \xrightarrow{A_n} S_{n+1}$$

where each S_i is a set of states such that $S_1 = \{s_0\}$, and for each $1 \leq i \leq n$ we have

$$\forall s \in S_i . \exists s \xrightarrow{A_i} S \in \delta \wedge S \subseteq S_{i+1} \ .$$

The run is accepting if $S_{n+1} \subseteq \mathscr{F}$. Thus, for a given state s, we define $\mathscr{L}_s(\mathscr{A})$ to be the set of words over which there is an accepting run of $\mathscr{A}$ from $\{s\}$.

When S_i is a singleton set, we will often omit the set notation. For example, the run above could be written

$$s_0 \xrightarrow{A_1} \cdots \xrightarrow{A_n} S_{n+1} \ .$$

Further more, when $w = A_1 \ldots A_n$ we will write $s \xrightarrow{w} S$ as shorthand for a run from s to S.

4.2 Pushdown reachability games

One of the simplest winning conditions for a game is the reachability condition. Given a target set of configurations C, the reachability condition states that Éloise wins the game from a given configuration if she can force all plays starting at that configuration to some configuration in C.

That is, a pushdown reachability game is a tuple $(Q, \Gamma, \bot, \Delta, C)$ such that (Q, Γ, Δ, W) is a pushdown game where

$$W = \{c_0, c_1, \ldots \mid \exists i. c_i \in C\}$$

is the set of all sequences of configurations containing some configuration in C.

4.2.1 Characterising the winning region

In the sequel we will need to combine least and greatest fixed points. We will use μ to denote the least fixed point operator, and ν to denote the greatest fixed point operator.

In the simple case of reachability for a pushdown system P and set of target configurations C we can characterise the winning region $\mathscr{W} = Pre^*_P(C)$ as

$$\mu Z. C \cup Pre_P(Z)$$

where

$$Pre_P(Z) = \left\{ (p, w) \;\middle|\; \begin{array}{lcll} p \in Q_E & \Rightarrow & \exists (p,w) \to c.\ c \in Z & \wedge \\ p \in Q_A & \Rightarrow & \forall (p,w) \to c.\ c \in Z & \end{array} \right\} .$$

That is, to appear in $\mathscr{W}$ for a configuration belonging to Éloise, it must be possible for her to choose a transition that progresses towards C. For configurations belonging to Abelard, it must be the case that he cannot help but choose a transition that progresses towards C.

4.2.2 Computing the winning region

Fix a pushdown reachability game $P = (Q, \Gamma, \Delta, C)$. We will show how to construct an automaton $\mathscr{B}$ whose state set includes the state p for all $p \in Q$ and $w \in \mathscr{L}_p(\mathscr{B})$ iff $(p, w) \in \mathscr{W}$.

Computing Éloise's winning region is a direct extension of the saturation algorithm for $Pre^*_P(C)$ in the non-game setting. We assume C is a regular set of configurations represented by an alternating automaton $\mathscr{A} = (\mathbb{S}, \Gamma, \delta, \mathscr{F})$ such that $Q \subseteq \mathbb{S}$ and there are no-incoming transitions to any state in Q.

The saturation algorithm constructs the automaton $\mathscr{B}$ that is the least fixed point of the sequence of automata $\mathscr{A}_0, \mathscr{A}_1, \ldots$ where $\mathscr{A}_0 = \mathscr{A} = (\mathbb{S}, \Gamma, \delta_0, \mathscr{F})$ and $\mathscr{A}_{i+1} = (\mathbb{S}, \Gamma, \delta_{i+1}, \mathscr{F})$ where δ_{i+1} is the smallest set of transitions such that

1. $\delta_i \subseteq \delta_{i+1}$, and
2. for each $q \in Q_E$, if $(q, A) \to (p, w) \in \Delta$ and $p \xrightarrow{w} S$ is a run of $\mathscr{A}_i$, then
$$q \xrightarrow{a} S \in \delta_{i+1}$$
and
3. for each $q \in Q_A$ and $A \in \Gamma$ and $S \subseteq \mathbb{S}$ such that for all
$$(q, A) \to (p, w) \in \Delta$$
there exists a run $p \xrightarrow{w} S'$ of $\mathscr{A}_i$ with $S' \subseteq S$, we have
$$q \xrightarrow{a} S \in \delta_{i+1} .$$

One can prove that $(p, w) \in \mathscr{W}$ iff $w \in \mathscr{L}_p(\mathscr{B})$. Thus we obtain regularity of the winning region. Since the maximum number of transitions of an alternating automaton is exponential in the number of states (and we do not add any new states), we have that $\mathscr{B}$ is constructible in exponential time.

Theorem 3 *The winning region of a pushdown reachability game is regular and constructible in exponential time.*

4.2.3 Winning strategies

Cachat has given two realisations of Éloise's winning strategy in a pushdown reachability game from a configuration in her winning region [13] . The first is a positional strategy that requires space linear in the size of the stack to compute. That is, he gives an algorithm that reads the stack and prescribes the next move that Éloise should make in order to win the game. The algorithm assigns costs to accepting runs of $\mathscr{B}$ for configurations in $\mathscr{W}$ by summing costs assigned to individual transitions.

Alternatively, Cachat presents a strategy that can be implemented by a pushdown automaton that tracks the moves of Abelard and recommends moves to Éloise. Since the automaton tracks the game, the strategy is not positional. However, the prescription of the next move requires only constant time.

In his PhD. thesis [14], Cachat also argues that similar strategies can be computed for Abelard for positions in his winning region.

4.3 Pushdown Büchi games

Plays of a game are infinite sequences. The reachability condition only depends on finite prefixes of these plays, hence games are won within a finite number of moves. This prevents the specification of liveness properties such as "every request is followed by an acknowledgment". Since it is not possible to know when to "stop waiting" for an acknowledgment to arrive, it is not possible to specify such conditions as simple reachability properties.

Büchi conditions allow liveness properties to be defined since deciding the winner of a particular play can take the whole infinite sequence into account. We define a pushdown Büchi game as a tuple $(Q, \Gamma, \perp, \Delta, F)$ – where $F \subseteq Q$ is a set of target control states – which defines a pushdown game $(Q, \Gamma, \perp, \Delta, W)$ with

$$W = \left\{ (p_0, w_0), (p_1, w_1), \ldots \;\middle|\; \forall i. \exists j \geq i. p_j \in F \right\} \ .$$

That is, Éloise wins the play if there is some control state in F that is visited infinitely often.

Cachat generalised the saturation method to construct the winning region of a pushdown Büchi game [13] by introducing the nesting of fixed point computations and projection described below.

To characterise the winning region of a pushdown Büchi game, a single least fixed point computation no longer suffices. Intuitively this is because satisfying the Büchi condition amounts to repeatedly satisfying a reachability condition; that is, repeatedly reaching a control state in F. We will begin by giving the characterisation, and then decoding it in the following paragraphs. By abuse of notation, we will write F to also denote the set of configurations $\{(p, w) \mid p \in F\}$ and $\overline{F}$ to denote its complement. The winning region of Éloise can be defined as

$$\nu Z_0 . \mu Z_1 . \left(F \cap Pre_P(Z_0)\right) \cup \left(\overline{F} \cap Pre_P(Z_1)\right) \ .$$

There are two pre-steps in the formula: $Pre_P(Z_0)$ and $Pre_P(Z_1)$. When a configuration is in F then we require that Éloise can force the next step of play to stay within Z_0. When the configuration is not in F we require that Éloise can force play to stay within Z_1.

To understand the role of the different fixed points, imagine a game where there is only one move from some configuration (p, w)

$$(p, w) \rightarrow (p, w) \ .$$

In the case where $p \in F$ it will be the case that (p, w) appears in the greatest fixed point Z_0. This is because greatest fixed points can be "self-supporting": if we include (p, w) in an approximation of Z_0, then it will appear in the next approximation of Z_0 by virtue of the fact that it was in the old valuation.

In the other case, when $p \notin F$, we would require (p,w) to appear in the least fixed point Z_1. However, since the least fixed point is the smallest possible fixed point, its members cannot be self-supporting. That is, if we took (p,w) out of our approximation, the next approximation would not include (p,w): there is nothing external compelling (p,w) to be in the least fixed point. This is why a reachability property is a least fixed point: it must contain only the configurations that eventually reach a target configuration – it cannot put off satisfying this obligation for an infinite number of steps.

In terms of Büchi games this difference makes sense: a play that repeatedly visits only the configuration (p,w) is only winning if $p \in F$. If $p \notin F$ then a configuration can only be winning if it eventually (after a finite number of steps) moves to a configuration that has a control state in F. Thus, the least fixed point represents configurations that must eventually reach a "good" configuration, while the greatest fixed point represents good configurations that are able to support themselves.

4.3.1 Computing the winning region

Automaton representation of multiple fixed points The saturation method for reachability properties computed a single fixed point with a single fixed point variable. We can think of the successive automata $\mathscr{A}_0, \mathscr{A}_1, \ldots$ as successive approximations of the value of Z. The final automaton computed gives the value of Z that is the solution to

$$\mu Z.C \cup Pre_P(Z)\ .$$

In the case of Büchi games, there are two nested fixed point computations over the variables Z_0 and Z_1. The winning region is the greatest fixed point for Z_0. However, in order to compute this fixed point we also have to compute the least fixed point for Z_1. Hence, we will need an automaton that can represent two different sets of configurations: the approximation of Z_0 as well as the approximation of Z_1. Thus, instead of having a state p of the alternating automaton for each control state p, we will have two states p^0 and p^1. A configuration (p,w) appears in the current approximation of Z_0 if it is accepted from p^0, and it appears in the current approximation of Z_1 if it is accepted from p^1. We will also use control states of the form p^2 to hold intermediate values of the computation.

Finally, the automata we build will have two additional states (these will be the only states that are not of the form p^α for some α). There will be one state $s_\perp$ that will be the only accepting state. Since all stacks finish with the bottom-of-stack symbol $\perp$, this state will have no outgoing transitions, and all incoming transitions will be of the form $s \xrightarrow{\perp} \{s_\perp\}$. No other transitions in the automaton will be labelled $\perp$.

The other additional state is s^* from which all stacks are accepted. This state has the outgoing transitions $s^* \xrightarrow{A} \{s^*\}$ for all $A \in \Gamma$ with $A \neq \perp$, and $s^* \xrightarrow{\perp} \{s_\perp\}$.

Evaluation strategy The saturation method computes fixed points following Knaster-Tarski theorem. That is, to compute a least fixed point, it begins with the smallest potential value (the set of target configurations C in the case of reachability properties, and the empty set in the case of Büchi properties). It then adds configurations to this set (by adding new transitions) that also necessarily appear in the least fixed point. This process is repeated until nothing more needs to be added – at which point the least fixed point has been calculated.

To compute a greatest fixed point Z_0 we follow the dual strategy. We begin with the largest possible value, which is the set of all configurations, which we will represent by states p^0 with all possible outgoing transitions. Next, the least fixed point Z_1 is calculated given the initial approximation of Z_0. Once the value of Z_1 is known, it becomes our new approximation of Z_0. Notice that this approximation is necessarily smaller than the initial attempt (both in terms of configurations accepted and transitions present). We then recalculate the least fixed point for Z_1 with the new smaller value of Z_0. In this way, starting from the largest possible value for Z_0 we successively shrink its value until a fixed point is found. This fixed point will be the greatest fixed point.

Projection When computing the greatest fixed point for Z_0 we repeatedly compute a least fixed point for Z_1. Each fixed point for Z_1 becomes the new approximation of Z_0. Hence, during our algorithm we need a method of assigning the value of Z_1 to Z_0. We call this manipulation of transitions *projection*.

Suppose the only outgoing transition from p^1 is

$$p^1 \xrightarrow{A} \{q^1, p^0\}$$

and we want to assign the new value of p^0. To do this we simply remove all transitions from p^0 (the old value) and introduce the transition

$$p^0 \xrightarrow{A} \{q^0, p^0\} \ .$$

There are several things to notice about this new transition. The first is that it emanates from p^0 rather than p^1. Next, we have changed the target state q^1 to q^0. This is because we are renaming all the states annotated with 1 to be annotated with 0. Finally, notice that we have not changed the target state p^0.

By leaving p^0 we are no longer simply transferring the value of Z_1 to Z_0 since we are changing the outgoing transitions from p^0. It is provable that this change in value is benign with respect to the fixed point of Z_0: since p^0 should accept all configurations (p,w) in the fixed point for Z_0, the fact that any run that reaches p^0 may accept additional configurations coming from the new value of p^0 rather than the old simply means that we are accelerating the computation of the fixed point.

For example, suppose we had a pushdown Büchi game with $p \in F \cap Q_E$ and an automaton with the transitions

$$p^1 \xrightarrow{A} \{p^0\} \text{ and } p^1 \xrightarrow{\perp} \{s_\perp\} \text{ and } p^0 \xrightarrow{\perp} \{s_\perp\}$$

and the pushdown game contains (amongst others) the rule $(p,A) \to (p,\varepsilon)$. In particular we accept the configuration $(p,A\perp)$ from p^1, and we do so because we can pop the A to reach $(p,\perp)$ (from which we suppose Éloise can win the game). After projection, we will have the transitions

$$p^0 \xrightarrow{A} \{p^0\} \text{ and } p^0 \xrightarrow{\perp} \{s_\perp\} \ .$$

Notice we now have a loop from p^0 enabling any configuration of the form $(p,A^*\perp)$ to be accepted from p^0. Thus we have increased the valuation during projection. However, this is benign because, by repeated applications of $(p,A) \to (p,\varepsilon)$ Éloise can reach $(p,\perp)$ and win the game. Thus, the projection has collapsed an unbounded sequence of moves into a single transition.

To calculate the fixed point for Z_1 we begin with the empty set as an initial approximation. Then we compute the new approximation for Z_1. While computing this approximation we will use states of the form p^2 to store the new value. Thus, to assign the new approximation to Z_1 we simply perform projection from the states p^2 to p^1 in the same way that we projected when assigning Z_1 to Z_0.

We thus define a projection function on states

$$\pi_{\alpha,\beta}(s) = \begin{cases} s & s = s^* \vee s = s_\perp \\ s & s = p^\gamma \wedge \gamma \neq \alpha \\ p^\beta & s = p^\alpha \end{cases}$$

which generalises naturally to a function on sets of states $\pi_{\alpha,\beta}(S) = \left\{\pi_{\alpha,\beta}(s) \mid s \in S\right\}$.

Algorithm Fix a pushdown Büchi game $P = (Q, \Gamma, \perp, \Delta, F)$. We begin our presentation of the algorithm by presenting a simple function for performing the projections described above. The function Proj($\mathscr{A}$, α, β) projects the value of the states p^α to p^β and deletes all the states p^α.

```
function Proj($\mathscr{A}$, $\alpha$, $\beta$)
    $(\mathbb{S}, \Gamma, \delta, \mathscr{F}) \leftarrow \mathscr{A}$
    $\mathbb{S}' \leftarrow \mathbb{S} \setminus \{p^\alpha \mid p \in Q\}$
    $\delta' \leftarrow \begin{array}{l} \left\{ s \xrightarrow{A} S \in \delta \mid \forall p \in Q. s \neq p^\alpha \wedge s \neq p^\beta \right\} \cup \\ \left\{ p^\beta \xrightarrow{A} \pi_{\alpha,\beta}(S) \;\middle|\; p^\alpha \xrightarrow{A} S \in \delta \right\} \end{array}$
     return $(\mathbb{S}', \Gamma, \delta', \mathscr{F})$
end function
```

The main algorithm contains two nested fixed point computations: the outer for Z_0 and the inner for Z_1. The initial automaton $\mathscr{A}^0$ contains only the states s^* and $s_\perp$ with transitions as described above. That is $\mathscr{A}^0 = (\{s^*, s_\perp\}, \Gamma, \delta, \{s_\perp\})$ with

$$\delta = \left\{ s^* \xrightarrow{A} \{s^*\} \mid A \in \Gamma \wedge A \neq \perp \right\} \cup \left\{ s^* \xrightarrow{\perp} \{s_\perp\} \right\} .$$

The algorithm is then a call to the function Fix$_0$($\mathscr{A}^0$) defined below. We define two functions for computing the fixed points for Z_0 and Z_1. Both of these functions are similar to each other: they begin by setting up an automaton representing the initial approximation of the fixed point, either by adding no transitions (the empty set) or all transitions (the largest set). They then enter a loop of computing the next approximation and then using projection to transfer (and accelerate) the new value to the states p^0 or p^1 as appropriate. The function Fix$_0$($\mathscr{A}$) computes the fixed point for Z_0 and uses Fix$_1$($\mathscr{A}$) to compute the next approximation, while Fix$_1$($\mathscr{A}$) computes the fixed point for Z_1 and uses a function Pre($\mathscr{A}$) to compute the next approximation. These two functions are thus defined

```
function Fix$_0$($\mathscr{A}$)
    $(\mathbb{S}, \Gamma, \delta, \mathscr{F}) \leftarrow \mathscr{A}$
    $\mathbb{S}' \leftarrow \mathbb{S} \cup \{p^0 \mid p \in Q\}$
    $\delta' \leftarrow \left\{ p^0 \xrightarrow{A} S \mid p \in Q \wedge A \in \Gamma \wedge A \neq \perp \wedge S \subseteq \mathbb{S}' \setminus \{s_\perp\} \right\} \cup \left\{ p^0 \xrightarrow{\perp} \{s_\perp\} \mid p \in Q \right\}$
    $\mathscr{B} \leftarrow (\mathbb{S}', \Gamma, \delta', \mathscr{F})$
    repeat
        $\mathscr{B} \leftarrow$ Fix$_1$($\mathscr{B}$)
        $\mathscr{B} \leftarrow$ Proj($\mathscr{B}$, 1, 0)
    until $\mathscr{B}$ unchanged
     return $\mathscr{B}$
end function
```

and

```
function Fix$_1$($\mathscr{A}$)
    $(\mathbb{S}, \Gamma, \delta, \mathscr{F}) \leftarrow \mathscr{A}$
    $\mathbb{S}' \leftarrow \mathbb{S} \cup \{p^1 \mid p \in Q\}$
    $\mathscr{B} \leftarrow (\mathbb{S}', \Gamma, \delta, \mathscr{F})$
    repeat
        $\mathscr{B} \leftarrow$ Pre($\mathscr{B}$)
        $\mathscr{B} \leftarrow$ Proj($\mathscr{B}'$, 2, 1)
    until $\mathscr{B}$ unchanged
```

```
        return $\mathscr{B}$
    end function .
```

The inner fixed point computation uses a function $\textsc{Pre}(\mathscr{A})$ to compute the step of the calculation corresponding to

$$(F \cap Pre_P(Z_0)) \cup \left(\overline{F} \cap Pre_P(Z_1)\right) \ .$$

This function adds transitions in the same way as the loop of saturation algorithm for reachability games, except it is sensitive to the two different fixed point variables. For convenience, we define the function Ω such that

$$\Omega(p) = \begin{cases} 0 & p \in F \\ 1 & p \notin F \ . \end{cases}$$

We can then define

```
    function $\textsc{Pre}(\mathscr{A})$
        $(\mathbb{S}, \Gamma, \delta, \mathscr{F}) \leftarrow \mathscr{A}$
        $\mathbb{S}' \leftarrow \mathbb{S} \cup \left\{ p^2 \mid p \in Q \right\}$
        $\delta' \leftarrow \begin{array}{l} \left\{ p^2 \xrightarrow{A} S \;\middle|\; p \in Q_E \wedge \exists (p,a) \to (q,w) \in \Delta . q^{\Omega(p)} \xrightarrow{w} S \right\} \cup \\ \left\{ p^2 \xrightarrow{A} S \;\middle|\; p \in Q_A \wedge \forall (p,a) \to (q,w) \in \Delta . \exists q^{\Omega(p)} \xrightarrow{w} S' . S' \subseteq S \right\} \end{array}$
        return $(\mathbb{S}', \Gamma, \delta', \mathscr{F})$
    end function .
```

The automaton $\mathscr{B}$ that is the result of $\textsc{Fix}_0(\mathscr{A}^0)$ will be such that $(p, w) \in \mathscr{W}$ iff $w \in \mathscr{L}_{p^0}(\mathscr{B})$. Since there are at most an exponential number of transitions in the automaton each fixed point may iterate at most an exponential number of times. This gives us an overall exponential run time for the algorithm.

Theorem 4 *The winning region of a pushdown Büchi game is regular and computable in exponential time.*

Note that for the one player case (*i.e.* all states belong to Éloise), the computation can be done in polynomial time [7, 22].

4.3.2 Winning strategies

Cachat also showed that, like in reachability games, it is possible to construct a linear space positional strategy and a constant time (though not positional) pushdown strategy for Éloise. However, in his PhD. thesis [14] Cachat observes that adopting his techniques for computing strategies for Abelard is not clear. However, it is known that, even for the full case of parity games, a pushdown strategy exists using different techniques [57, 40].

4.4 Pushdown parity games

Parity games allow more complex liveness properties to be checked. To define a parity game, each configuration is assigned a "colour" from a set of colours represented by natural numbers. The winner of the game depends on the smallest colour appearing infinitely often in the run: if it is even then Éloise wins the game, else Abelard wins.

More formally, given a sequence of configurations $\rho = (q_0, w_0), (q_1, w_1), \ldots$ let $\text{Inf}(\rho)$ be the set of control states appearing infinitely often in ρ. That is

$$\text{Inf}(\rho) = \left\{ q \;\middle|\; \forall i \exists j > i . q_j = q \right\} \ .$$

Given a set of control states Q and maximum colour κ, let $\Omega : Q \to \{0, \ldots, \kappa\}$ be a colouring function assigning colours to each control state. We can generalise Ω to sets of control states P by taking the image of P. That is, $\Omega(P) = \{\alpha \mid \exists p \in P.\Omega(p) = \alpha\}$.

A pushdown parity game is a tuple $(Q, \Gamma, \perp, \Delta, \Omega)$ where $\Omega : Q \to \{0, \ldots, \kappa\}$ is a colouring function assigning to each control state a colour from the set $\{0, \ldots, \kappa\}$. Moreover, the tuple defines a pushdown game $(Q, \Gamma, \perp, \Delta, W)$ where

$$W = \{\rho \mid \min(\Omega(\text{Inf}(\rho))) \text{ is even}\} \ .$$

Thus, a Büchi game is a special case of a parity game, where the set of colours is $\{0, 1\}$ and

$$\Omega(p) = \begin{cases} 0 & p \in F \\ 1 & p \notin F \ . \end{cases}$$

4.4.1 Characterising the winning region

The characterisation of Éloise's winning region in terms of fixed points is a natural extension of the Büchi version. That is, assuming κ to be odd and writing C_α to denote $\{(p, w) \mid \Omega(p) = \alpha\}$, we need

$$\nu Z_0.\mu Z_1.\cdots.\nu Z_{\kappa-1}.\mu Z_\kappa. \bigcup_{0 \leq \alpha \leq \kappa} (C_\alpha \cap Pre_P(Z_\alpha)) \ .$$

This formula can be understood as a generalisation of the Büchi formula, where $F = C_0$ and $\overline{F} = C_1$. When the colour of a configuration is odd, then it is bound by a least fixed point. Hence, it must eventually exit this fixed point by visiting a configuration with a smaller colour (just like a configuration in $\overline{F}$ had to visit a configuration in F). When the colour is even, then it is bound by a greatest fixed point – hence a play can stay within this fixed point, never visiting a smaller colour, and satisfy the winning condition for Éloise.

4.4.2 Computing the winning region

Fix a pushdown parity game $P = (Q, \Gamma, \perp, \Delta, \Omega)$. Computing the winning region in a pushdown parity game is a direct extension of the algorithm presented for Büchi games. Since a Büchi game is simply a pushdown parity game with two colours, we generalise the nesting of the fixed point calls to an arbitrary number of colours. To this end we introduce a function DISPATCH($\mathscr{A}$, α) that manages the level of nesting, and performs a fixed point or a pre-step analysis as appropriate.

```
function DISPATCH($\mathscr{A}$, $\alpha$)
    if $\alpha = \kappa + 1$ then
        return PRE($\mathscr{A}$)
    else
        return FIX($\mathscr{A}$, $\alpha$)
    end if
end function
```

Using this function we can define a generic fixed point function based on the Büchi functions. This function performs the nested calculations and the projection as before. The initial transitions from the new states introduced by the function depend on the parity of α: when computing an even (greatest) fixed point, we add all transitions, and when computing an odd (least) fixed point, we add no transitions.

```
function FIX($\mathscr{A}$, $\alpha$)
```

```
    $(\mathbb{S}, \Gamma, \delta, \mathscr{F}) \leftarrow \mathscr{A}$
    $\mathbb{S}' \leftarrow \{p^\alpha \mid p \in Q\}$
    **if** $\alpha$ is even **then**
        $\delta' \leftarrow \left\{p^\alpha \xrightarrow{A} S \mid p \in Q \wedge A \in \Gamma \wedge A \neq \perp \wedge S \subseteq \mathbb{S}' \setminus \{s_\perp\}\right\} \cup \left\{p^\alpha \xrightarrow{\perp} \{s_\perp\} \mid p \in Q\right\}$
    **else**
        $\delta' \leftarrow \emptyset$
    **end if**
    $\mathscr{B} \leftarrow (\mathbb{S} \cup \mathbb{S}', \Gamma, \delta \cup \delta', \mathscr{F})$
    **repeat**
        $\mathscr{B} \leftarrow$ DISPATCH($\mathscr{B}$, $\alpha + 1$)
        $\mathscr{B} \leftarrow$ PROJ($\mathscr{B}$, $\alpha + 1$, $\alpha$)
    **until** $\mathscr{B}$ unchanged
    **return** $\mathscr{B}$
**end function**
```

Finally, we redefine the PRE($\mathscr{A}$) function to add transitions to the correct initial states. Note, we were already using Ω to distinguish between different fixed point variables, hence this function is almost identical to the Büchi case.

```
**function** PRE($\mathscr{A}$)
    $(\mathbb{S}, \Gamma, \delta, \mathscr{F}) \leftarrow \mathscr{A}$
    $\mathbb{S}' \leftarrow \mathbb{S} \cup \left\{p^{\kappa+1} \mid p \in Q\right\}$
    $\delta' \leftarrow \begin{array}{l|l} \left\{p^{\kappa+1} \xrightarrow{A} S \right. & \left. p \in Q_E \wedge \exists (p,a) \to (q,w) \in \Delta . q^{\Omega(p)} \xrightarrow{w} S\right\} \cup \\ \left\{p^{\kappa+1} \xrightarrow{A} S \right. & \left. p \in Q_A \wedge \forall (p,a) \to (q,w) \in \Delta . \exists q^{\Omega(p)} \xrightarrow{w} S' . S' \subseteq S\right\} \end{array}$
    **return** $(\mathbb{S}', \Gamma, \delta', \mathscr{F})$
**end function**
```

Thus, to compute the winning region of a pushdown parity game, we make the call DISPATCH($\mathscr{A}^0$, 0) where $\mathscr{A}^0$ is the initial automaton with only the states s^* and $s_\perp$ as defined in the Büchi case.

The automaton $\mathscr{B}$ that is the result of DISPATCH($\mathscr{A}^0$, 0) will be such that $(p, w) \in \mathscr{W}$ iff $w \in \mathscr{L}_{p^0}(\mathscr{B})$. Since there are at most an exponential number of transitions in the automaton each fixed point may iterate at most an exponential number of times. This gives us an overall exponential run time for the algorithm.

Theorem 5 *The winning region of a pushdown parity game is regular and computable in exponential time.*

4.4.3 Winning strategies

Unfortunately, it is currently unknown how to compute the winning strategies for Éloise and Abelard using the saturation technique for pushdown parity games. However, using a different approach, both Walukiewicz [57] and Serre [40] have shown that a pushdown strategy exists for both players.

5 Implementations and Applications of Saturation Methods

In this article, we have presented the saturation method from a theoretical standpoint. The method, however, is an algorithmic approach that is well suited to implementation, and several tools have been constructed using saturation as its core technique.

5.1 Single Player Implementations

Perhaps the most famous of these tools is Moped [21, 39] and its incarnation as a model checker for Java, JMoped [52, 50]. In taking the algorithm from a theoretical tool to a practical one, a number of new concerns had to be taken into account.

The rules of a pushdown system roughly correspond to the statements in a program. In a program with thousands of lines, a fixed point iteration that checks, during each iteration, whether each rule leads to new transitions in the automaton would be woefully inefficient. In constructing Moped, Esparza *et al.* [20] showed how this naive outer loop can be reorganised such that, at each iteration, only the relevant rules of the system were considered, leading to a significant improvement in performance.

A second consideration of applications to the analysis of program models is the handling of data values. Boolean programs are essentially pushdown systems where each control state and stack character contains a valuation of a set of global and local boolean variables respectively. These boolean programs are the natural output of predicate abstraction tools such as SATABS [18] as well as the target compilation language of JMoped.

Since there are only finitely many valuations of sets of boolean variables, they can directly be encoded as control states or characters and standard pushdown analysis techniques can be employed. However, since they are also exponential in number, such an approach is inherently inefficient. Hence, Esparza *et al.* introduced *symbolic pushdown systems* [21] which make boolean valuations first class objects. The saturation technique was extended by adding BDDs representing variable valuations to the edges of the P-automata, leading to an implementation capable of analysing symbolic pushdown systems derived from real-world programs.

Around this time it was observed by Reps that the BDDs could be replaced by any abstract domain of values that was sufficiently well behaved, and many static analyses could be derived. This led to the introduction of *weighted pushdown systems* [37] (and, indeed, *extended* weighted pushdown systems amongst other improvements [33, 32]), of which symbolic pushdown systems and their BDD representation were an instance. The developers of Moped created the *weighted pushdown system library* [58] as a component of Moped, and Reps *et al.* developed WALi [56] implementing these new algorithms.

5.2 Two-Player Implementations

Perhaps the most straight-forward optimisation to make to the saturation technique as presented for two-player games is via the observation that a transition

$$s \xrightarrow{A} S$$

is effectively redundant if there exists another transition

$$s \xrightarrow{A} S'$$

with $S' \subseteq S$. This is because an accepting run from S contains within it an accepting run from S', and thus the former transition can be removed.

When considering reachability games, it is also possible to improve the naive fixed point iteration, as in the single-player case, to avoid checking against all pushdown rules during each step of the implementation. Such an optimisation was introduced by Suwimonteerabuth *et al.* and implemented with applications to certificate chain analysis [53].

This work has recently been built upon by Song who has developed various tools based upon reductions to Büchi games and tools for their analysis. Primarily this work has focussed on a specification language that is an extension of CTL and its translation into symbolic pushdown Büchi games [44, 46] resulting in the tool PuMoC [45]. The main application of this work has been in the detection of malware. More recently still, this work has been developed for LTL-like properties to deal with situations where the CTL approach was insufficient [47], culminating in the PoMMaDe tool [49].

However, the combination of BDD representations and alternating automata is not an easy one, since BDDs lack the necessary alternation for a direct embedding. Hence, Song's algorithm pays an extra exponential in its worst-case complexity (doubly exponential rather than exponential), although the practical runtime is improved. The optimal inclusion of symbolic representations into the analysis of two-player games remains an open problem.

The saturation technique for the full case of parity games has been implemented in the PDSolver tool [27] and applied to dataflow analysis problems for Java programs. Due to the interactions between the several layers of fixed points, it is not clear how to adapt Esparza *et al.* and Suwimonteerabuth *et al.*'s efficient algorithms to this case, nor how to include symbolic representations. These remain limitations of the tool, and interesting avenues for future work.

6 Extensions of the Saturation Method

In this article we have looked at the different saturation methods for pushdown systems. Across several articles, the technique has proved to be applicable to various extensions to the basic model. We briefly list some of these results here.

Concurrency The reachability problem for pushdown systems with two or more stacks is well known to be undecidable. Since multiple stacks are needed to model multi-thread recursive programs, a number of underapproximation techniques have been studied for which the reachability problem is decidable. One such technique is *bounded context switching* [36] where the number of interactions between the threads is limited to an *a priori* fixed number *k*. While this cannot prove the absence of errors, it is effective at finding bugs in programs, since, empirically, bugs usually manifest themselves within a small number of interactions. This restriction can be relaxed further by allowing a bounded number of *phases* [54] (where all threads run concurrently, but during each phase only one thread is allowed to pop from its stack), or a bounded scope [55] (where, threads are scheduled in a round-robin fashion, and characters may only be removed from the stack if they were pushed at most a fixed number of rounds earlier).

The saturation technique has proved useful for each of these restrictions. In particular, Moped has been extended to provide context bounded analysis of multi-stack pushdown systems [51] by Suwimonteerabuth *et al.* and saturation was used by Seth to provide a regular solution to the global reachability problem for phase bounded pushdown systems [43]. The original proof that the reachability problem for scope bounded pushdown systems is decidable was itself an extension of the saturation technique [55].

An alternative restriction that permits a decidable reachability and LTL model checking problem is that of *ordered multi-pushdown systems* where only the leftmost non-empty stack is able to remove characters. Atig provides two extensions of the saturation technique in this direction [1]. First, instead of each pushdown rule adding a fixed sequence of characters to the stack, he allows rules to contain languages of sequences that may be pushed. If it is decidable whether the language of a rule intersected with a regular language is empty, then an augmented saturation technique leads to an effective analysis algorithm. In particular, the model checking problem for ordered pushdown systems can be solved with this formalism.

Finally, Song generalises his LTL model checking algorithms to the case of pushdown systems with dynamic thread creation [48], again using a saturation technique at its core.

Ground Tree Rewrite Systems and Resources Ground tree rewrite systems can be thought of as pushdown systems with a single control state and a more complex stack structure. That is, the stack is a tree rather than a word. Rewrite rules in this system replace complete subtrees. For example a push rule $(p,A) \rightarrow (p,BC)$ can be considered to be replacing the subtree consisting in the leaf node A with the subtree $B(C)$ (i.e. a B-node with a C-leaf as a child). In 1987, Dauchet *et al.* used saturation to show that the confluence problem for these systems is decidable [19]. More recently, Lang and Löding adapted this method to analyse prefix replacement systems with resource usage [34].

Higher-Order and Collapsible Pushdown Systems Pushdown systems provide a natural model for first-order recursive programs. When considering higher-order programs, we can use *higher-order pushdown systems* [35] whose stacks have a nested "stack-of-stacks" structure. These systems correspond to *higher-order recursion schemes* satisfying a *safety* constraint [30]. Recently, these systems were generalised to *collapsible pushdown systems* (via *panic automata* [31]), providing an automata model without the need for the safety constraint [25].

The saturation technique was first applied to the analysis of higher-order systems by Bouajjani and Meyer [8] who considered higher-order pushdown systems with a single control state. This algorithm was generalised by Hague and Ong to permit an arbitrary number of control states [26]. An alternative construction in the case of second order higher-order pushdown systems was provided by Seth [41].

More recently this approach was developed by Broadbent *et al.* to obtain a saturation algorithm for the full case of collapsible pushdown systems [9], leading to the analysis tool C-SHORe [10]. This algorithm was applied directly to the analysis of recursion schemes (without the intermediate automata model) by Broadbent and Kobayashi, resulting in the HorSat tool [11].

Finally, the case of concurrent higher-order systems has been briefly considered. Seth used saturation to show that parity games over phase-bounded higher-order pushdown systems (without collapse) are effectively solvable [42]. Recently, Hague showed that the saturation approaches for first-order phase-bounded, ordered and scope-bounded pushdown systems can be adapted to solve the analogous reachability problems for collapsible pushdown systems [24].

References

[1] M. F. Atig (2012): *Model-Checking of Ordered Multi-Pushdown Automata*. *Logical Methods in Computer Science* 8(3), doi:10.2168/LMCS-8(3:20)2012.

[2] T. Ball, V. Levin & S. K. Rajamani (2011): *A decade of software model checking with SLAM*. *Commun. ACM* 54(7), pp. 68–76, doi:10.1145/1965724.1965743.

[3] T. Ball & S. K. Rajamani (2000): *Bebop: A Symbolic Model Checker for Boolean Programs*. In: *Proceedings of SPIN'00*, pp. 113–130, doi:10.1007/10722468_7.

[4] Y. Bar-Hillel, M. Perles & E. Shamir (1961): *On formal properties of simple phrase structure grammars*. *Z. Phonetik Sprachwiss. Kommunikat.* 14, p. 143172.

[5] M. Benois (1969): *Parties rationnelles du groupe libre*. *Comptes-Rendus de l'Acamdémie des Sciences de Paris, Série A* 269, pp. 1188–1190.

[6] M. Benois & J. Sakarovitch (1986): *On the Complexity of Some Extended Word Problems Defined by Cancellation Rules*. *Inf. Process. Lett.* 6, pp. 281–287, doi:10.1016/0020-0190(86)90087-6.

[7] A. Bouajjani, J. Esparza & O. Maler (1997): *Reachability Analysis of Pushdown Automata: Application to Model-Checking*. In: *Proceedings of CONCUR'97*, pp. 135–150, doi:`10.1007/3-540-63141-0_10`.

[8] A. Bouajjani & A. Meyer (2004): *Symbolic Reachability Analysis of Higher-Order Context-Free Processes*. In: *Proceedings of FSTTCS'04*, pp. 135–147, doi:`10.1007/978-3-540-30538-5_12`.

[9] C. H. Broadbent, A. Carayol, M. Hague & O. Serre (2012): *A Saturation Method for Collapsible Pushdown Systems*. In: *Proceedings of ICALP'12*, pp. 165–176, doi:`10.1007/978-3-642-31585-5_18`.

[10] C. H. Broadbent, A. Carayol, M. Hague & O. Serre (2013): *C-SHORe: a collapsible approach to higher-order verification*. In: *Proceedings of ICFP'13*, pp. 13–24, doi:`10.1145/2500365.2500589`.

[11] C. H. Broadbent & N. Kobayashi (2013): *Saturation-Based Model Checking of Higher-Order Recursion Schemes*. In: *Proceedings of CSL'13*, pp. 129–148, doi:`10.4230/LIPIcs.CSL.2013.129`.

[12] R. J Büchi (1964): *Regular canonical systems*. *Archive for Mathematical Logic* 6(3), pp. 91–111, doi:`10.1007/BF01969548`.

[13] T. Cachat (2002): *Symbolic Strategy Synthesis for Games on Pushdown Graphs*. In: *Proceedings of ICALP'02*, pp. 704–715, doi:`10.1007/3-540-45465-9_60`.

[14] T. Cachat (2003): *Games on Pushdown Graphs and Extensions*. Ph.D. thesis, RWTH Aachen. Available at `http://www.liafa.jussieu.fr/~txc/Download/Cachat-PhD.pdf`.

[15] D. Caucal (1988): *Récritures suffixes de mots*. Research Report RR-0871, INRIA.

[16] D. Caucal (1990): *On the Regular Structure of Prefix Rewriting*. In: *Proceedings of CAAP'90, Lecture Notes in Computer Science* 431, Springer, pp. 87–102, doi:`10.1007/3-540-52590-4_42`.

[17] D. Caucal (2008): *Deterministic graph grammars*. In Jörg Flum, Erich Grädel & Thomas Wilke, editors: *Logic and Automata: History and Perspectives in Honor of Wolfgang Thomas, Texts in Logic and Games* 2, Amsterdam University Press, pp. 169–250.

[18] E. M. Clarke, D. Kroening, N. Sharygina & K. Yorav (2005): *SATABS: SAT-Based Predicate Abstraction for ANSI-C*. In: *Proceedings of TACAS'05*, pp. 570–574.

[19] M. Dauchet, S. Tison, T. Heuillard & P. Lescanne (1987): *Decidability of the Confluence of Ground Term Rewriting Systems*. In: *Proceedings of LICS'87*, pp. 353–359.

[20] J. Esparza, D. Hansel, P. Rossmanith & S. Schwoon (2000): *Efficient Algorithms for Model Checking Pushdown Systems*. In: *Proceedings of CAV'00*, pp. 232–247, doi:`10.1007/10722167_20`.

[21] J. Esparza & S. Schwoon (2001): *A BDD-Based Model Checker for Recursive Programs*. In: *Proceedings of CAV'01*, pp. 324–336, doi:`10.1007/3-540-44585-4_30`.

[22] A. Finkel, B. Willems & P. Wolper (1997): *A direct symbolic approach to model checking pushdown systems*. *Electr. Notes Theor. Comput. Sci.* 9, pp. 27–37, doi:`10.1007/3-540-45465-9_60`.

[23] S. A. Greibach (1967): *A note on pushdown store automata and regular systems*. *Proceedings of the American Mathematical Society*, pp. 263–268, doi:`10.1090/S0002-9939-1967-0209086-1`.

[24] M. Hague (2013): *Saturation of Concurrent Collapsible Pushdown Systems*. In: *Proceedings of FSTTCS'13*, pp. 313–325, doi:`10.4230/LIPIcs.FSTTCS.2013.313`.

[25] M. Hague, A. S. Murawski, C.-H. Luke Ong & O. Serre (2008): *Collapsible Pushdown Automata and Recursion Schemes*. In: *Proceedings of LICS'08*, pp. 452–461, doi:`10.1109/LICS.2008.34`.

[26] M. Hague & C.-H. L. Ong (2008): *Symbolic Backwards-Reachability Analysis for Higher-Order Pushdown Systems*. *Logical Methods in Computer Science* 4(4), doi:`10.2168/LMCS-4(4:14)2008`.

[27] M. Hague & C.-H. L. Ong (2010): *Analysing Mu-Calculus Properties of Pushdown Systems*. In: *Proceedings of SPIN'10*, pp. 187–192, doi:`10.1007/978-3-642-16164-3_14`.

[28] M. Hague & C.-H. Luke Ong (2009): *Winning Regions of Pushdown Parity Games: A Saturation Method*. In: *Proceedings of CONCUR'09*, pp. 384–398, doi:`10.1007/978-3-642-04081-8_26`.

[29] N. D. Jones & S. S. Muchnick (1977): *Even Simple Programs Are Hard To Analyze*. *J. ACM* 24(2), pp. 338–350, doi:`10.1145/322003.322016`.

[30] T. Knapik, D. Niwinski & P. Urzyczyn (2002): *Higher-Order Pushdown Trees Are Easy*. In: *Proceedings of FoSSaCS'02*, pp. 205–222, doi:`10.1007/3-540-45931-6_15`.

[31] T. Knapik, D. Niwinski, P. Urzyczyn & I. Walukiewicz (2005): *Unsafe Grammars and Panic Automata*. In: *Proceedings of ICALP'05*, pp. 1450–1461, doi:`10.1007/11523468_117`.

[32] A. Lal & T. W. Reps (2006): *Improving Pushdown System Model Checking*. In: *Proceedings of CAV'06*, pp. 343–357, doi:`10.1007/11817963_32`.

[33] A. Lal, T. W. Reps & G. Balakrishnan (2005): *Extended Weighted Pushdown Systems*. In: *Proceedings of CAV'05*, pp. 434–448, doi:`10.1007/11513988_44`.

[34] M. Lang & C. Löding (2013): *Modeling and Verification of Infinite Systems with Resources*. *Logical Methods in Computer Science* 9(4), doi:`10.2168/LMCS-9(4:22)2013`.

[35] A. N. Maslov (1976): *Multilevel stack automata*. *Problems of Information Transmission* 15, pp. 1170–1174.

[36] S. Qadeer (2008): *The Case for Context-Bounded Verification of Concurrent Programs*. In: *Proceedings of the SPIN'08*, Springer-Verlag, Berlin, Heidelberg, pp. 3–6, doi:`10.1007/978-3-540-85114-1_2`.

[37] T. W. Reps, S. Schwoon, S. Jha & D. Melski (2005): *Weighted pushdown systems and their application to interprocedural dataflow analysis*. *Sci. Comput. Program.* 58(1-2), pp. 206–263, doi:`10.1016/j.scico.2005.02.009`.

[38] J. Sakarovitch (2009): *Elements of Automata Theory*. Cambridge University Press, doi:`10.1017/CBO9781139195218`.

[39] S. Schwoon (2002): *Model-checking Pushdown Systems*. Ph.D. thesis, Technical University of Munich.

[40] O. Serre (2004): *Contribution à létude des jeux sur des graphes de processus à pile*. Ph.D. thesis, Université Paris 7 – Denis Diderot, UFR dinformatique. Available at `http://tel.archives-ouvertes.fr/tel-00011326`.

[41] A. Seth (2008): *An Alternative Construction in Symbolic Reachability Analysis of Second Order Pushdown Systems*. *Int. J. Found. Comput. Sci.* 19(4), pp. 983–998, doi:`10.1142/S012905410800608X`.

[42] A. Seth (2009): *Games on Higher Order Multi-stack Pushdown Systems*. In: *Proceedings of RP'09*, pp. 203–216, doi:`10.1007/978-3-642-04420-5_19`.

[43] A. Seth (2010): *Global Reachability in Bounded Phase Multi-stack Pushdown Systems*. In: *Proceedings of CAV'10*, pp. 615–628, doi:`10.1007/978-3-642-14295-6_53`.

[44] F. Song & T. Touili (2011): *Efficient CTL Model-Checking for Pushdown Systems*. In: *Proceedings of CONCUR'11*, pp. 434–449, doi:`10.1007/978-3-642-23217-6_29`.

[45] F. Song & T. Touili (2012): *PuMoC: a CTL model-checker for sequential programs*. In: *Proceedings of ASE'12*, pp. 346–349, doi:`10.1145/2351676.2351743`.

[46] F. Song & T. Touili (2012): *Pushdown Model Checking for Malware Detection*. In: *Proceedings of TACAS'12*, pp. 110–125, doi:`10.1007/978-3-642-28756-5_9`.

[47] F. Song & T. Touili (2013): *LTL Model-Checking for Malware Detection*. In: *Proceedings of TACAS'13*, pp. 416–431, doi:`10.1007/978-3-642-36742-7_29`.

[48] F. Song & T. Touili (2013): *Model Checking Dynamic Pushdown Networks*. In: *Proceedings of APLAS'13*, pp. 33–49, doi:`10.1007/978-3-319-03542-0_3`.

[49] F. Song & T. Touili (2013): *PoMMaDe: pushdown model-checking for malware detection*. In: *Proceedings of ESEC/FSE'13*, pp. 607–610, doi:`10.1145/2491411.2494599`.

[50] D. Suwimonteerabuth, F. Berger, S. Schwoon & J. Esparza (2007): *jMoped: A Test Environment for Java Programs*. In: *Proceedings of CAV'07*, pp. 164–167, doi:`10.1007/978-3-540-73368-3_19`.

[51] D. Suwimonteerabuth, J. Esparza & S. Schwoon (2008): *Symbolic Context-Bounded Analysis of Multi-threaded Java Programs*. In: *Proceedings of SPIN'08*, pp. 270–287, doi:`10.1007/978-3-540-85114-1_19`.

[52] D. Suwimonteerabuth, S. Schwoon & J. Esparza (2005): *jMoped: A Java Bytecode Checker Based on Moped*. In: *Proceedings of TACAS'05*, pp. 541–545, doi:10.1007/978-3-540-31980-1_35.

[53] D. Suwimonteerabuth, S. Schwoon & J. Esparza (2006): *Efficient Algorithms for Alternating Pushdown Systems with an Application to the Computation of Certificate Chains*. In: *Proceedings of ATVA'06*, pp. 141–153, doi:10.1007/11901914_13.

[54] S. La Torre, P. Madhusudan & G. Parlato (2007): *A Robust Class of Context-Sensitive Languages*. In: *Proceedings of LICS'07*, pp. 161–170, doi:10.1109/LICS.2007.9.

[55] S. La Torre & M. Napoli (2011): *Reachability of Multistack Pushdown Systems with Scope-Bounded Matching Relations*. In: *Proceedings of CONCUR'11*, pp. 203–218, doi:10.1007/978-3-642-23217-6_14.

[56] WALi: Weighted Automata Library: https://research.cs.wisc.edu/wpis/wpds/download.php.

[57] I. Walukiewicz (2001): *Pushdown Processes: Games and Model-Checking*. *Inf. Comput.* 164(2), pp. 234–263, doi:10.1006/inco.2000.2894.

[58] WPDS Library: http://www2.informatik.uni-stuttgart.de/fmi/szs/tools/wpds/.

From Finite Automata to Regular Expressions and Back—A Summary on Descriptional Complexity

Hermann Gruber
knowledgepark AG, Leonrodstr. 68,
80636 München, Germany
hermann.gruber@knowledgepark-ag.de

Markus Holzer
Institut für Informatik, Universität Giessen,
Arndtstr. 2, 35392 Giessen, Germany
holzer@informatik.uni-giessen.de

The equivalence of finite automata and regular expressions dates back to the seminal paper of Kleene on events in nerve nets and finite automata from 1956. In the present paper we tour a fragment of the literature and summarize results on upper and lower bounds on the conversion of finite automata to regular expressions and *vice versa*. We also briefly recall the known bounds for the removal of spontaneous transitions (ε-transitions) on non-ε-free nondeterministic devices. Moreover, we report on recent results on the average case descriptional complexity bounds for the conversion of regular expressions to finite automata and brand new developments on the state elimination algorithm that converts finite automata to regular expressions.

1 Introduction

There is a vast literature documenting the importance of the notion of finite automata and regular expressions as an enormously valuable concept in theoretical computer science and applications. It is well known that these two formalisms are equivalent, and in almost all monographs on automata and formal languages one finds appropriate constructions for the conversion of finite automata to equivalent regular expressions and back. Regular expressions, introduced by Kleene [68], are well suited for human users and therefore are often used as interfaces to specify certain patterns or languages. For example, in the widely available programming environment UNIX, regular(-like) expressions can be found in legion of software tools like, e.g., `awk`, `ed`, `emacs`, `egrep`, `lex`, `sed`, `vi`, etc., to mention a few of them. On the other hand, automata [94] immediately translate to efficient data structures, and are very well suited for programming tasks. This naturally raises the interest in conversions among these two different notions. Our tour on the subject covers some (recent) results in the fields of descriptional and computational complexity. During the last decade descriptional aspects on finite automata and regular expressions formed an extremely vivid area of research. For recent surveys on descriptional complexity issues of finite automata and regular expressions we refer to, for example, [39, 56, 57, 58, 59, 60, 103]. This was not only triggered by appropriate conferences and workshops on that subject, but also by the availability of mathematical tools and the influence of empirical studies. For obvious reasons, this survey lacks completeness, as finite automata and regular expressions fall short of exhausting the large number of related problems considered in the literature. We give a view of what constitutes, in our opinion, the most interesting recent links to the problem area under consideration.

Before we start our tour some definitions are in order. First of all, our nomenclature of finite automata is as follows: a *nondeterministic finite automaton with ε-transitions* (ε-NFA) is a quintuple $A = (Q, \Sigma, \delta, q_0, F)$, where Q is the finite set of *states*, Σ 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 $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$ is the *transition function*. If a finite automaton has no ε-transitions, i.e., the transition function is restricted to $\delta : Q \times \Sigma \to 2^Q$, then we simply speak of a *nondeterministic finite automaton* (NFA). Moreover, a nondeterministic finite automaton is *deterministic* (DFA) if and only if $|\delta(q,a)| = 1$, for all states $q \in Q$ and letters $a \in \Sigma$. The *language accepted* by the finite automaton A is defined as $L(A) = \{\, w \in \Sigma^* \mid \delta(q_0, w) \cap F \neq \emptyset \,\}$, where the transition function is recursively extended to $\delta : Q \times \Sigma^* \to 2^Q$. Second, we turn to the definition of regular expressions: the *regular expressions* over an alphabet Σ and the languages they describe are defined inductively in the usual way:[1] $\emptyset$, ε, and every letter a with $a \in \Sigma$ is a regular expression, and when s and t are regular expressions, then $(s+t)$, $(s \cdot t)$, and $(s)^*$ are also regular expressions. The language defined by a regular expression r, denoted by $L(r)$, is defined as follows: $L(\emptyset) = \emptyset$, $L(\varepsilon) = \{\varepsilon\}$, $L(a) = \{a\}$, $L(s+t) = L(s) \cup L(t)$, $L(s \cdot t) = L(s) \cdot L(t)$, and $L(s^*) = L(s)^*$. For further details on finite automata and regular expressions we refer to, e.g., [61].

We start our tour on the subject with the question on the appropriate measure for finite automata and regular expressions. We discuss this topic in detail in Section 2. There we also concentrate on two specific measures: on star height for regular expressions and cycle rank for the automaton side. By Eggan's theorem [26] both measures are related to each other. Recent developments, in particular on the conversion from finite automata to regular expressions, utilize this connection to prove upper and lower bounds. Then in Section 3 we take a closer on the conversion from regular expressions to equivalent finite automata. We recall the most prominent conversion algorithms such as Thompson's construction and its optimized version the follow automaton, the position or Glushkov automaton, and conversion by computations of the (partial-)derivatives. We summarize the known relations on these devices, which were mostly found during the last decade. Significant differences on these constructions are pointed out and the presented developments on lower bound and upper bound results enlighten the efficiency of these algorithms. Some of the bounds are sensitive to the size of the alphabet. Besides worst case descriptional complexity results on the synthesis problem of finite automata from regular expressions, we also list some recent results on the average case complexity of the transformation of regular expressions to finite automata. Finally, in Section 4 we consider the converse transformation. Again, we summarize some of the few conversion techniques, but then stick in more detail to the so-called state elimination technique. The reason for that is, that in [97], it was shown that almost all conversion methods can be recast as variants of the state elimination technique. Here, the ordering in which the states are eliminated can largely affect the size of the regular expression corresponding to the given finite automaton. We survey some heuristics that have been proposed for this goal. For appropriate choices of the ordering, nontrivial upper bounds on regular expression size can be proved. By looking at the transition structure of the NFA, results from graph theory can help in obtaining shorter expressions. There we try to illustrate the key insights with the aid of examples, thereby avoiding the need for a deeper dive into graph theoretic concepts. We also explain the technique by which the recent lower bounds on regular expression size were obtained. In this part, the known upper and lower bounds match only in the sense that we can identify the rough order of magnitude. So we observe an interesting tension between algorithms with provable performance guarantees, other heuristics that are observed to behave better in experiments, and finally some lower bounds, which seize the expectations that we may have on practical algorithms.

2 Measures on Finite Automata and Regular Expressions

What can be said about the proper measure on finite automata and regular expressions? For finite automata there are two commonly accepted measures, namely the number of states and the number of

[1] For convenience, parentheses in regular expressions are sometimes omitted and the concatenation is simply written as juxtaposition. The priority of operators is specified in the usual fashion: concatenation is performed before union, and star before both product and union.

transitions. The measure sc (nsc, respectively) counts the number of states of a deterministic (nondeterministic, respectively) finite automaton and tc (ntc, respectively) does the same for the number of transitions for the appropriate devices. Moreover, nsc_ε (ntc_ε, respectively) gives the number of states (transitions, respectively) in an ε-NFA. The following relations between these measures are well known—see also [84, 86, 94].

Theorem 1 *Let $L \subseteq \Sigma^*$ be a regular language. Then*

1. $\mathsf{nsc}_\varepsilon(L) = \mathsf{nsc}(L) \leq \mathsf{sc}(L) \leq 2^{\mathsf{nsc}(L)}$ *and* $\mathsf{tc}(L) = |\Sigma| \cdot \mathsf{sc}(L)$ *and*
2. $\mathsf{nsc}(L) - 1 \leq \mathsf{ntc}_\varepsilon(L) \leq \mathsf{ntc}(L) \leq |\Sigma| \cdot (\mathsf{nsc}(L))^2$,

where $\mathsf{sc}(L)$, $\mathsf{tc}(L)$ ($\mathsf{nsc}(L)$, $\mathsf{ntc}(L)$, respectively) refers to the minimum (nsc, ntc, respectively) among all DFAs (NFAs, respectively) accepting L. Similarly, $\mathsf{nsc}_\varepsilon(L)$ ($\mathsf{ntc}_\varepsilon(L)$, respectively) is the minimum nsc_ε (ntc_ε, respectively) among all ε-NFAs for the language L.

As it is defined above, deterministic transition complexity is not an interesting measure by itself, because it is directly related to sc, the deterministic state complexity. But the picture changes when deterministic transition complexity is defined in terms of partial DFAs. Here, a *partial* DFA is an NFA which transition function δ satisfies $|\delta(q,a)| \leq 1$, for all states $q \in Q$ and all alphabet symbols $a \in \Sigma$. A partial DFA cannot save more than one state compared to an ordinary DFA, but it can save a considerable number of transitions in some cases. This phenomenon is studied, e.g., in [30, 75, 76]. Further measures for the complexity of finite automata, in particular measures related to unambiguity and limited nondeterminism, can be found in [39, 40, 41, 59, 63, 70, 71, 72, 92, 93, 95].

Now let us come to measures on regular expressions. While there are the two commonly accepted measures for finite automata, there is no general agreement in the literature about the proper measure for regular expressions. We summarize some important ones: the measure size is defined to be the total number of symbols (including $\emptyset$, ε, symbols from alphabet Σ, all operation symbols, and parentheses) of a completely bracketed regular expression (for example, used in [2], where it is called length). Another measure related to the reverse polish notation of a regular expression is rpn, which gives the number of nodes in the syntax tree of the expressions (parentheses are not counted). This measure is equal to the length of a (parenthesis-free) expression in post-fix notation [2]. The alphabetic width awidth is the total number of alphabetic symbols from Σ (counted with multiplicity) [27, 83]. Relations between these measures have been studied, e.g., in [27, 28, 42, 66].

Theorem 2 *Let $L \subseteq \Sigma^*$ be a regular language. Then*

1. $\mathsf{size}(L) \leq 3 \cdot \mathsf{rpn}(L)$ *and* $\mathsf{size}(L) \leq 8 \cdot \mathsf{awidth}(L) - 3$,
2. $\mathsf{awidth}(L) \leq \frac{1}{2} \cdot (\mathsf{size}(L) + 1)$ *and* $\mathsf{awidth}(L) \leq \frac{1}{2} \cdot (\mathsf{rpn}(L) + 1)$, *and*
3. $\mathsf{rpn}(L) \leq \frac{1}{2} \cdot (\mathsf{size}(L) + 1)$ *and* $\mathsf{rpn}(L) \leq 4 \cdot \mathsf{awidth}(L) - 1$,

where $\mathsf{size}(L)$ ($\mathsf{rpn}(L)$, $\mathsf{awidth}(L)$, respectively) refers to the minimum size (rpn, awidth, respectively) among all regular expressions denoting L.

Further measures for the complexity of regular expressions can be found in [8, 27, 28, 49]. To our knowledge, these latter measures received far less attention to date.

In the remainder of this section we concentrate on two important measures on regular expression and finite automata that at first glance do not seem to be related to each other: *star height* and *cycle rank* or *loop complexity*. Both measures are very important, in particular, for the conversion of finite automata to regular expressions and for proving lower bound results on the latter. Intuitively, the star height of an expression measures the nesting depth of Kleene-star operations. More precisely, for a regular expression, the *star height* is inductively defined by

$$\mathsf{height}(\emptyset) = \mathsf{height}(\varepsilon) = \mathsf{height}(a) = 0,$$
$$\mathsf{height}(s+t) = \mathsf{height}(s \cdot t) = \max\left(\mathsf{height}(s), \mathsf{height}(t)\right),$$

and

$$\mathsf{height}(s^*) = 1 + \mathsf{height}(s).$$

The star height of a regular language L, denoted by $\mathsf{height}(L)$ is then defined as the minimum star height among all regular expressions describing L. The seminal work dealing with the star height of regular expressions [26] established a relation between the theory of regular languages and the theory of digraphs. The *cycle rank*, or *loop complexity*, of a digraph D is defined inductively by the following rules: (i) the cycle rank of an acyclic digraph is zero, (ii) cycle rank of a strongly connected component (SCC) of the digraph with at least one arc is 1 plus the minimum cycle rank among the digraphs obtainable from D by deleting a vertex, and (iii) the cycle rank of a digraph with multiple SCCs equals the maximum cycle rank among the sub-digraphs induced by these components. So, roughly speaking, the cycle rank of a digraph is large if the cycle structure of the digraph is intricate and highly connected. The following relation between cycle rank of automata and star height of regular languages became known as *Eggan's Theorem* [26, 97]:

Theorem 3 *The star height of a regular language L equals the minimum cycle rank among all ε-NFAs accepting L.*

An apparent difficulty with applying Eggan's Theorem is that the minimum is taken over infinitely many automata, and the cycle rank of the minimum DFA for the language does not always attain that minimum. That makes the star height a very intricate property of regular languages. Indeed, the decidability status of the star height problem was open for more than two decades, until a very difficult algorithm was given in [54]. For recent progress on algorithms for the star height problem, the reader is referred to [67]. From the above it is immediate that $\mathsf{height}(L) \leq \mathsf{nsc}(L)$. If the language is given as a regular expression, a result from [43] tells us a much sweeter truth:

Lemma 4 *Let $L \subseteq \Sigma^*$ be a regular language with alphabetic width n. Then height$(L) \leq 3\log(n+1)$.*

The idea behind the proof of this lemma is that we can convert a regular expression into a ε-NFA of similar size. The cycle structure of that automaton is well-behaved; and thus its cycle rank is low compared to the size of the automaton. Then Eggan's Theorem is used to convert the automaton back into a regular expression of low star height.

We return to the relationship between required size and star height of regular expressions later on. Now let us turn our attention to the conversion of regular expressions into equivalent finite automata.

3 From Regular Expressions to Finite Automata

The conversion of regular expressions into small finite automata has been intensively studied for more than half a century. Basically the algorithms can be classified according to whether the output is an ε-NFA, NFA, or even a DFA. In principle one can distinguish between the following three major construction schemes and variants thereof:

1. *Thompson's construction* [100] and optimized versions, such as the *follow automaton* [66, 91],

2. construction of the *position automaton*, or *Glushkov automaton* [38, 83], and
3. computation of the *(partial) derivative automaton* [4, 14].

Further automata constructions from regular expressions can be found in, e.g., [6, 13, 19, 65, 31, 102]. We briefly explain some of these approaches in the course of action—for further readings on the subject we refer to [97].

Thompson's construction [100] was popularized by the implementation of the UNIX command `grep` (globally search a regular expression and print). It amounts to the recursive connection of sub-automata *via* ε-transitions. These sub-automata are connected in parallel for the union, in series for the concatenation, and in an iterative fashion for the Kleene star. This yields an ε-NFA with a linear number of states and transitions. A structural characterization of the Thompson automaton in terms of the underlying digraph is given in [36, 37]. Thompson's classical construction went through several stages of adaption and optimization. The construction with the least usage of ε-transitions was essentially given already in 1961 by Ott and Feinstein [91], which also can be found in [24, 77, 82]—see Figure 1. Later this

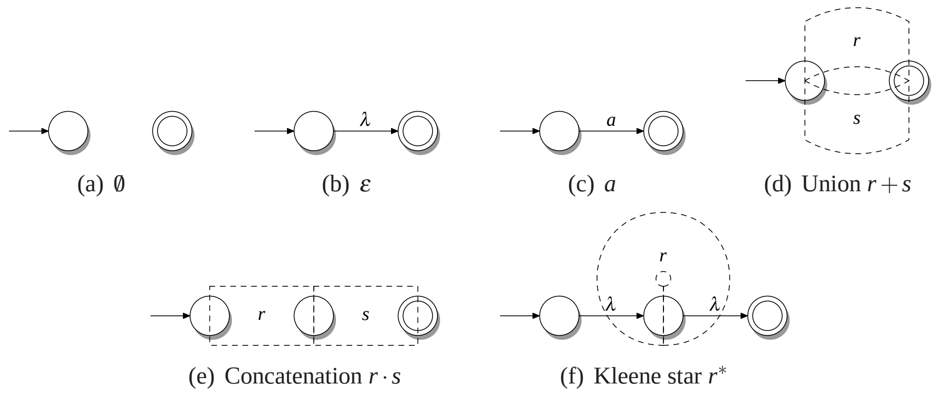


Figure 1: The inductive construction of Ott and Feinstein [91] yielding the precursor of the follow automaton $A_f(r)$ for a regular expression r.

construction was refined by Ilie and Yu [66] and promoted under the name *follow automaton*. In fact, the follow automaton is constructed from a regular expression r by recursively applying the construction of Ott and Feinstein and simultaneously improving on the use of ε-transitions in the following sense: (i) in the concatenation construction a ε-transition into the common state to both sub-automata leads to an appropriate state merging; similarly a state merging is done for an ε-transition leaving the common state, (ii) in the Kleene star construction, if the middle state is on a cycle of ε-transitions, all these transitions are removed, and all states of the cycle are merged, and (iii) after the construction is finished, a possible ε-transition from the start state is removed and both involved states are merged appropriately. Notice, that the automaton thus constructed may still contain ε-transitions. In order to amend the situation, an ε-removal procedure is applied: simply replace any sequence of an ε-transition followed by an a-transition by directly connecting the states on both ends of the sequence by a single a-transition directly. A final step takes care about the ε-transition to the final state. This results in the follow automaton $A_f(r)$ of [66], for the regular expression r.

Example 5 *Imagine a software buffer supporting the actions a ("add work packet") and b ("remove work packet"), with a total capacity of n packets. Let r_n denote the regular expression for the action sequences that result in an empty buffer and never cause the buffer to exceed its capacity. Then*

$$r_1 = (ab)^* \quad \textit{and} \quad r_n = (a \cdot r_{n-1} \cdot b)^*, \quad \textit{for } n \geq 2.$$

Following the construction of the follow automaton as described in [66] results in the automaton depicted in Figure 2. Observe the constructed automaton is minimal, which is not the case in general. This is our

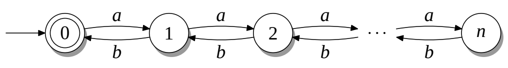


Figure 2: The follow automaton $A_f(r_n)$ accepting $L(r_n)$.

running example, where the behaviour of the state elimination technique described in the next section is discussed in more detail. □

Preliminary bounds on the required size of a finite automaton equivalent to a given regular expression were given in [66]. Later, a tight bound in terms of reverse polish notation [51], and also a tight bound in terms of alphabetic width was found [42]. In the next theorem we summarize the results from [42, 51, 66]—here size of an automaton refers to the sum of the number of states and the number transitions:

Theorem 6 *Let $n \geq 1$, and r be a regular expression of alphabetic width n. Then size $\frac{22}{5}n$ is sufficient for an equivalent ε-NFA accepting $L(r)$. In terms of reverse polish length, the bound is $\frac{22}{15}(rpn(r)+1)+1$. Furthermore, there are infinitely many languages for which both bounds are tight.*

The aid for the tight bound in terms of the alphabetic width stated in the previous theorem is a certain normal form for regular expressions, which is a refinement of the *star normal form* from [13]. The definition reads as follows—transformation into strong star normal form preserves the described language, and is weakly monotone with respect to all usual size measures:

Definition 7 *The operators $\circ$ and $\bullet$ are defined on regular expressions*[2] *over alphabet Σ. The first operator is given by: $a^\circ = a$, for $a \in \Sigma$, $(r+s)^\circ = r^\circ + s^\circ$, $r^{?\circ} = r^\circ$, $r^{*\circ} = r^\circ$; finally, $(r \cdot s)^\circ = r \cdot s$, if $\varepsilon \notin L(rs)$ and $r^\circ + s^\circ$ otherwise. The second operator is given by: $a^\bullet = a$, for $a \in \Sigma$, $(r+s)^\bullet = r^\bullet + s^\bullet$, $(r \cdot s)^\bullet = r^\bullet \cdot s^\bullet$, $r^{*\bullet} = r^{\bullet\circ *}$; finally, $r^{?\bullet} = r^\bullet$, if $\varepsilon \in L(r)$ and $r^{?\bullet} = r^{\bullet ?}$ otherwise. The* strong star normal form *of an expression r is then defined as $r^\bullet$.*

What about the transformation of a regular expression into a finite automaton if ε-transitions are not allowed? One way to obtain an NFA directly is to perform the standard algorithm for removing ε-transitions, see, e.g., [61], which may increase the number of transitions at most quadratically. Another way is to directly implement the procedure during the recursive construction using non-ε-transitions to connect the sub-automata appropriately. Constructions of this kind can be found in, e.g., [3, 69]. For the conversion of ε-NFAs to NFAs the lower bound of [64] applies. There it was shown that there are infinitely many languages which are accepted by ε-NFAs with $O(n \cdot (\log n)^2)$ transitions, such that any NFA needs at least $\Omega(n^2)$ transitions. This lower bound is witnessed by a language over a growing size alphabet and shows that, in this case, the standard algorithm for removing ε-transitions cannot be improved significantly. For the case of binary alphabets, a lower bound of $\Omega(n \cdot 2^{c \cdot \sqrt{\log n}})$, for every $c < \frac{1}{2}$, was proved in [64] as well.

Another possibility to obtain ordinary NFAs is to directly construct the *position automaton*, also called the *Glushkov automaton* [38]—see also [83]. Intuitively, the states of this automaton correspond to the alphabetic symbols or, in other words, to positions between subsequent alphabetic symbols in the regular expression. Let us be more precise: assume that r is a regular expression over Σ of alphabetic

[2]Since $\emptyset$ is only needed to denote the empty set, and the need for ε can be substituted by the operator $L^? = L \cup \{\varepsilon\}$, an alternative is to introduce also the $^?$-operator and instead forbid the use of $\emptyset$ and ε inside non-atomic expressions. This is sometimes more convenient, since one avoids unnecessary redundancy already at the syntactic level [42].

width n. In r we attach subscripts to each letter referring to its position (counted from left to right) in r. This yields a *marked* expression $\overline{r}$ with distinct input symbols over an alphabet $\overline{\Sigma}$ that contains all letters that occur in $\overline{r}$. To simplify our presentation we assume that the same notation is used for unmarking, i.e., $\overline{\overline{r}} = r$. Then in order to describe the position automaton we need to define the following sets of positions on the marked expression. Let $\mathsf{Pos}(r) = \{1, 2, \ldots, \mathsf{awidth}(r)\}$ and $\mathsf{Pos}_0(r) = \mathsf{Pos}(r) \cup \{0\}$. The position set First takes care of the possible beginnings of words in $L(\overline{r})$. It is inductively defined as follows:

$$\begin{aligned} \mathsf{First}(\emptyset) &= \mathsf{First}(\varepsilon) = \emptyset, \\ \mathsf{First}(a_i) &= \{i\}, \\ \mathsf{First}(s+t) &= \mathsf{First}(s) \cup \mathsf{First}(t), \\ \mathsf{First}(s \cdot t) &= \begin{cases} \mathsf{First}(s) \cup \mathsf{First}(t) & \text{if } \varepsilon \in L(s) \\ \mathsf{First}(s) & \text{otherwise,} \end{cases} \end{aligned}$$

and

$$\mathsf{First}(s^*) = \mathsf{First}(s).$$

Accordingly the position set Last takes care of the possible endings of words in $L(\overline{r})$. Its definition is similar to the definition of First, except for the concatenation, which reads as follows:

$$\mathsf{Last}(s \cdot t) = \begin{cases} \mathsf{Last}(s) \cup \mathsf{Last}(t) & \text{if } \varepsilon \in L(t) \\ \mathsf{Last}(t) & \text{otherwise.} \end{cases}$$

Finally, the set Follow takes care about the possible continuations in the words in $L(\overline{r})$. It is inductively defined as

$$\begin{aligned} \mathsf{Follow}(\emptyset) &= \mathsf{Follow}(\varepsilon) = \mathsf{Follow}(a_i) = \emptyset \\ \mathsf{Follow}(s+t) &= \mathsf{Follow}(s) \cup \mathsf{Follow}(t) \\ \mathsf{Follow}(s \cdot t) &= \mathsf{Follow}(s) \cup \mathsf{Follow}(t) \cup \mathsf{Last}(s) \times \mathsf{First}(t) \end{aligned}$$

and

$$\mathsf{Follow}(s^*) = \mathsf{Follow}(s) \cup \mathsf{Last}(s) \times \mathsf{First}(s).$$

Then the position automaton for r is defined as $A_{pos}(r) = (\mathsf{Pos}_0(r), \Sigma, \delta_{pos}, 0, F_{pos})$, where $\delta(0, a) = \{\, j \in \mathsf{First}(\overline{r}) \mid a = \overline{a_j}\,\}$, for every $a \in \Sigma$ and $\delta(i, a) = \{\, j \mid (i, j) \in \mathsf{Follow}(\overline{r}) \text{ and } a = \overline{a_j}\,\}$, for every $i \in \mathsf{Pos}(r)$ and $a \in \Sigma$, and $F_{pos} = \mathsf{Last}(\overline{r})$, if $\varepsilon \notin L(r)$, and $F_{pos} = \mathsf{Last}(\overline{r}) \cup \{0\}$ otherwise.

Example 8 *Consider the regular expression r_n from Example 5. If we mark the regular expression r_n, then we obtain $\overline{r_n} = (a_1(a_2(a_3 \ldots b_{2n-2})^* b_{2n-1})^* b_{2n})^*$. Easy calculations show that the position sets read as follows:*

$$\begin{aligned} \mathit{First}(\overline{r_n}) &= \{1\} \\ \mathit{Last}(\overline{r_n}) &= \{2n\} \end{aligned}$$

and

$$\mathit{Follow}(\overline{r_n}) = \{\, (i, i+1) \mid 1 \le i < 2n \,\} \cup \{\, (i, 2n-i+1), (2n-i+1, i) \mid 1 \le i \le n \,\}$$

The position automaton on state set $Pos_0(r)$ is depicted in Figure 3. Here the set of final states is $F_{pos} = \{0, 2n\}$, since $\varepsilon \in L(r_n)$. Observe, that the follow automaton $A_f(r_n)$ can be obtained from $A_{pos}(r_n)$

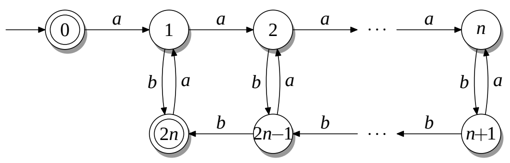


Figure 3: The position automaton $A_{pos}(r_n)$ accepting $L(r_n)$.

by taking the quotient of automata, i.e., merging of states, with respect to the relation $\equiv_f$ *described in [66], which contains the elements* $(i, 2n-i)$*, for* $0 \leq i \leq 2n$*. This leads to the merging of states* 0 *and* $2n$*, states* 1 *and* $2n-1$*, states* 2 *and* $2n-2$*, up to states* $n-1$ *and* $n+1$*.* $\square$

An immediate advantage of the position automaton is observed, e.g., in [1, 7]: for a regular expression r of alphabetic width n, for $n \geq 0$, the position automaton $A_{pos}(r)$ always has precisely $n+1$ states. Simple examples, such as the singleton set $\{a^n\}$, show that this bound is tight. Nevertheless, several optimizations have been developed that give NFAs having often a smaller number of states, while the underlying constructions are mathematically sound refinements of the basic construction. A characterization of the position automaton is given in [16]. Moreover, structural comparisons between the position automaton with its refined versions, namely the *follow automaton*, the *partial derivative automaton* [4], or the *continuation automaton* [7] is given in [18, 66]. The partial derivative automaton is known under different names, such as *equation automaton* [85] or *Antimirov automaton* [4]. Further results on structural properties of these automata, when built from regular expressions in star normal form, can be found in [17, 20]. A quantitative comparison on the sizes of the the aforementioned NFAs for specific languages shows that they can differ a lot. The results listed in Table 1 are taken from [66]—here size of an automaton refers to the sum of the number of states and the number transitions. For comparison rea-

	Finite Automaton			
Expression	$A_f(\cdot)$	$A_{pd}(\cdot)$	$A_{pos}(\cdot)$	$A_{cfs}(\cdot)$
$r_1 = (a_1+\varepsilon)^*$ and $r_{n+1} = (r_n + s_n)^*$ with $s_n = r_n[a_j \mapsto a_{j+2^{n-1}}]$	$\Theta(\lvert r_n\rvert)$	$\Theta(\lvert r_n\rvert^2)$		$\Theta(\lvert r_n\rvert \cdot (\log \lvert r_n\rvert)^2)$
$r_{n,m} = (\sum_{i=1} a_i)(\sum_{i=1}^n a_i + \sum_{i=1}^m b_i)^*$	$\Theta(\lvert r_{n,m}\rvert)$		$\Theta(\lvert r_{n,m}\rvert^2)$	$\Theta(\lvert r_{n,m}\rvert \cdot (\log \lvert r_{n,m}\rvert)^2)$
$r_n = \sum_{i=1}^n a_i \cdot (b_1 + b_2 + \ldots + b_n)^*$	$\Theta(\lvert r_n\rvert)$	$\Theta(\lvert r_n\rvert^{1/2})$	$\Theta(\lvert r_n\rvert^{3/2})$	$\Theta(\lvert r_n\rvert \cdot (\log \lvert r_n\rvert)^2)$
$r_n = (a_1+\varepsilon)\cdot(a_2+\varepsilon)\cdots(a_n+\varepsilon)$	$\Theta(\lvert r_n\rvert^2)$			$\Theta(\lvert r_n\rvert \cdot (\log \lvert r_n\rvert)^2)$

Table 1: Comparing sizes of some automata constructions for specific languages from the literature—gray shading marks the smallest automaton. Here A_f refers to the follow automaton, A_{pd} to the partial derivative automaton, A_{pos} to the position automaton, and A_{cfs} to the common follow set automaton. Moreover, $|r_n|$ ($|r_{n,m}|$, respectively) refers to the alphabetic width of the regular expression r_n ($r_{n,m}$, respectively).

sons also the *common follow set automaton* A_{cfs} is listed—since the description of A_{cfs} is quite involved we refer the reader to [65]. There, this automaton was used to prove an upper bound on the number of transitions. The issue on transitions for NFAs, in particular when changing from an ε-NFA to an NFA, is discussed next.

Despite the mentioned optimizations, except for the common follow set automaton, all of these constructions share the same problem with respect to the number of transitions. An easy upper bound on the number of transitions in the position automaton is $O(n^2)$, independent of alphabet size. It is not hard to prove that the position automaton for the regular expression

$$r_n = (a_1 + \varepsilon) \cdot (a_2 + \varepsilon) \cdots (a_n + \varepsilon)$$

has $\Omega(n^2)$ transitions. It appears to be difficult to avoid such a quadratic blow-up in actual size if we stick to the NFA model. Also if we transform the expression first into a ε-NFA and perform the standard algorithm for removing ε-transitions, see, e.g., [61], we obtain no better result. This naturally raises the question of comparing the descriptional complexity of NFAs over regular expressions. For about forty years, it appears to have been considered as an unproven factoid that a quadratic number of transitions will be inherently necessary in the worst case (cf. [65]). A barely super-linear lower bound of $\Omega(n \log n)$ on the number of transitions of any NFA accepting the language of the expression r_n was proved [65]. More interestingly, the main result of that paper is an algorithm transforming a regular expression of size n into an equivalent NFA with at most $O(n \cdot (\log n)^2)$ transitions. See Figure 4 on how the algorithm of [65] saves transitions for regular expression r_n, explained for $n = 5$. In fact, this upper bound made their lower

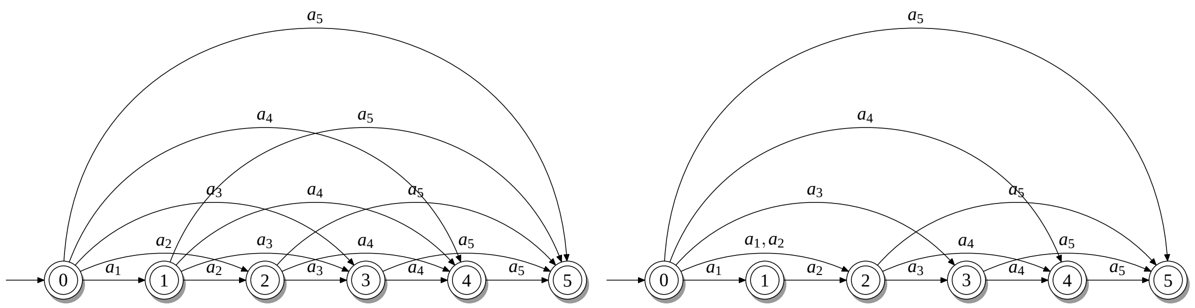


Figure 4: Let $r_n = (a_1 + \varepsilon) \cdot (a_2 + \varepsilon) \cdots (a_n + \varepsilon)$ and $n = 5$. Position automaton $A_{pos}(r_5)$ (left) and its refined version the common follow set automaton $A_{cfs}(r_5)$ (right) accepting language $L(r_5)$; in both cases the dead state and all transitions leading to it are not shown. The automaton $A_{cfs}(r_5)$ is obtained as follows: the state 1 of $A_{pos}(r_5)$ is split such that the new state gets the outgoing transitions labeled with a_3, a_4, and a_5, and is finally identified with state 2, which can be done since it has the same outgoing transitions.

bound look reasonable at once! Shortly thereafter, an efficient implementation of that conversion algorithm was presented [52], and the lower bound was improved in [73] to $\Omega(n \cdot (\log n)^2 / \log \log n)$. Later work [98] established that any NFA accepting language $L(r_n)$ indeed must have at least $\Omega(n \cdot (\log n)^2)$ transitions. So the upper bound of $O(n \cdot (\log n)^2)$ from [65] is asymptotically tight:

Theorem 9 *Let $n \geq 1$ and r be a regular expression of alphabetic width n. Then $O(n \cdot (\log n)^2)$ transitions are sufficient for an NFA to accept $L(r)$. Furthermore, there are infinitely many languages for which this bound is tight.*

Notice that the example witnessing the lower bound is over an alphabet of growing size. For alphabets of size two, the upper bound was improved first [33] to $O(n \cdot \log n)$, and then even to $n \cdot 2^{O(\log^* n)}$, where $\log^*$ denotes the iterated binary logarithm [98]. Moreover, a lower bound of $\Omega(n \cdot (\log k)^2)$ on the size of NFAs with k-letter input alphabet was show in [98], too. Thus the question from [62] whether a conversion from regular expressions over a binary alphabet into NFAs of linear size is possible, is almost settled by now.

Theorem 10 *Let $n \geq 1$ and r be a regular expression of alphabetic width n over a binary alphabet. Then $n \cdot 2^{O(\log^* n)}$ transitions are sufficient for a NFA to accept $L(r)$.*

Next, let us briefly discuss the problem of converting regular expressions to DFAs. Again, this problem has been studied by many authors. The obvious way to obtain a DFA is by applying the well known *subset* or *power-set construction* [94]. Due to this construction the obtained DFA may be of exponential size. A more direct and convenient way is to use Brzozowski's derivatives of expressions [14]. A taxonomy comparing many different conversion algorithms is given in [102]. Regarding the descriptional complexity, a tight bound of $2^n + 1$ states in terms of alphabetic width is given in [69]. The mentioned work also establishes a matching lower bound, but for a rather nonstandard definition of size. In terms of alphabetic width, the best lower bound known to date is from [28]. Together, we have the following result:

Theorem 11 *Let $n \geq 1$ and r be a regular expression of alphabetic width n over a binary alphabet. Then $2^n + 1$ states are sufficient for a DFA to accept $L(r)$. In contrast, for infinitely many n there are regular expressions r_n of alphabetic width n over a binary alphabet, such that the minimal DFA accepting $L(r_n)$ has at least $\frac{5}{4}2^{\frac{n}{2}}$ states.*

Recent developments on the conversion of regular expressions to finite automata show an increasing attention on the study of descriptional complexity in the average case. For instance, in [89] it was shown that, when choosing the expression uniformly at random, the position automaton has $\Theta(n)$ transitions on average, where n refers to the nodes in the parse tree of the expression. A similar result holds w.r.t. alphabetic width, for the position automaton as well as for the partial derivative automaton [11]. A closer look reveals that the number of transitions in the partial derivative automaton is, on average, half the size of the number of transitions in the position automaton [11], for large alphabet sizes; this also holds for the number of states [10]. Results on the average size of ε-NFAs built from Thompson's construction and variants thereof [66, 99, 100] can be found in [12]—in their investigation the authors consider the follow automaton before the final ε-removal is done. Let us call this device *ε-follow automaton*. It turns out that the ε-follow automaton is superior to the other constructions considered. In particular, the number of ε-transitions asymptotically tends to zero, i.e., the ε-follow automaton approaches the follow-automaton.

Almost all of these results were obtained with the help of the framework of analytic combinatorics [29]. The idea to use this approach is quite natural. Recall, that the number of regular expressions of a certain size measured by, e.g., alphabetic width, can be counted by using generating functions—for more involved measures, one has to use multivariate generating functions. To this end one transforms a grammar describing regular expressions such as, e.g., the grammar devised in [50], into a generating function. Since the grammar describes a combinatorial class, the generating function can be obtained by the *symbolic method* of [29], and the coefficients of the power series can be estimated to give approximations of the measure under consideration.

Finally, let us note, that the results on the average size of automata depends on the probability distribution that is used for the average-case analysis. In [90] it was shown that the number of transitions of the position automaton is in $\Theta(n^2)$ under a distribution that is inspired from random binary search trees (BST-like model). To our knowledge, average case analysis under the BST-like model for other automata such as the follow automaton or the partial derivative automaton, has not been conducted so far.

4 From Finite Automata to Regular Expressions

There are a few classical algorithms for converting finite automata into equivalent regular expressions, namely

1. the *algorithm based on Arden's lemma* [5, 22], and
2. the *McNaughton-Yamada algorithm* [83], and
3. the *state elimination technique* [15].

These procedures look different at first glance. We briefly explain the main idea of these approaches—for a detailed description along with an explanation of the differences between the methods, the reader is referred to [97]. There it is shown, that all of the above approaches are more or less reformulations of the same underlying algorithmic idea, and they yield (almost) the same regular expressions.[3]

An algebraic approach to solve the conversion problem from finite automata to regular expressions is the *algorithm based on Arden's lemma* [5, 22]. It puts forward a set of language equations for a given finite automaton. Here, the ith equation describes the set X_i of words w such that the given automaton can go from the ith state to an accepting state on reading w. That system of equations can be resolved by eliminating the indeterminates X_i using a method that resembles Gaussian elimination. But we work in a an algebraic structure different from a field, so for the elimination of variables, we have to resort to *Arden's lemma*:

Lemma 12 *Let Σ be an alphabet, and let $K, L \subseteq \Sigma^*$, where K does not contain the empty word ε. Then the set K^*L is the unique solution to the language equation $X = K \cdot X + L$, where X is the indeterminate.*

Now let us have a look on how Arden's lemma can be applied to our running example.

Example 13 *From the automaton depicted in Figure 2 one reads off the equations*

$$X_0 = a \cdot X_1 + \varepsilon, \quad X_i = a \cdot X_{i+1} + b \cdot X_{i-1}, \quad \text{for } 1 \leq i < n, \text{ and} \quad X_n = b \cdot X_{n-1}.$$

Substituting the right hand side of X_n in the next to last equation and solving it by Arden's lemma results in $X_{n-1} = (ab)^* b \cdot X_{n-2}$. For short, $X_{n-1} = r_1 \cdot b \cdot X_{n-2}$, where r_i is defined as in Example 5. Next this solution is substituted into the equation for X_{n-2}. Solving for X_{n-2} gives us $X_{n-2} = r_2 \cdot b \cdot X_{n-3}$. Proceeding in this way up to the very first equation gives us $X_0 = a \cdot r_{n-1} \cdot b \cdot X_0 + \varepsilon$. The solution to the indeterminate X_0 is according to Arden's lemma $(a \cdot r_{n-1} \cdot b)^* \cdot \varepsilon = r_n$, by applying obvious simplifications. Hence, for instance, in case $n = 6$ we obtain $(a(a(a(a(a(ab)^*b)^*b)^*b)^*b)^*b)^*$. □

The *McNaughton-Yamada algorithm* [83] maintains a matrix with regular expression entries, where the rows and columns are the states of the given automaton. The iterative algorithm uses a ranking on the state set, and proceeds in n rounds, if n is the number of states in the given automaton A. In the matrix $(a_{jk})_{j,k}$ computed in round i, the entry a_{jk} is an expression describing the nonempty labels w of computations of A starting in j and ending in k, such that none of the intermediate states of the computation is ranked higher than i. From these expressions, it is not difficult to obtain a regular expression describing $L(A)$.

Example 14 *Running the McNaughton-Yamada algorithm on the automaton depicted in Figure 2 for $n = 3$ with the ranking $3, 2, 1, 0$ starts with the following matrix:*

$$\begin{array}{c|cccc} & 3 & 2 & 1 & 0 \\ 3 & \emptyset & b & \emptyset & \emptyset \\ 2 & a & \emptyset & b & \emptyset \\ 1 & \emptyset & a & \emptyset & b \\ 0 & \emptyset & \emptyset & a & \emptyset \end{array}$$

[3]Let us also mention that there is another algebraic algorithm from [22], which is based on the recursive decomposition of matrices into blocks. Here, the precise relation to the aforementioned algorithms remains to be investigated [97].

If $(a_{jk})_{j,k}$ denotes the matrix computed in round i, then the matrix $(b_{jk})_{j,k}$ for round $i+1$ can be computed using the rule

$$b_{jk} = a_{jk} + a_{ji}(a_{ii})^* a_{ik}$$

After the first round, the entry in the upper left corner of the matrix reads as $\emptyset + \emptyset\emptyset^*\emptyset$. It is of course helpful to simplify the intermediate regular expressions, by applying some obvious simplifications. As noted in [83], we can use in particular

$$b_{ij} = (a_{ii})^* a_{ij} \quad \textit{and} \quad b_{ji} = a_{ji}(a_{ii})^*.$$

Then the matrix computed in the first round reads as

$$\begin{pmatrix} \emptyset & b & \emptyset & \emptyset \\ a & ab & b & \emptyset \\ \emptyset & a & \emptyset & b \\ \emptyset & \emptyset & a & \emptyset \end{pmatrix},$$

the one from the second round is

$$\begin{pmatrix} b(ab)^*a & b(ab)^* & b(ab)^*b & \emptyset \\ (ab)^*a & (ab)^*ab & (ab)^*b & \emptyset \\ a(ab)^*a & a(ab)^* & a(ab)^*b & b \\ \emptyset & \emptyset & \emptyset & \emptyset \end{pmatrix},$$

and the computation is continued in the same vein. Finally, the entry in the lower-right corner of the matrix reads as $(a(a(ab)^*b)^*b)^*a(a(ab)^*b)^*b$, and the desired regular expression describing L_3 is obtained by adding the empty word: $\varepsilon + (a(a(ab)^*b)^*b)^*a(a(ab)^*b)^*b$.

A few industrious readers, who have worked out the calculation of the previous example until the final matrix, may have observed that many of the intermediate expressions were actually not needed for the final result. Indeed, in a computer implementation [79, page 8] of the basic McNaughton-Yamada algorithm during the 1960s, the author notes: "*a basic fault of the method is that it generates such cumbersome and so numerous expressions initially.*" Below we discuss how the generation of unnecessary sub-expressions can be avoided.

We now come to an algorithm that we describe in greater detail, namely the *state elimination algorithm* [15]. This procedure maintains an extended finite automaton, whose transitions are labeled with regular expressions, rather than alphabet symbols. The computation of an NFA A can be thought of as reading the input word letter by letter, thereby nondeterministically changing its state with each letter in a way that is consistent with its transition table δ. On reading a word $w \in \Sigma$, we say that the finite automaton A and can go on input w from state j to state k, if there is a computation on input w taking A from state j to k. Similarly, for a subset U of the state set Q of the automaton A, we say that A can go on input w from state j *through* U to state k, if there is a computation on input w taking A from state j to k, without going through any state outside U, except possibly j and k. With the *rôles* of j, k, and U fixed as above, we now define the language L^U_{jk} as the set of input words on which the automaton A can go from j to k through U. The state elimination scheme fixes an ordering on the state set Q. Starting with $U = \emptyset$, regular expressions denoting the languages $L^{\emptyset}_{jk}$ for all pairs $(j,k) \in Q \times Q$ can be easily read off from the transition table of A. Now an important observation is that for each state $i \in Q \setminus U$ holds

$$L^{U\cup\{i\}}_{jk} = L^U_{jk} \cup L^U_{ji} \cdot (L^U_{ii})^* \cdot L^U_{ik}.$$

Letting i run over all states according to the ordering, we can grow the set U one by one, in each round computing the intermediate expressions $r_{jk}^{U\cup\{i\}}$ for all j and k. The final regular expression is obtained by utilizing the fact $L(A) = \bigcup_{f\in F} L_{q_0 f}^{Q}$.

As observed already by McNaughton and Yamada [83], we have $\mathtt{awidth}(L_{jk}^{\emptyset}) \leq |\Sigma|$, and each round increases the alphabetic width of each intermediate sub-expression by a factor of at most 4. Another convenient trick is to modify the automaton, by adding a new initial state s and a new final state t to the automaton without altering the language, such that t is the single final state, and there are no transitions entering s or leaving t. Then s and t need not to be added to the set U. Instead, observe that $L(A) = L_{st}^{U}$, with $U = Q \setminus \{s,t\}$. We also note[4] that the computation of r_{jk}^{U} needs to be carried out only for those j and k not in U. We thus obtain the following bound:

Theorem 15 *Let $n \geq 1$ and A be an n-state NFA over alphabet Σ. Then alphabetic width $|\Sigma| \cdot 4^n$ is sufficient for a regular expression describing $L(A)$. Such an expression can be constructed by state elimination.*

In contrast, the state elimination algorithm might suddenly yield a much simpler regular expression once we change the ordering in which the states are eliminated. We illustrate the influence of the elimination ordering on a small example.

Example 16 *Consider our software buffer from Example 5 for $n = 6$. Let $L_n := L(r_n)$. For illustration, a minimal DFA for L_6 is depicted in Figure 5. The two regular expressions*

$$(a(a(a(a(a(ab)^*b)^*b)^*b)^*b)^*b)^*$$

and

$$\varepsilon + a(ab+ba)^*b + a(ab+ba)^*aa\,(ab+ba+bb(ab+ba)^*aa+aa(ab+ba)^*bb)^*\,bb(ab+ba)^*b$$

both describe the language L_6. The first expression is obtained by eliminating the states in the order 6, 5, 4, 3, 2, 1, *and* 0, *while the second expression is produced by the order* 0, 2, 4, 6, 1, 5, *and* 3. *Note*

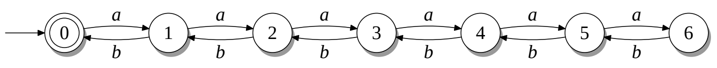


Figure 5: A minimal DFA accepting the language $L_6 := L(r_6)$.

that the expressions have very different structure. The first is much shorter, but has star height 6, *while the second, and longer expression, has star height* 2. *Indeed, in [81] it was shown that the minimum star height among all regular expressions denoting L_n equals $\lfloor \log(n+1) \rfloor$, so the star height of the second expression is optimal. The authors suspect that this language family exhibits a trade-off in the sense that the regular expressions for L_n cannot be simultaneously short and of low star height.*

Perhaps the earliest reference mentioning the influence of the elimination ordering is from 1960. In [83], they proposed to identify the states that "bear the most traffic," i.e., those vertices in the underlying graph with the highest degree, and to eliminate these states at last. Since then, various heuristics for computing elimination orderings that yield short regular expressions have been proposed in the literature. In [74], a simple greedy heuristic was devised. It was proposed to assign a measure to each

[4]The same trick applies for the McNaughton-Yamada algorithm: If the single initial and the single final state are not eliminated, we can erase the entries of the *i*th row and the *i*th column of the computed matrix in round *i*.

state, and this measure is recomputed each time when a state is eliminated. This measure indicates the priority in which the states are eliminated. Observe that eliminating a state tends to introduce new arcs in the digraph underlying the automaton. Thus we can order the states by a measure that is defined as the number of ingoing arcs times the number of outgoing arcs. In [23] a refined version of the same idea is proposed, which takes also the lengths of the intermediate expressions into account, instead of just counting the ingoing and outgoing arcs. Later, a different strategy for accounting the priority of a state was suggested: as measure function, simply take the number of cycles passing through a state. There are some automata, where this heuristic outperforms the one we previously described, but on most random DFAs the performance is comparable. For the heuristic based on counting the number of cycles, recomputing the measure after the elimination of each state does not make a big difference [87]. Another idea is to look for simple structures in finite automata, such as bridge states [53]. A bridge state typically exists if the language under consideration can be written as the concatenation of two nontrivial regular languages. Unfortunately, a random DFA almost surely contains no bridge states at all, as the number of states grows larger [87]. These and other heuristics were compared empirically on a large set of random DFAs as input in [48, 87]. Although there are also advanced strategies for choosing an elimination ordering, which have provable performance guarantees, the greedy heuristic from [23] performs best in most cases.

Beyond heuristics, we can use elimination orderings to prove nontrivial upper bounds on the conversion of DFAs over small alphabets into regular expressions. For the case of binary alphabets, a bound of $O(1.742^n)$ was given in [44], which was then improved to $O(1.682^n)$ in [25]. These bounds can be reached with state elimination by using appropriate elimination orderings. The latest record is $O(1.588^n)$, and the algorithm departs from pure state elimination, see [47].

Theorem 17 *Let $n \geq 1$ and A be an n-state DFA over a binary alphabet. Then size $O(1.588^n)$ is sufficient for a regular expression describing $L(A)$.*

Similar bounds, but with somewhat larger constants in place of 1.588, can be derived for larger alphabets. Moreover, the same holds for NFAs having a comparably low density of transitions.

We sketch how to establish a simpler upper bound than this, which after all gives $o(4^n)$ for all alphabets of constant size. To get things going, assume that we want to determine L_{jk}^{U}, and that the underlying sub-graph induced by U falls apart into two mutually disconnected sub-graphs A and B. Then on reading a word w, the automaton goes from j to k *either* through A *or* through B, and thus $L_{jk}^{U} = L_{jk}^{A} \cup L_{jk}^{B}$, and this is reflected by the regular expressions computed using state elimination. In particular, if the sub-graph induced by U is an independent set, i.e., a set of isolated vertices, in the underlying graph, then $L_{jk}^{U} = \bigcup_{i \in U} L_{jk}^{\{i\}}$. In this case, the blow-up factor incurred by eliminating U is linear in $|U|$, instead of exponential in $|U|$. For a DFA A over constant alphabet, the underlying graph has a linear number of edges. It is known that such graphs have an independent set of size cn, where c is a constant depending on the number of edges. Suppose that U is such an independent set. Then we partition the state set of A into an "easy" part U and a "hard" part $Q \setminus U$. Eliminating U increases the size of the intermediate expressions by a factor linear in $|U|$. Thereafter, eliminating the remaining $(1-c)n$ states may incur a size blow-up by a factor of $4^{(1-c)n}$. Altogether, this gives a regular expression of alphabetic width in $|\Sigma| \cdot o(4^n)$ for $L(A)$.

Let us again take a look at an example.

Example 18 *For illustrating the above said, consider the language*

$$L_3 = (a_1 b_1)^* \sqcup\!\sqcup (a_2 b_2)^* \sqcup\!\sqcup (a_3 b_3)^*,$$

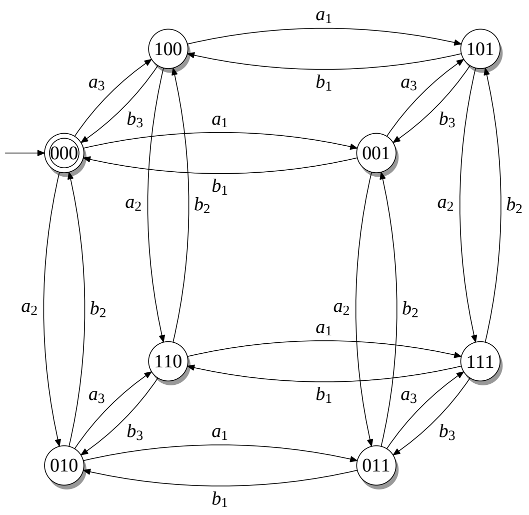


Figure 6: Automaton accepting the language $L_3 = (a_1b_1)^* \sqcup\!\sqcup (a_2b_2)^* \sqcup\!\sqcup (a_3b_3)^*$. The underlying graph is the 3-dimensional cube.

where the interleaving, *or* shuffle, *of two languages L_1 and L_2 over alphabet Σ is*

$$L \sqcup\!\sqcup M = \{\, w \in \Sigma^* \mid w \in x \sqcup\!\sqcup y \text{ for some } x \in L \text{ and } y \in M \,\},$$

and the interleaving $x \sqcup\!\sqcup y$ of two words x and y is defined as the set of all words of the form $x_1y_1x_2y_2\cdots x_ny_n$, where $x = x_1x_2\cdots x_n$, $y = y_1y_2\cdots y_n$ with $x_i, y_i \in \Sigma^*$, for $n \geq 1$ and $1 \leq i \leq n$. Note that in this definition, some of the sub-words x_i and y_i can be empty.

The language L_3 can be accepted by a partial DFA over the state set $\{0,1\}^3$, and whose transition function is given such that input a_i sets the ith bit left of the rightmost bit of the current state from 0 *to* 1*, and input b_i resets the ith bit, again counting from right to left, of the current state from* 1 *to* 0*. All other transitions are undefined. The initial state is* 000*, which is also the single final state. Notice that the graph underlying this automaton is the* 3*-dimensional cube, with* 8 *vertices—see Figure 6. Generalizing this example to $d \geq 3$, the underlying graph of L_d is the d-dimensional hypercube, with 2^d many vertices.*

It is well known that the d-dimensional hypercube is 2*-colorable, and thus has an independent set that contains at least half of the vertices. Eliminating this independent set before the other vertices yields a regular expression of alphabetic width $O(n \cdot 2^n)$, which is way better than the trivial bound of $O(4^n)$.*

We present another application of this idea. Planar finite automata are a special case of finite automata, which were first studied in [9]. To convert a planar finite automaton into a regular expression, one can look for a small set of vertices, whose removal leaves to mutually disconnected sub-graphs with vertex sets A and B. Then again, we have $L^U_{jk} = L^A_{jk} \cup L^B_{jk}$, and this is reflected by the regular expressions computed by state elimination. Since the sub-graphs induced by A and B are again planar, one can apply the trick recursively. Also for this special case, tight upper and lower bounds were found recently [28, 43, 46].

Theorem 19 *Let $n \geq 1$ and A be an n-state planar DFA or NFA over alphabet Σ. Then size $|\Sigma| \cdot 2^{O(\sqrt{n})}$ is sufficient for a regular expression describing $L(A)$. Such an expression can be constructed by state elimination.*

Taking this idea again a step further, one can arrive at a parametrization where the conversion problem from finite automata to regular expressions is fixed-parameter tractable, in the sense that the problem is exponential in that parameter, but not in the size of the input. Recall that we have introduced the concept of cycle rank of a digraph in the course of discussing the star height in Section 2. Now for a digraph D, let D^{sym} denote the symmetric digraph obtained by replacing each arc in D with a pair of anti-parallel arcs. The *undirected cycle rank* of D is defined as the cycle rank of D^{sym}. If the conversion problem from finite automata to regular expressions is parametrized by the undirected cycle rank of the given automaton, one can prove the following bound [46]:

Theorem 20 *Let $n \geq 1$ and A be an n-state DFA or NFA over alphabet Σ, whose underlying digraph is of* undirected *cycle rank at most c, for some $c \geq 1$. Then size $|\Sigma| \cdot 4^c \cdot n$ is sufficient for a regular expression describing $L(A)$. Such an expression can be constructed by state elimination.*

Observe that fixed-parameter tractability also holds in the sense of computational complexity, since computing the undirected cycle rank is fixed-parameter tractable, see, e.g., [96]. A natural question is now whether we can find a similar parametrization in terms of cycle rank, instead of undirected cycle rank. Well, there are acyclic finite automata that require regular expressions of super-polynomial size [27, 49]. Notice that these automata have cycle rank 0. Hence the best we can hope for is a parametrization that is quasi-polynomial when the cycle rank is bounded. One can indeed obtain such an estimate [47], but the method is more technical, and no longer uses only state elimination. The upper bound in terms of directed cycle rank reads as follows:

Theorem 21 *Let $n \geq 1$ and A be an n-state DFA or NFA over alphabet Σ, whose underlying digraph is of cycle rank at most c, for some $c \geq 1$. Then size $|\Sigma| \cdot n^{O(c \cdot \log n)}$ is sufficient for a regular expression describing $L(A)$.*

But in the general case, the exponential blow-up when moving from finite automata to regular expressions is inherent, that is, independent of the conversion method. Already in the 1970s the existence of languages L_n was shown, that admit n-state finite automata, but require regular expressions of alphabetic width at least 2^{n-1}, for all $n \geq 1$, see [27]. Their witness language is over an alphabet of growing size, which is quadratic in the number of states. Their proof technique was tailored to the witness language involved. The question whether a comparable size blow-up can also occur for constant alphabet size [28] was settled only a few years ago. The answer was provided around the same time by two independent groups of researchers, who worked with different proof techniques, and gave different examples [35, 43].

How are such lower bounds established? We shall describe a general method, which has been used to prove lower bounds on regular expression size in various contexts [34, 43, 45, 55]. In the context of lower bounds for regular expression size, a more convenient formulation of Lemma 4 is the star height lemma, which reads as follows:

Lemma 22 *Let L be a regular language. Then $\mathit{awidth}(L) \geq 2^{\Omega(\mathit{height}(L))}$.*

That is, the minimum regular expression size of a regular language is at least exponential in the minimum required star height. But now this looks as if we have replaced one evil with another, since determining the star height is eminently difficult in general [67]. But there is an important special case, in which the star height can be determined more easily: a *partial* deterministic finite automaton is called bideterministic, if it has a single final state, and if the NFA obtained by reversing all transitions and exchanging the roles of initial and final state is again a partial DFA—notice that, by construction, this NFA in any case accepts the reversed language. A regular language L is *bideterministic* if there exists a bideterministic finite automaton accepting L. These languages form a proper subclass of the regular languages. For these languages, *McNaughton's Theorem* [80] states that the star height is equal to the cycle rank of the digraph underlying the minimal partial DFA.

Example 23 *Define $K_m = \{ w \in \{a,b\}^* \mid |w|_a \equiv 0 \mod m \}$ and $L_n = \{ w \in \{a,b\}^* \mid |w|_b \equiv 0 \mod n \}$. For simplicity, assume $m \leq n$. It is straightforward to construct deterministic finite automata with m states (with n states, respectively) arranged in a directed cycle describing the languages K_m and L_n, respectively. By applying the standard product construction on these automata, we obtain a deterministic finite automaton A accepting the language $K_m \cap L_n$. The digraph underlying automaton A is the directed*

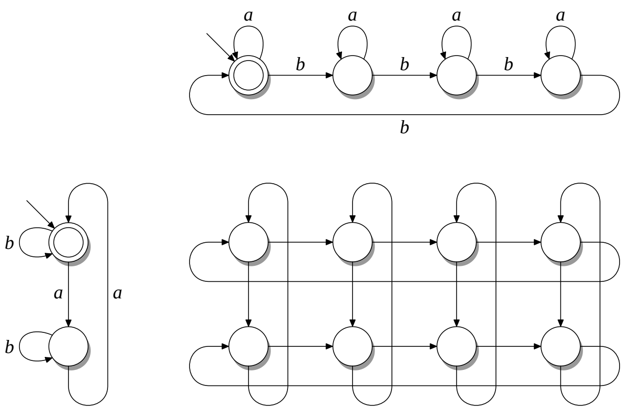


Figure 7: Drawing of the discrete directed $(m \times n)$-torus in the case where $m = 2$ and $n = 4$, induced by the automata for the languages K_m and L_n.

discrete torus. This digraph can be described as the Cartesian graph product of two directed cycles, see Figure 7 for illustration. The cycle rank of the $(m \times n)$-torus is equal to m if $m = n$, and equal to $m+1$ otherwise [43]. It is easily observed that the automaton A is bideterministic, hence the star height of $L(A)$ coincides with the cycle rank of its underlying digraph. By invoking the star height lemma, we can derive a lower bound of $2^{\Omega(m)}$ on the minimum regular expression size required for $L_m \cap K_n$.

For the succinctness gap between DFAs and regular expressions over binary alphabets, a lower bound of $2^{\Omega(\sqrt{n/\log n})}$ was reported in [35], while a parallel effort [43] resulted in an asymptotically tight lower bound of $2^{\Omega(n)}$. We have the following result:

Theorem 24 *Let $n \geq 1$ and A be an n-state DFA or NFA over alphabet Σ. Then size $|\Sigma| \cdot 2^{\Theta(n)}$ is sufficient and necessary in the worst case for a regular expression describing $L(A)$. This already holds for alphabets with at least two letters.*

Recall that the notation $2^{\Theta(n)}$ implies a lower bound of c^n, for some $c > 1$. The hidden constant in the lower bound for binary alphabets is much smaller compared to the lower bound of 2^{n-1} previously obtained in [27] for large alphabets. The upper bound from Theorem 17 implies that c can be at most 1.588 for alphabets of size two. Narrowing down the interval for the best possible c for various alphabet sizes is a challenge for further research.

We turn our attention to interesting special cases of regular languages, namely the *finite* and the *unary* regular languages. Here, the situation is significantly different, as we can harness specialized techniques which are more powerful than state elimination. Also, finite and unary languages have star height at most 1, and thus more tailored techniques than the star height lemma are needed to establish lower bounds. Indeed, the case of finite languages was already addressed in the very first paper on the descriptional complexity of regular expressions [27]. They give a specialized conversion algorithm for finite languages, which is different from the state elimination algorithm. Their results imply that every n-state DFA accepting a finite language can be converted into an equivalent regular expression of size $n^{O(\log n)}$. The method is quite interesting, since it is not based on state elimination, but rather on a clever application of the repeated squaring trick. They also provide a lower bound of $n^{\Omega(\log\log n)}$ when using an alphabet of size $O(n^2)$. The challenge of tightening this gap was settled more than thirty years later in [49], where a lower bound technique from communication complexity is adapted, which originated in the study of monotone circuit complexity.

Theorem 25 *Let $n \geq 1$ and A be an n-state DFA or NFA over alphabet Σ accepting a finite language. Then size $|\Sigma| \cdot n^{\Theta(\log n)}$ is sufficient and necessary in the worst case for a regular expression describing $L(A)$. This still holds for constant alphabets with at least two letters.*

The case of unary languages was discussed in [32, 78, 101]. Here the main idea is that one can exploit the simple cycle structure of unary DFAs and of unary NFAs in Chrobak normal form [21]. In the case of NFAs, elementary number theory helps to save a logarithmic factor of the quadratic upper bound [32]. The main results are summarized in the following theorem.

Theorem 26 *Let $n \geq 1$ and A be an n-state DFA over a unary alphabet. Then size $\Theta(n)$ is sufficient and necessary in the worst case for a regular expression describing $L(A)$. When considering NFAs, the upper bound changes to $O(n^2/\log n)$.*

The tight bounds for the conversion of unary NFAs to regular expressions thus remain to be determined. The conversion problem has been studied also for a few other special cases of finite automata. Examples include finite automata whose underlying digraph is an acyclic series-parallel digraph [88], Thompson digraphs [36], and digraphs induced by Glushkov automata [16].

Acknowledgments. Thanks to Sebastian Jakobi for helpful comments and suggestions on an earlier draft of this paper.

References

[1] A. Aho, R. Sethi & J. D. Ullman (1986): *Compilers: Principles, Techniques, and Tools*. Addison Wesley.

[2] A. V. Aho, J. E. Hopcroft & J. D. Ullman (1974): *The Design and Analysis of Computer Algorithms*. Addision-Wesley.

[3] A. V. Aho & J. D. Ullman (1972): *The Theory of Parsing, Translation and Compiling*. I, Prentice-Hall.

[4] V. Antimirov (1996): *Partial derivatives of regular expressions and finite automaton constructions*. *Theoretical Computer Science* 155(2), pp. 291–319, doi:10.1016/0304-3975(95)00182-4.

[5] D. N. Arden (1961): *Delayed-Logic and Finite-State Machines*. In T. Mott, editor: *Proceedings of the 1st and 2nd Annual Symposium on Switching Theory and Logical Design*, American Institute of Electrical Engineers, New York, Detroit, Michigan, USA, pp. 133–151, doi:10.1109/FOCS.1961.13.

[6] A. Asperti, C. Sacerdoti Coen & E. Tassi (2010): *Regular Expressions, au point*. arXiv:1010.2604v1 [cs.FL].

[7] G. Berry & R. Sethi (1986): *From Regular Expressions to Deterministic Automata*. *Theoretical Computer Science* 48(3), pp. 117–126, doi:10.1016/0304-3975(86)90088-5.

[8] Ph. Bille & M. Thorup (2010): *Regular Expression Matching with Multi-Strings and Intervals*. In M. Charikar, editor: *Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms*, Society for Industrial and Applied Mathematics, Austin, Texas, USA, pp. 1297–1308, doi:10.1137/1.9781611973075.104.

[9] R. V. Book & A. K. Chandra (1976): *Inherently Nonplanar Automata*. *Acta Informatica* 6(1), pp. 89–94, doi:10.1007/BF00263745.

[10] S. Broda, A. Machiavelo, N. Moreira & R. Reis (2010): *On the Average Number of States of Partial Derivative Automata*. In Y. Gao, H. Lu, S. Seki & S. Yu, editors: *Proceedings of the* 14*th International Conference Developments in Language Theory*, *LNCS* 6224, Springer, London, Ontario, Canada, pp. 112–123, doi:10.1007/978-3-642-14455-4_12.

[11] S. Broda, A. Machiavelo, N. Moreira & R. Reis (2012): *On the Average Size of Glushkov and Partial Derivative Automata*. *International Journal of Foundations of Computer Science* 23(5), pp. 969–984, doi:10.1142/S0129054112400400.

[12] S. Broda, A. Machiavelo, N. Moreira & R. Reis (2014): *A Hitchhiker's Guide to Descriptional Complexity Through Analytic Combinatorics*. *Theoretical Computer Science* 528, pp. 85–100, doi:10.1016/j.tcs.2014.02.013.

[13] A. Brüggemann-Klein (1993): *Regular Expressions into Finite Automata*. *Theoretical Computer Science* 120, pp. 197–213, doi:10.1016/0304-3975(93)90287-4.

[14] J. A. Brzozowski (1964): *Derivatives of regular expressions*. *Journal of the ACM* 11, pp. 481–494, doi:10.1145/321239.321249.

[15] J. A. Brzozowski & E. J. McCluskey (1963): *Signal flow graph techniques for sequential circuit state diagrams*. *IEEE Transactions on Computers* C-12(2), pp. 67–76, doi:10.1109/PGEC.1963.263416.

[16] P. Caron & D. Ziadi (2000): *Characterization of Glushkov automata*. *Theoretical Computer Science* 233(1–2), pp. 75–90, doi:10.1016/S0304-3975(97)00296-X.

[17] J.-M. Champarnaud, F. Ouardi & D. Ziadi (2007): *Normalized Expressions and Finite Automata*. *International Journal of Algebra and Computation* 17(1), pp. 141–154, doi:10.1142/S021819670700355X.

[18] J.-M. Champarnaud & D. Ziadi (2002): *Canonical derivatives, partial derivatives and finite automaton constructions*. *Theoretical Computer Science* 289(1), pp. 137–163, doi:10.1016/S0304-3975(01)00267-5.

[19] C.H. Chang & R. Paige (1992): *From Regular Expressions to DFA's Using Compressed NFA's*. In A. Apostolico, M. Chrochemore, Z. Galil & U. Manber, editors: *Proceedings of the* 3*rd Annual Symposium on Combinatorial Pattern Matching*, *LNCS* 644, Springer, Tucson, Arizon, USA, pp. 90–110, doi:10.1007/3-540-56024-6_8.

[20] H. Chen (2010): *Finite Automata of Expressions in the Case of Star Normal Form and One-Unambiguity*. Technical Report ISCAS-LCS-10-11, Chinese Academy of Sciences, Institute of Software, State Key Laboratory of COmputer Science, Beijing 100190 China.

[21] M. Chrobak (1986): *Finite automata and unary languages*. *Theoretical Computer Science* 47, pp. 149–158, doi:10.1016/0304-3975(86)90142-8.

[22] J. H. Conway (1971): *Regular Algebra and Finite Machines*. Chapman and Hall.

[23] M. Delgado & J. Morais (2004): *Approximation to the Smallest Regular Expression for a Given Regular Language*. In M. Domaratzki, A. Okhotin, K. Salomaa & S. Yu, editors: *Proceedings of the* 9*th Conference on Implementation and Application of Automata*, *LNCS* 3317, Springer, Kingston, Ontario, Canada, pp. 312–314, doi:10.1007/978-3-540-30500-2_31.

[24] D.-Z. Du & K.-I. Ko (2001): *Problem Solving in Automata, Languages, and Complexity*. John Wiley & Sons, doi:10.1002/0471224642.

[25] K. Edwards & G. Farr (2012): *Improved Upper Bounds for Planarization and Series-Parallelization of Degree-Bounded Graphs*. *The Electronic Journal of Combinatorics* 19(2), p. #P25.

[26] L. C. Eggan (1963): *Transition graphs and the star height of regular events*. *Michigan Mathematical Journal* 10, pp. 385–397, doi:10.1307/mmj/1028998975.

[27] A. Ehrenfeucht & H. P. Zeiger (1976): *Complexity Measures for Regular Expressions*. *Journal of Computer and System Sciences* 12(2), pp. 134–146, doi:10.1016/S0022-0000(76)80034-7.

[28] K. Ellul, B. Krawetz, J. Shallit & M.-W. Wang (2004): *Regular Expressions: New Results and Open Problems*. *Journal of Automata, Languages and Combinatorics* 9(2/3), pp. 233–256.

[29] Ph. Flajolet & R. Sedgewick (2009): *Analytic Combinatorics*. Cambridge University Press, doi:10.1017/CBO9780511801655.

[30] Y. Gao, K. Salomaa & S. Yu (2011): *Transition Complexity of Incomplete DFAs*. *Fundamenta Informaticae* 110(1–4), pp. 143–158.

[31] P. García, D. López, J. Ruiz & G. I. Álvarez (2011): *From regular expressions to smaller NFAs*. *Theoretical Computer Science* 412, pp. 5802–5807, doi:10.1016/j.tcs.2011.05.058.

[32] P. Gawrychowski (2011): *Chrobak Normal Form Revisited, with Applications*. In B. Bouchou-Markhoff, P. Caron, J.-M. Champarnaud & D. Maurel, editors: *Proceedings of the* 16*th Conference on Implementation and Application of Automata*, *LNCS* 6807, Springer, Blois, France, pp. 142–153, doi:10.1007/978-3-642-22256-6_14.

[33] V. Geffert (2003): *Translation of binary regular expressions into nondeterministic ε-free automata with $O(n\log n)$ transitions*. *Journal of Computer and System Sciences* 66(3), pp. 451–472, doi:10.1016/S0022-0000(03)00036-9.

[34] W. Gelade (2010): *Succintness of regular expressions with interleaving, intersection, and counting*. *Theoretical Computer Science* 411(31–33), pp. 2987–2998, doi:10.1016/j.tcs.2010.04.036.

[35] W. Gelade & F. Neven (2008): *Succinctness of Complement and Intersection of Regular Expressions*. In S. Albers & P. Weil, editors: *Proceedings of the* 25*th International Symposium on Theoretical Aspects of Compter Science*, *Leibniz International Proceedings in Informatics* 1, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, Bordeaux, France, pp. 325–336.

[36] D. Giammarresi, J.-L. Ponty, D. Wood & D. Ziadi (2004): *A Characterization of Thompson Digraphs*. *Discrete Applied Mathematics* 134(1–3), pp. 317–337, doi:10.1016/S0166-218X(03)00299-3.

[37] D. Giammarresi, J.-L. Pony & D. Wood (1999): *Thompson Languages*. In J. Karhumäki, H. Maurer, G. Păun & G. Rozenberg, editors: *Jewels are Forever: Contributions on Theoretical Computer Science in Honor of Arto Salomaa*, Springer, pp. 16–24, doi:10.1007/978-3-642-60207-8_2.

[38] V. M. Glushkov (1961): *The abstract theory of automata*. *Russian Mathematics Surveys* 16, pp. 1–53, doi:10.1070/RM1961v016n05ABEH004112.

[39] J. Goldstine, M. Kappes, C. M. R. Kintala, H. Leung, A. Malcher & D. Wotschke (2002): *Descriptional Complexity of Machines with Limited Resources*. *Journal of Universal Computer Science* 8(2), pp. 193–234, doi:10.1142/9781848165458_0001.

[40] J. Goldstine, C. M. R. Kintala & D. Wotschke (1990): *On Measuring Nondeterminism in Regular Languages*. *Information and Computation* 86(2), pp. 179–194, doi:10.1016/0890-5401(90)90053-K.

[41] J. Goldstine, H. Leung & D. Wotschke (1992): *On the relation between amibuity and nondeterminism in finite automata*. *Information and Computation* 100, pp. 261–170, doi:10.1016/0890-5401(92)90014-7.

[42] H. Gruber & St. Gulan (2010): *Simplifying Regular Expressions*. In A. H. Dediu, H. Fernau & C. Martín-Vide, editors: *Proceedings of the* 4*th International Conference Language and Automata Theory and Applications*, *LNCS* 6031, Springer, Trier, Germany, pp. 285–296, doi:10.1007/978-3-642-13089-2_24.

[43] H. Gruber & M. Holzer (2008): *Finite Automata, Digraph Connectivity, and Regular Expression Size*. In L. Aceto, I. Damgaard, L. A. Goldberg, M. M. Halldórsson, A. Ingólfsdóttir & I. Walkuwiewicz, editors: *Proceedings of the* 35*th International Colloquium on Automata, Languages and Propgramming*, *LNCS* 5126, Springer, Reykjavik, Iceland, pp. 39–50, doi:10.1007/978-3-540-70583-3_4.

[44] H. Gruber & M. Holzer (2008): *Provably Shorter Regular Expressions from Deterministic Finite Automata (Extended Abstract)*. In M. Ito & M. Toyama, editors: *Proceedings of the* 12*th International Conference Developments in Language Theory*, *LNCS* 5257, Springer, Kyoto, Japan, pp. 383–395, doi:10.1007/978-3-540-85780-8_30.

[45] H. Gruber & M. Holzer (2009): *Tight Bounds on the Descriptional Complexity of Regular Expressions*. In V. Diekert & D. Nowotka, editors: *Proceedings of the* 13*th International Conference Developments in Language Theory*, *LNCS* 5583, Springer, Stuttgart, Germany, pp. 276–287, doi:10.1007/978-3-642-02737-6_22.

[46] H. Gruber & M. Holzer (2013): *Provably Shorter Regular Expressions From Finite Automata*. *International Journal of Foundations of Computer Science* 24(8), pp. 1255–1279, doi:10.1142/S0129054113500330.

[47] H. Gruber & M. Holzer (2014): *Regular Expressions From Deterministic Finite Automata, Revisited*. IFIG Research Report 1403, Institut für Informatik, Justus-Liebig-Universität Gießen, Arndtstr. 2, D-35392 Gießen, Germany.

[48] H. Gruber, M. Holzer & M. Tautschnig (2009): *Short Regular Expressions from Finite Automata: Empirical Results*. In S. Maneth, editor: *Proceedings of the* 14*th Conference on Implementation and Application of Automata*, *LNCS* 5642, Springer, Sydney, Australia, pp. 188–197, doi:10.1007/978-3-642-02979-0_22.

[49] H. Gruber & J. Johannsen (2008): *Tight Bounds on the Descriptional Complexity of Regular Expressions*. In R. Amadio, editor: *Proceedings of the* 11*th Conference Foundations of Software Science and Computational Structures*, *LNCS* 4962, Springer, Budapest, Hungary, pp. 273–286, doi:10.1007/978-3-540-78499-9_20.

[50] H. Gruber, J. Lee & J. Shallit (2012): *Enumerating regular expressions and their languages*. arXiv:1204.4982 [cs.FL].

[51] St. Gulan & H. Fernau (2008): *An Optimal Comstruction of Finite Automata From Regular Expressions*. In R. Hariharan, M. Mukund & V. Vinay, editors: *Proceedings of the* 28*th Conference on Foundations of Software Technology and Theoretical Compter Science*, *Dagstuhl Seminar Proceedings* 08002, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, Bangalore, India, pp. 211–222.

[52] Ch. Hagenah & A. Muscholl (2000): *Computing ε-free NFA from regular expressions in $O(n\log^2(n))$ time*. *RAIRO–Informatique théorique et Applications / Theoretical Informatics and Applications* 34(5), pp. 257–277, doi:10.1051/ita:2000116.

[53] Y.-S. Hand & D. Wood (2007): *Obtaining shorter regular expressions from finite-state automata*. *Theoretical Computer Science* 370(1–3), pp. 110–120, doi:10.1016/j.tcs.2006.09.025.

[54] K. Hashiguchi (1988): *Algorithms for determining the relative star height and star height*. *Information and Computation* 78(2), pp. 124–169, doi:10.1016/0890-5401(88)90033-8.

[55] M. Holzer & S. Jakobi (2011): *Chop Operations and Expressions: Descriptional Complexity Considerations*. In G. Mauri & A. Leporati, editors: *Proceedings of the* 15*th International Conference Developments in Language Theory*, *LNCS* 6795, Springer, Milan, Italy, pp. 264–275, doi:10.1007/978-3-642-22321-1_23.

[56] M. Holzer & M. Kutrib (2009): *Nondeterministic Finite Automata—Recent Results on the Descriptional and Computational Complexity*. *International Journal of Foundations of Computer Science* 20(4), pp. 563–580, doi:10.1142/S0129054109006747.

[57] M. Holzer & M. Kutrib (2010): *The Complexity of Regular(-Like) Expressions*. In Y. Gao, H. Lu, S. Seki & S. Yu, editors: *Proceedings of the* 14*th International Conference Developments in Language Theory*, *LNCS* 6224, Springer, London, Ontario, Canada, pp. 16–30, doi:10.1007/978-3-642-14455-4_3.

[58] M. Holzer & M. Kutrib (2010): *Descriptional Complexity—An Introductory Survey*. In C. Martín-Vide, editor: *Scientific Applications of Language Methods*, World Scientific, pp. 1–58, doi:10.1142/9781848165458_0001.

[59] M. Holzer & M. Kutrib (2010): *Descriptional Complexity of (Un)ambiguous Finite State Machines and Pushdown Automata*. In A. Kucera & I. Potapov, editors: *Proceedings of the* 4*th Workshop on Reachability Problems*, *LNCS* 6227, Springer, Brno, Czech Republic, pp. 1–23, doi:10.1007/978-3-642-15349-5_1.

[60] M. Holzer & M. Kutrib (2011): *Descriptional and Computational Complexity of Finite Automata—A Survey*. *Information and Computation* 209(3), pp. 456–470, doi:10.1016/j.ic.2010.11.013.

[61] J. E. Hopcroft & J. D. Ullman (1979): *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley.

[62] J. Hromkovič (2002): *Descriptional Complexity of Finite Automata: Concepts and Open Problems*. *Journal of Automata, Languages and Combinatorics* 7(4), pp. 519–531.

[63] J. Hromkovič, J. Karhumäki, H. Klauck, G. Schnitger & S. Seibert (2002): *Communication complexity method for measuring nondeterminism in finite automata*. *Information and Computation* 172(2), pp. 202–217, doi:`10.1006/inco.2001.3069`.

[64] J. Hromkovič & G. Schnitger (2005): *NFAs with and without ε-transitions*. In L. Caires, G. F. Italiano, L. Monteiro, C. Palamidessi & M. Yung, editors: *Proceedings of the* 32*nd International Colloquium Automata, Languages and Programming*, *LNCS* 3580, Springer, Lisbon, Portugal, pp. 385–396, doi:`10.1007/11523468_32`.

[65] J. Hromkovič, S. Seibert & Th. Wilke (2001): *Translating Regular Expressions into Small ε-Free Automata*. *Journal of Computer and System Sciences* 62(4), pp. 565–588, doi:`10.1006/jcss.2001.1748`.

[66] L. Ilie & S. Yu (2003): *Follow automata*. *Information and Computation* 186(1), pp. 140–162, doi:`10.1016/S0890-5401(03)00090-7`.

[67] D. Kirsten (2005): *Distance desert automata and the star height problem*. *RAIRO–Informatique théorique et Applications / Theoretical Informatics and Applications* 39(3), pp. 455–509, doi:`10.1051/ita:2005027`.

[68] S. C. Kleene (1956): *Representation of events in nerve nets and finite automata*. In C. E. Shannon & J. McCarthy, editors: *Automata studies*, *Annals of mathematics studies* 34, Princeton University Press, pp. 2–42.

[69] E. Leiss (1981): *The complexity of restricted regular expressions and the synthesis problem for finite automata*. *Journal of Computer and System Sciences* 23(3), pp. 348–254, doi:`10.1016/0022-0000(81)90070-2`.

[70] H. Leung (1998): *On Finite Automata with Limited Nondeterminism*. *Acta Informatica* 35(7), pp. 595–624, doi:`10.1007/s002360050133`.

[71] H. Leung (1998): *Separating exponentially ambiguous finite automata from polynomially ambiguous finite automata*. *SIAM Journal on Computing* 27(4), pp. 1073–1082, doi:`10.1137/S0097539793252092`.

[72] H. Leung (2005): *Descriptional complexity of NFA of different ambiguity*. *International Journal of Foundations of Computer Science* 16(5), pp. 975–984, doi:`10.1142/S0129054105003418`.

[73] Y. Lifshits (2003): *A lower bound on the size of ε-free NFA corresponding to a regular expression*. *Information Processing Letters* 85(6), pp. 293–299, doi:`10.1016/S0020-0190(02)00436-2`.

[74] S. Lombardy, Y. Régis-Gianas & J. Sakarovitch (2004): *Introducing VAUCANSON*. *Theoretical Computer Science* 328(1–2), pp. 77–96, doi:`10.1016/j.tcs.2004.07.007`.

[75] E. Maia, N. Moreira & R. Reis (2013): *Incomplete Transition Complexity of Basic Operations on Finite Languages*. In S. Konstantinidis, editor: *Proceedings of the* 18*th International Conference on Implementation and Application of Automata*, *LNCS* 7982, Springer, Halifax, Nova Scotia, Canada, pp. 349–356, doi:`10.1007/978-3-642-39274-0_31`.

[76] E. Maia, N. Moreira & R. Reis (2013): *Incomplete Transition Complexity of Some Basic Operations*. In P. v. Emde Boas, F. C. A. Groen, G. F. Italiano, J. R. Nawrocki & H. Sack, editors: *Proceedings of the* 39*th International Conference on Current Trends in Theory and Practice of Computer Science*, *LNCS* 7741, Springer, Špindlerøuv Mlýn, Czech Republic, pp. 319–331.

[77] Z. Manna (1974): *Mathematical Theory of Computation*. McGraw-Hill.

[78] A. Martinez (2002): *Efficient Computation of Regular Expressions from Unary NFAs*. In J. Dassow, M. Hoeberechts, H. Jürgensen & D. Wotschke, editors: *Pre-Proceedings of the* 4*th Workshop on Descriptional Complexity of Formal Systems*, *Report No.* 586, Department of Computer Science, The University of Western Ontario, Canada, London, Ontario, Canada, pp. 216–230.

[79] H. V. McIntosh (1968): *REEX: A CONVERT Program to Realize the McNaughton-Yamada Analysis Algorithm*. Technical Report AIM-153, MIT Artificial Intelligence Laboratory.

[80] R. McNaughton (1967): *The loop complexity of pure-group events*. *Information and Control* 11(1–2), pp. 167–176, doi:`10.1016/S0019-9958(67)90481-0`.

[81] R. McNaughton (1969): *The loop complexity of regular events*. *Information Sciences* 1, pp. 305–328, doi:`10.1016/S0020-0255(69)80016-2`.

[82] R. McNaughton (1982): *Elementary computability, formal languages, and automata*. Prentice-Hall.

[83] Robert McNaughton & Hisao Yamada (1960): *Regular expressions and state graphs for automata*. *IRE Transactions on Electronic Computers* EC-9(1), pp. 39–47, doi:`10.1109/TEC.1960.5221603`.

[84] A. R. Meyer & M. J. Fischer (1971): *Economy of description by automata, grammars, and formal systems*. In: *Proceedings of the* 12*th Annual Symposium on Switching and Automata Theory*, IEEE Computer Society Press, pp. 188–191, doi:`10.1109/T-C.1971.223108`.

[85] B. G. Mirkin (1966): *An Algorithm for Constructing a Base in a Language of Regular Expressions*. *Engineering Cybernetics* 5, pp. 110–116.

[86] F. R. Moore (1971): *On the bounds for state-set size in the proofs of equivalence between deterministic, nondeterministic, and two-way finite automata*. *IEEE Transaction on Computing* C-20, pp. 1211–1219, doi:`10.1109/T-C.1971.223108`.

[87] N. Moreira, D. Nabais & R. Reis (2010): *State Elimination Ordering Strategies: Some Experimental Results*. In I. McQuillan & G. Pighizzini, editors: *Proceedings of the* 12*th Workshop on Descriptional Complexity of Formal Systems*, *EPTCS* 31, Saskatoon, Saskatchewan, Canada, pp. 139–148.

[88] N. Moreira & R. Reis (2009): *Series-Parallel Automata and Short Regular Expressions*. *Fundamenta Informaticae* 91(3–4), pp. 611–629.

[89] C. Nicaud (2009): *On the Average Size of Glushov's Automaton*. In A. H. Dediu, A. M. Ionescu & C. Martín-Vide, editors: *Proceedings of the* 3*rd International Conference Language and Automata Theory and Applications*, *LNCS* 5457, Springer, Tarragona, Spain, pp. 626–637, doi:`10.1007/978-3-642-00982-2_53`.

[90] C. Nicaud, C. Pivoteau & B. Razet (2010): *Average Analysis of Glushkov Automata under a BST-Like Model*. In K. Lodaya & M. Mahajan, editors: *Proceedings of the* 30*th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, *Leibniz International Proceedings in Informatics* 8, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, Chennai, India, pp. 388–399.

[91] G. Ott & N. H. Feinstein (1961): *Design of Sequential Machines From Their Regular Expressions*. *Journal of the ACM* 8(4), pp. 585–600, doi:`10.1145/321088.321097`.

[92] A. Palioudakis, K. Salomaa & S. Akl (2013): *Comparisons Between Measures of Nondeterminism on Finite Automata*. In H. Jürgensen & R. Reis, editors: *Proceedings of the* 15*th International Workshop Descriptional Complexity of Formal Systems*, *LNCS* 8031, Springer, London, Ontario, Canada, pp. 229–240, doi:`10.1007/978-3-642-39310-5_21`.

[93] A. Palioudakis, K. Salomaa & S. G. Akl (2012): *State Complexity and Limited Nondeterminism*. In M. Kutrib, N. Moreira & R. Reis, editors: *Proceedings of the* 14*th Workshop on Descriptional Complexity of Formal Systems*, *LNCS* 7386, Springer, Braga, Portugal, pp. 252–265, doi:`10.1007/978-3-642-31623-4_20`.

[94] M. O. Rabin & D. Scott (1959): *Finite Automata and Their Decision Problems*. *IBM Journal of Research and Development* 3, pp. 114–125, doi:`10.1147/rd.32.0114`.

[95] B. Ravikumar & O. H. Ibarra (1989): *Relating the Type of Ambiguity of Finite Automata to the Succinctness of Their Representation*. *SIAM Journal on Computing* 18(6), pp. 1263–1282, doi:`10.1137/0218083`.

[96] F. Reidl, P. Rossmanith, F. Sánchez Villaamil & S. Sikdar (2014): *A Faster Parameterized Algorithm for Treedepth*. arXiv:1401.7540v3 [cs.DS].

[97] J. Sakarovitch (2009): *Elements of Automata Theory*. Cambridge University Press, doi:`10.1017/CBO9781139195218`.

[98] G. Schnitger (2006): *Regular Expressions and NFAs Without ε-Transitions*. In B. Durand & W. Thomas, editors: *Proceedings of the* 23*th International Symposium on Theoretical Aspects of Computer Science*, *LNCS* 3884, Springer, Marseille, France, pp. 432–443.

[99] S. Sippu & E. Soisalon-Soininen (1988): *Parsing Theory, Volume I: Languages and Parsing*. *EATCS Monographs on Theoretical Computer Science* 15, Springer, doi:`10.1007/978-3-642-61345-6`.

[100] K. Thompson (1968): *Regular Expression Search Algorithm*. *Communications of the ACM* 11(6), pp. 419–422, doi:`10.1145/363347.363387`.

[101] A. W. To (2009): *Unary finite automata vs. arithmetic progressions*. *Information Processing Letters* 109(17), pp. 1010–1014, doi:`10.1016/j.ipl.2009.06.005`.

[102] B. W. Watson (1995): *Taxonomies and Toolkits of Regular Language Algorithms*. PhD thesis, Eindhoven University of Technology, Department of Mathematics and Computer Science, Den Dolech 2, 5612 AZ Eindhoven, The Netherlands.

[103] S. Yu (2001): *State complexity of regular languages*. *Journal of Automata, Languages and Combinatorics* 6, pp. 221–234.

On Varieties of Automata Enriched with an Algebraic Structure (Extended Abstract)

Ondřej Klíma
Department of Mathematics and Statistics,
Masaryk University,
Brno, Czech Republic [*]
klima@math.muni.cz

Eilenberg correspondence, based on the concept of syntactic monoids, relates varieties of regular languages with pseudovarieties of finite monoids. Various modifications of this correspondence related more general classes of regular languages with classes of more complex algebraic objects. Such generalized varieties also have natural counterparts formed by classes of finite automata equipped with a certain additional algebraic structure. In this survey, we overview several variants of such varieties of enriched automata.

Algebraic theory of regular languages is a well established field in the theory of formal languages. A basic ambition of this theory is to obtain effective characterizations of various natural classes of regular languages. The fundamental concept is the notion of *syntactic monoid* of a given regular language L, which is the smallest possible monoid recognizing the language L, and which is isomorphic to the transition monoid of the minimal automaton of L. First examples of natural classes of languages, which were effectively characterized by properties of syntactic monoids, were the star-free languages [34] having aperiodic syntactic monoids and the piecewise testable languages [36] having $\mathscr{J}$-trivial syntactic monoids. A general framework for discovering relationships between properties of regular languages and properties of monoids was provided by Eilenberg [9], who established a one-to-one correspondence between so-called *varieties* of regular languages and *pseudovarieties* of finite monoids. Here varieties of languages are classes closed under taking quotients, preimages under morphisms and Boolean operations. On the other hand pseudovarieties of finite monoids are classes closed under taking finite direct products, submonoids and morphic images. Thus a membership problem for a given variety of regular languages can be translated to a membership problem for the corresponding pseudovariety of finite monoids. An advantage of this translation is that pseudovarieties of monoids are exactly classes of finite monoids which have equational description by pseudoidentities [32].

The goal of this contribution is not to overview all notions and applications of the algebraic theory of regular languages. For thorough introduction to that theory we refer to [25]. Other overviews are for example [26] and [44]. A more detailed information concerning the theory of pseudovarieties of finite monoids can be found in the survey [2] or in the books [1] and [33].

We should mention that many interesting classes of regular languages, which are studied by the algebraic methods, come from logic. It is well known that regular languages which are definable in the first order logic of finite linear orderings are exactly star-free languages [23]. Within the class of star-free languages, there were defined the so-called *dot-depth hierarchy* [7] and closely related *Straubing–Thérien hierarchy* [38, 41]. In [42] it was shown that a language belongs to the nth level of the latter

[*]Supported by the project “Algebraic Methods in Quantum Logic” by ESF, No. CZ.1.07/2.3.00/20.0051.

hierarchy if and only if it is definable by a formula with n alternations of quantifiers. Moreover, the class of star-free languages is exactly the class of all languages definable by linear temporal logic [15]. For a recent survey on the classes of languages given by fragments of first-order logic we refer to [8] and [40]. Some recent results can be found for example in [19] and [29].

Since not every natural class of languages is closed under all mentioned operations, various generalizations of the notion of varieties of languages were studied. One possible generalization is the notion of *positive varieties* of languages introduced in [24] for which an equational characterization was given in [27]; the positive varieties need not be closed under complementation. In the same direction one can consider varieties which need not be closed under taking unions (see [30]). We shall return to these concepts later. Another possibility is to weaken the closure property for preimages under morphisms. In this way one can consider $\mathscr{C}$-varieties of regular languages which were introduced in [39] and whose equational description was given in [20]. Here we require the presence of preimages under morphisms only for morphisms from a certain special class $\mathscr{C}$. An important example is the class formed by morphisms which map letters to letters; such varieties of languages (so-called *literal varieties)* and the corresponding pseudovarieties of monoids with marked generators (so-called *monoid-generator pairs*) were studied in [12]. Classes of languages with a complete absence of the preimages requirement were studied in [13].

In our contribution we would like to consider varieties of automata as another natural counterpart to varieties of regular languages. We should emphasize that the considered automata are deterministic finite automata. Characterizing of varieties of languages by properties of minimal automata is quite natural, since usually we assume that an input of a membership problem for a fixed variety of languages is given exactly by a minimal deterministic automaton. For example, if we want to effectively test whether an input language is piecewise testable, we do not want to compute its syntactic monoid which could be quite large[1]. Instead of that we consider a condition which must be satisfied by its minimal automaton and which was given in the original Simon's paper [36]. This characterization was used in [37] and [43] to obtain a polynomial and quadratic algorithm, respectively, for testing piecewise testability. In [18] Simon's condition was reformulated and the so-called *locally confluent acyclic automata*[2] were defined. Therefore we are looking for a general definition of a term *variety of automata*, to obtain a setting in which we could talk, for example, about the variety of locally confluent acyclic automata.

Let us consider a minimal automaton $\mathscr{A}_L$ of a regular language L. A first easy observation is the following: if we change the initial state in $\mathscr{A}_L$ then the resulting automaton recognizes a left quotient of the original language L. Similarly but not trivially, if we change the final states, the resulting automaton recognizes a Boolean combination of right quotients of the original language L. Since we are interested in characterizations of varieties of languages, the choice of an initial state and final states can be left free and we can consider only underlying labeled graphs[3] which will form our varieties of automata. Furthermore, since varieties of languages are closed under taking unions and intersections, we need to include direct products of automata in our varieties of automata. Considering a preimage of a given regular language L under some morphism f, one can construct an automaton from the minimal automaton $\mathscr{A}_L$ of L, so-called *f-subautomaton*, where states form a subset and a new action by each letter a is the same as the action by the word $f(a)$ in the original automaton. Since these constructions generate new automata, namely products of automata and f-subautomata, and since we are mainly interested in minimal automata, we also include into our variety of automata all morphic images of existing automata. Finally, from technical reasons we add disjoint unions of automata. Thus a variety of automata will be a class of automata closed

[1]More than $(n-1)!$ where n is the number of states of the minimal automaton, see [5] for precise bounds.

[2]These automata recognize exactly piecewise testable languages and paper [18] contains a new (purely automata based) proof which does not use Simon's original result.

[3]Such automata without initial and final states are sometimes called semiautomata in the literature.

under taking products, disjoint unions, morphic images and f-subautomata. And of course, when we are limited to morphisms from a certain class $\mathscr{C}$, we can even talk about $\mathscr{C}$-varieties of automata. Then one can prove an Eilenberg type correspondence: varieties of languages correspond to varieties of automata. This concept occurred in [10] in the case of literal morphisms and in [6] under the name varieties of $\mathscr{C}$-actions. In particular, one can consider the variety of all counter-free automata [23] characterizing star-free languages or the variety of all locally confluent acyclic automata.

Now we enrich automata by an algebraic structure. If we start with a deterministic automaton where all states are reachable from the initial one then we can assign to each state q the set L_q consisting of all words which are acceptable if the computation starts from this state. Sometimes L_q is called the *future* of the state q. It is known [4] that identifying the states with the same future produces a minimal automaton. Thus a state q in the minimal automaton $\mathscr{A}_L$ can be identified with its future L_q and therefore it is a subset of A^*. Then such states are ordered by inclusion, which means that each minimal automaton is implicitly equipped with a partial order. Moreover, final states[4] form an upward closed subset. This leads to a notion of partially ordered automata where actions by letters are isotone mappings and languages are recognized by final states which form an upward closed subset.

Furthermore, varieties, or more generally $\mathscr{C}$-varieties, of partially ordered automata[5] can be defined once again as classes which are closed under taking products, disjoint unions, morphic images and f-subautomata. Now one can prove that these varieties of partially ordered automata correspond to positive varieties of languages. A well known example is the level $1/2$ in the Straubing-Thérien hierarchy of star-free languages. The effective characterization of the level $1/2$ can be found in [3]. This characterization can be equivalently stated as validity of the identity $1 \leq x$ in the syntactic ordered monoid of a language [28]. Therefore, the corresponding variety of partially ordered automata is formed by automata where actions are increasing mappings (for a state q and a letter a we have $q \cdot a \geq q$).

Now we return to the representation of the minimal automaton $\mathscr{A}_L$ of a regular language L where a state $q = L_q$ is a subset of A^* and we consider all possible intersections of states. Since we have only finitely many states in $\mathscr{A}_L$, we obtain finitely many intersections. The resulting meet-semilattice $\mathscr{S}_L$ can be naturally equipped with actions by letters: applying a letter a to an intersection $\bigcap_{i \in I} q_i$, i.e. a state in $\mathscr{S}_L$, is the intersection of all states $q_i \cdot a$. If we use as final states those which contain the empty word, then final states form a principal filter in the semilattice $\mathscr{S}_L$. This idea leads to a notion of a *meet automaton* which was introduced in [16]. Here the corresponding varieties of languages are not closed under taking unions, since in the product of automata the corresponding set of final states is not a principal filter. Therefore the corresponding classes of regular languages are conjunctive varieties which were defined in [30]. We have already mentioned that the syntactic (ordered) monoid of a language is isomorphic to the transition (ordered) monoid of the (ordered) minimal automaton of the language. Analogous statement is valid in the case of meet automata. In particular, the *canonical meet automaton* $\mathscr{S}_L$ of a language L is a minimal meet automaton of a given language. Moreover, its transition structure is a *syntactic semiring* which is a minimal semiring recognizing the language and which can be defined analogously to a syntactic monoid (see [30]). In the paper [16] there are mentioned some examples of $\mathscr{C}$-varieties of languages which can be characterized via varieties of meet automata. There is also a close connection between the notion of a canonical meet automaton and a notion of a *universal automaton* which contains all minimal non-deterministic automata of a given regular language (see [31] and [22]).

One can make one step further. As we add intersections to the representation of minimal automaton,

[4] A state is final if and only if it contains the empty word.

[5] There exist several papers which use the term ordered (deterministic, non-deterministic or two-way) automaton in a different meaning, e.g. in [35] it is required that an action by a letter is increasing but need not be isotone.

we can try to add also unions. In other words, we consider the sublattice of the lattice 2^{A^*} generated by $\mathscr{A}_L$. Since this lattice is distributive, we define an abstract notion of a *distributive lattice automata* (*DL-automata*) which are automata enriched by a distributive lattice structure, where both operations are compatible with actions by letters. Note that this model differs from lattice automata defined in [21]. We want to define varieties of *DL*-automata as a natural counterpart of generalized varieties of languages which are not required to be closed under taking any of Boolean operations. Indeed, such classes naturally occur in the theory of formal languages: for example, many classes defined by models of quantum automata are of this kind. The goal is a characterization of such classes. Note that it is also possible to extend this principle, consider the Boolean subalgebra of 2^{A^*} generated by $\mathscr{A}_L$ and define a notion of a *BA-automaton*. Before developing this theory we prefer to clarify all aspects of the theory of *DL*-automata, since there are some difficulties. For example, in the case of meet automata, since actions by letters are morphism with respect to the meet operation, actions by sets of letters are also morphisms with respect to this operation. In the case of *DL*-automata, such an extension is not valid.

At the end we could mention that one can extend the construction in at least two natural directions. First, the theory of tree languages is a field where many fundamental ideas from the theory of deterministic automata were successfully generalized. Another recent notion of biautomata (see [17] and [14]) is based on considering both-sided quotients instead of left quotients only. In both cases one can try to apply the previous constructions and consider varieties of automata (enriched by an algebraic structure). Some papers in this direction already exist [11].

Acknowledgement

I would like to express my gratitude to my colleagues Michal Kunc and Libor Polák for our numerous interesting discussions on the topic.

References

[1] J. Almeida (1994): *Finite semigroups and universal algebra*. World Scientific, Singapore, doi:`10.1142/2481`.

[2] J. Almeida (2005): *Profinite semigroups and applications*. In V.B. Kudryavtsev, I.G. Rosenberg & M. Goldstein, editors: *Structural theory of automata, semigroups, and universal algebra*, *NATO Science Series II: Mathematics, Physics and Chemistry* 207, Springer, pp. 1–45, doi:`10.1007/1-4020-3817-8_1`.

[3] M. Arfi (1987): *Polynomial Operations on Rational Languages*. In F.-J. Brandenburg, G. Vidal-Naquet & M. Wirsing, editors: *STACS*, *Lecture Notes in Computer Science* 247, Springer, pp. 198–206, doi:`10.1007/BFb0039607`.

[4] J. Brzozowski (1962): *Canonical regular expressions and minimal state graphs for definite events*. In: *Mathematical theory of Automata*, *Symposia series* 12, Research Institute, Brooklyn, pp. 529–561.

[5] J. Brzozowski & B. Li (2013): *Syntactic Complexity of R- and J-Trivial Regular Languages*. In H. Jürgensen & R. Reis, editors: *DCFS*, *Lecture Notes in Computer Science* 8031, Springer, pp. 160–171, doi:`10.1007/978-3-642-39310-5_16`.

[6] L. Chaubard, J.-É. Pin & H. Straubing (2006): *Actions, wreath products of C-varieties and concatenation product*. *Theor. Comput. Sci.* 356(1-2), pp. 73–89, doi:`10.1016/j.tcs.2006.01.039`.

[7] R. Cohen & J. Brzozowski (1971): *Dot-Depth of Star-Free Events*. *J. Comput. Syst. Sci.* 5(1), pp. 1–16, doi:`10.1016/S0022-0000(71)80003-X`.

[8] V. Diekert, P. Gastin & M. Kufleitner (2008): *A Survey on Small Fragments of First-Order Logic over Finite Words*. *Int. J. Found. Comput. Sci.* 19(3), pp. 513–548, doi:`10.1142/S0129054108005802`.

[9] S. Eilenberg (1976): *Automata, Languages and Machines, vol. B.* Academic Press.

[10] Z. Ésik & M. Ito (2003): *Temporal Logic with Cyclic Counting and the Degree of Aperiodicity of Finite Automata*. *Acta Cybern.* 16(1), pp. 1–28. Available at `http://www.inf.u-szeged.hu/actacybernetica/edb/vol16n1/Esik_2003_ActaCybernetica.xml`.

[11] Z. Ésik & S. Iván (2008): *Some Varieties of Finite Tree Automata Related to Restricted Temporal Logics*. *Fundam. Inform.* 82(1-2), pp. 79–103. Available at `http://iospress.metapress.com/content/4216mrh7r6477172/`.

[12] Z. Ésik & K.G. Larsen (2003): *Regular languages definable by Lindström quantifiers*. *RAIRO - Theoretical Informatics and Applications* 37(3), pp. 179–241, doi:`10.1051/ita:2003017`.

[13] M. Gehrke, S. Grigorieff & J.-É. Pin (2008): *Duality and Equational Theory of Regular Languages*. In L. Aceto, I. Damgård, L.A. Goldberg, M.M. Halldórsson, A. Ingólfsdóttir & I. Walukiewicz, editors: *ICALP (2)*, *Lecture Notes in Computer Science* 5126, Springer, pp. 246–257, doi:`10.1007/978-3-540-70583-3_21`.

[14] M. Holzer & S. Jakobi (2013): *Minimization and characterizations for biautomata*. In S. Bensch, F. Drewes, R. Freund & F. Otto, editors: *NCMA*, 294, Österreichische Computer Gesellschaft, pp. 179–193.

[15] H. Kamp (1968): *Tense logic and theory of linear orders*. Ph.D. thesis, University of California.

[16] O. Klíma & L. Polák (2008): *On varieties of meet automata*. *Theor. Comput. Sci.* 407(1-3), pp. 278–289, doi:`10.1016/j.tcs.2008.06.005`.

[17] O. Klíma & L. Polák (2012): *On biautomata*. *RAIRO - Theor. Inf. and Applic.* 46(4), pp. 573–592, doi:`10.1051/ita/2012014`.

[18] O. Klíma & L. Polák (2013): *Alternative Automata Characterization of Piecewise Testable Languages*. In M.-P. Béal & O. Carton, editors: *Developments in Language Theory*, *Lecture Notes in Computer Science* 7907, Springer, pp. 289–300, doi:`10.1007/978-3-642-38771-5_26`.

[19] M. Kufleitner & P. Weil (2012): *On logical hierarchies within FO^2-definable languages*. *Logical Methods in Computer Science* 8(3), doi:`10.2168/LMCS-8(3:11)2012`.

[20] M. Kunc (2003): *Equational description of pseudovarieties of homomorphisms*. *RAIRO - Theoretical Informatics and Applications* 37(3), pp. 243–254, doi:`10.1051/ita:2003018`.

[21] O. Kupferman & Y. Lustig (2007): *Lattice Automata*. In B. Cook & A. Podelski, editors: *VMCAI*, *Lecture Notes in Computer Science* 4349, Springer, pp. 199–213, doi:`10.1007/978-3-540-69738-1_14`.

[22] S. Lombardy & J. Sakarovitch (2008): *The universal automaton*. In J. Flum, E. Grädel & T. Wilke, editors: *Logic and Automata*, *Texts in Logic and Games* 2, Amsterdam University Press, pp. 457–504.

[23] R. McNaughton & S. Papert (1971): *Counter-Free Automata*. M.I.T. Press.

[24] J.-É. Pin (1995): *A Variety Theorem Without Complementation*. *Russian Mathematics* 39, pp. 80–90. Available at `http://www.liafa.jussieu.fr/~jep/publications.html`.

[25] J.-É. Pin (1997): *Syntactic semigroups*. In G. Rozenberg & A. Salomaa, editors: *Handbook of Formal Languages*, 1, Springer, pp. 679–746, doi:`10.1007/978-3-642-59136-5_10`. Available at `www.liafa.jussieu.fr/~jep/publications.html`.

[26] J.-É. Pin (2012): *Equational Descriptions of Languages*. *Int. J. Found. Comput. Sci.* 23(6), pp. 1227–1240, doi:`10.1142/S0129054112400497`.

[27] J.-É. Pin & P. Weil (1996): *A Reiterman theorem for pseudovarieties of finite first-order structures*. *Algebra Universalis* 35(4), pp. 577–595, doi:`10.1007/BF01243597`.

[28] J.-É. Pin & P. Weil (1997): *Ponynominal Closure and Unambiguous Product*. *Theory Comput. Syst.* 30(4), pp. 383–422, doi:`10.1007/BF02679467`.

[29] T. Place & M. Zeitoun (2014): *Separating Regular Languages with First-Order Logic*. *CoRR* abs/1402.3277. Available at `http://arxiv.org/abs/1402.3277`.

[30] L. Polák (2004): *A classification of rational languages by semilattice-ordered monoids*. *Archivum Mathematicum* 40(4), pp. 395–406. Available at `http://emis.muni.cz/journals/AM/04-4/index.html`.

[31] L. Polák (2005): *Minimalizations of NFA using the universal automaton*. *Int. J. Found. Comput. Sci.* 16(5), pp. 999–1010, doi:`10.1142/S0129054105003431`.

[32] J. Reiterman (1982): *The Birkhoff theorem for finite algebras*. *Algebra Universalis* 14, pp. 1–10, doi:`10.1007/BF02483902`.

[33] J. Rhodes & B. Steinberg (2009): *The q-theory of Finite Semigroups*. Monographs in Mathematics, Springer, doi:`10.1007/b104443`.

[34] M. P. Schützenberger (1965): *On Finite Monoids Having Only Trivial Subgroups*. *Information and Control* 8(2), pp. 190–194, doi:`10.1016/S0019-9958(65)90108-7`.

[35] T. Schwentick, D. Thérien & H. Vollmer (2001): *Partially-Ordered Two-Way Automata: A New Characterization of DA*. In W. Kuich, G. Rozenberg & A. Salomaa, editors: *Developments in Language Theory*, *Lecture Notes in Computer Science* 2295, Springer, pp. 239–250, doi:`10.1007/3-540-46011-X_20`.

[36] I. Simon (1975): *Piecewise testable events*. In H. Barkhage, editor: *Automata Theory and Formal Languages*, *Lecture Notes in Computer Science* 33, Springer, pp. 214–222, doi:`10.1007/3-540-07407-4_23`.

[37] J. Stern (1985): *Complexity of Some Problems from the Theory of Automata*. *Information and Control* 66(3), pp. 163–176, doi:`10.1016/S0019-9958(85)80058-9`.

[38] H. Straubing (1981): *A Generalization of the Schützenberger Product of Finite Monoids*. *Theor. Comput. Sci.* 13, pp. 137–150, doi:`10.1016/0304-3975(81)90036-0`.

[39] H. Straubing (2002): *On Logical Descriptions of Regular Languages*. In S. Rajsbaum, editor: *LATIN*, *Lecture Notes in Computer Science* 2286, Springer, pp. 528–538, doi:`10.1007/3-540-45995-2_46`.

[40] H. Straubing & P. Weil (2012): *An introduction to finite automata and their connection to logic*. In D. D'Souza & P. Shankar, editors: *Modern Applications of Automata Theory*, IISc Research Monographs Series, World Scientific, doi:`10.1142/9789814271059_0001`. Available at `http://arxiv.org/abs/1011.6491`.

[41] D. Thérien (1981): *Classification of Finite Monoids: The Language Approach*. *Theor. Comput. Sci.* 14, pp. 195–208, doi:`10.1016/0304-3975(81)90057-8`.

[42] W. Thomas (1982): *Classifying Regular Events in Symbolic Logic*. *J. Comput. Syst. Sci.* 25(3), pp. 360–376, doi:`10.1016/0022-0000(82)90016-2`.

[43] A. Trahtman (2001): *Piecewise and Local Threshold Testability of DFA*. In R. Freivalds, editor: *FCT*, *Lecture Notes in Computer Science* 2138, Springer, pp. 347–358, doi:`10.1007/3-540-44669-9_33`.

[44] P. Weil (2004): *Algebraic Recognizability of Languages*. In J. Fiala, V. Koubek & J. Kratochvíl, editors: *MFCS*, *Lecture Notes in Computer Science* 3153, Springer, pp. 149–175, doi:`10.1007/978-3-540-28629-5_8`. Available at `http://arxiv.org/abs/cs/0609110`.

Decision Problems for Deterministic Pushdown Automata on Infinite Words

Christof Löding
Lehrstuhl Informatik 7
RWTH Aachen University
Germany
loeding@cs.rwth-aachen.de

The article surveys some decidability results for DPDAs on infinite words (ω-DPDA). We summarize some recent results on the decidability of the regularity and the equivalence problem for the class of weak ω-DPDAs. Furthermore, we present some new results on the parity index problem for ω-DPDAs. For the specification of a parity condition, the states of the omega-DPDA are assigned priorities (natural numbers), and a run is accepting if the highest priority that appears infinitely often during a run is even. The basic simplification question asks whether one can determine the minimal number of priorities that are needed to accept the language of a given ω-DPDA. We provide some decidability results on variations of this question for some classes of ω-DPDAs.

1 Introduction

Finite automata, which are used as a tool in many areas of computer science, have good closure and algorithmic properties. For example, language equivalence and inclusion are decidable (see [9]), and for many subclasses of the regular languages it is decidable whether a given automaton accepts a language inside this subclass (see [19] for some results of this kind). In contrast to that, the situation for pushdown automata is much more difficult. For nondeterministic pushdown automata, many problems like language equivalence and inclusion are undecidable (see [9]), and it is undecidable whether a given nondeterministic pushdown automaton accepts a regular language. The class of languages accepted by deterministic pushdown automata forms a strict subclass of the context-free languages. While inclusion remains undecidable for this subclass, a deep result from [15] shows the decidability of the equivalence problem. Furthermore, the regularity problem for deterministic pushdown automata is also decidable [17, 20].

While automata on finite words are a very useful model, some applications, in particular in verification by model checking (see [2]), require extensions of these models to infinite words. Although the theory of finite automata on infinite words (called ω-automata in the following) usually requires more complex constructions because of the more complex acceptance conditions, many of the good properties of finite automata on finite words are preserved (see [13] for an overview). Pushdown automata on infinite words (pushdown ω-automata) have been studied because of their ability to model executions of nonterminating recursive programs. In [6] efficient algorithms for checking emptiness of Büchi pushdown automata are developed (a Büchi automaton accepts an infinite input word if it visits an accepting state infinitely often during its run). Besides these results, the algorithmic theory of pushdown ω-automata has not been investigated very much. For example, in [5] the decidability of the regularity problem for deterministic pushdown ω-automata has been posed as an open question and to our knowledge no answer to this question is known. Furthermore, it is unknown whether the equivalence of deterministic pushdown ω-automata is decidable.

Z. Ésik and Z. Fülöp (Eds.): Automata and Formal Languages 2014 (AFL 2014)
EPTCS 151, 2014, pp. 55–73, doi:10.4204/EPTCS.151.4

The first part of this article summarizes some recent partial results on the regularity and equivalence problem for deterministic pushdown ω-automata from [12].

In the second part we consider decision problems concerning the acceptance condition of the automata. One of the standard acceptance conditions of ω-automata is the parity condition (see [8] for an overview of possible acceptance conditions). Such a condition is specified by assigning priorities (natural numbers) to the states of the automaton, using even priorities for "good" states and odd priorities for the "bad" states. A run is accepting if among the states that occur infinitely often the highest priority is even. For deterministic automata (independent of the precise automaton model), one can show that more languages can be accepted if more priorities are used. So the number of priorities required for accepting a language is a measure for the complexity of the language. A natural decision problem arising from that, is the question of determining for a given deterministic parity automaton the smallest number of priorities that are needed for accepting the language of the automaton. This referred to as the parity index problem.

For finite deterministic parity automata, the minimal number of priorities required for accepting the language can be computed in polynomial time, and a corresponding automaton can be constructed by simply reassigning priorities in the allowed range to the states of the given automaton [4]. For deterministic pushdown parity automata it was shown in [10] that it is decidable whether a given automaton is equivalent to a deterministic pushdown Büchi automaton. We present here the general result that the parity index problem for deterministic pushdown parity automata is decidable. The method is based on parity games on pushdown graphs and has already been described in the PhD thesis [14].

We further consider a model of deterministic pushdown automata in which the types of the action on the pushdown store are determined by the input symbols, called visibly pushdown automata (VPA) [1]. In these automata, the input alphabet is partitioned into three sets of symbols, referred to as call, return, and internal symbols. On reading a call, the pushdown automaton has to add a symbol to the stack, on reading a return, it has to remove a symbol from the stack, and on reading an internal, it does not alter the stack. It turns out that, for a fixed partition of the input alphabet, this class of automata has good closure and algorithmic properties [1]. On finite words it is even possible to determinize such VPAs. However, it turns out that Büchi VPAs cannot, in general, be transformed into equivalent deterministic Muller or parity VPAs [1]. To resolve this problem, in [11] a variation of the parity condition has been proposed, referred to as stair parity condition. It is defined as a standard parity condition, however, it is not evaluated on the sequence of all states but only on the sequence of states that occur on steps of the run. A step is a configuration in the run such that no later configuration has a smaller stack height. In [11] it is shown that each nondeterministic Büchi VPA can be transformed into an equivalent deterministic stair parity VPA. We prove here that the stair parity index problem for deterministic VPAs can be solved in polynomial time. We also consider the question whether a given stair parity VPA is equivalent to a parity VPA (with a standard parity condition instead of a stair condition). For the particular case of stair Büchi VPAs we show that this problem is decidable.

The remainder of this paper is structured as follows. In Section 2 we introduce some basic terminology and definitions. In Section 3 we consider the regularity and equivalence problem for ω-DPDAs. Section 4 is about the parity index of parity DPDAs and stair parity DVPAs. In Section 5 we show how to decide whether the stair condition is needed for accepting the language of a given stair Büchi DVPAs. In Section 6 we give a short conclusion.

2 Preliminaries

We denote the set of natural numbers (including 0) by $\mathbb{N}$. For a set S we denote its cardinality by $|S|$. Let A be an alphabet, i.e., a finite set of symbols, then A^* is the set of finite words over A, and A^ω the set of ω-words over A, i.e., infinite sequences of A symbols indexed by the natural numbers. The subsets of A^* are called languages, and subsets of A^ω are called ω-languages. The length of a finite word $w \in A^*$ is denoted by $|w|$, and the empty word is ε. We assume the reader to be familiar with regular languages, i.e., the languages specified by regular expressions or equivalently by finite state automata (see, for example, [9] for basics on regular languages).

We are mainly concerned with deterministic pushdown automata in this work. We first define pushdown machines, which are pushdown automata without acceptance condition. We then obtain pushdown automata by adding an acceptance condition.

A *deterministic pushdown machine* $\mathscr{M} = (Q, A, \Gamma, \delta, q_0, \perp)$ consists of

- a finite state set Q and initial state $q_0 \in Q$,
- a finite input alphabet A (we abbreviate $A_\varepsilon = A \cup \{\varepsilon\}$),
- a finite stack alphabet Γ and initial stack symbol $\perp \notin \Gamma$ (let $\Gamma_\perp = \Gamma \cup \{\perp\}$),
- a partial transition function $\delta : Q \times \Gamma_\perp \times A_\varepsilon \to Q \times \Gamma_\perp^*$ such that for each $p \in Q$ and $A \in \Gamma_\perp$:
  - $\delta(p, Z, a)$ is defined for all $a \in A$ and $\delta(p, Z, \varepsilon)$ is undefined, or the other way round.
  - For each transition $\delta(p, Z, a) = (q, W)$ with $a \in A_\varepsilon$ the bottom symbol $\perp$ stays at the bottom of the stack and only there, i.e., $W \in \Gamma^* \perp$ if $Z = \perp$ and $W \in \Gamma^*$ if $Z \neq \perp$.

The set of configurations of $\mathscr{M}$ is $Q\Gamma^*\perp$ where $q_0\perp$ is the initial configuration. The stack consisting only of $\perp$ is called the empty stack. A configuration $q\sigma$ is also written (q, σ). For a given input word $w \in A^*$ or $w \in A^\omega$, a finite resp. infinite sequence $q_0\sigma_0, q_1\sigma_1, \ldots$ of configurations with $q_0\sigma_0 = q_0\perp$ is a run of w on $\mathscr{M}$ if there are $a_i \in A_\varepsilon$ with $w = a_1 a_2 \cdots$ and $\delta(q_i, Z, a_{i+1}) = (q_{i+1}, U)$ is such that $\sigma_i = ZV$ and $\sigma_{i+1} = UV$ for some stack suffix $V \in \Gamma_\perp^*$.

For finite words, we consider the model of a deterministic pushdown automaton (DPDA) $\mathscr{A} = (\mathscr{M}, F)$ consisting of a deterministic pushdown machine $\mathscr{M} = (Q, A, \Gamma, \delta, q_0, \perp)$ and a set of final states $F \subseteq Q$. It accepts a word $w \in A^*$ if w induces a run ending in a final state. These words form the language $L_*(\mathscr{A}) \subseteq A^*$. For ω-words, we consider two types of acceptance conditions, namely Büchi and parity conditions. A Büchi DPDA $\mathscr{A} = (\mathscr{M}, F)$ is specified in the same way as a DPDA on finite words. The ω-language $L_\omega(\mathscr{A})$ defined by $\mathscr{A}$ is the set of all ω-words w for which the run of $\mathscr{A}$ on w contains a state from F at infinitely many positions.

For a parity DPDA, the acceptance condition is specified by a function $\Omega : Q \to \mathbb{N}$, which assigns a number to each state, which is referred to as its priority. A run is accepting if the highest priority that occurs infinitely often is even. Note that Büchi conditions can be specified as parity conditions by assigning priority 2 to states in F and priority 1 to states outside F.

In Section 3 we consider the class of weak DPDAs. These are parity DPDAs, in which the transitions can never lead from one state q to another state q' with a smaller priority. Hence, in a run of a weak DPDA the sequence of priorities is monotonically increasing, which implies that the sequence is ultimately constant. It follows that each weak DPDA is equivalent to the Büchi DPDA that uses the set of states with even priority as set of final states. We therefore also use term weak Büchi DPDAs to emphasize that it is a subclass of Büchi DPDAs.

In general, we refer to DPDAs on infinite words as ω-DPDAs if we do not explicitly specify the type of acceptance. For simplicity, we assume that infinite sequences of ε-transitions are not possible in ω-DPDAs. Such sequences can be eliminated by redirecting certain ε-transitions into corresponding sink states (the acceptance status of such a state would depend on the exact semantics one uses for runs that end in an infinite ε-sequence). It is sufficient to compute the pairs (q,Z) of states q and top stack symbols Z such that there is a run of ε-transitions leading from $qZ\bot$ to some configuration of the form $qZWZ\bot$, such that the Z at the bottom of the stack is never removed during this run. These pairs can be computed efficiently (see [6]), and it is not difficult to see that redirecting the ε-transitions from these pairs (q,Z) is sufficient for eliminating all infinite ε-sequences.

We also consider the model of deterministic visibly pushdown automata (DVPA) [1]. These automata are defined with respect to a partitioned alphabet $A = A_c \cup A_i \cup A_r$, where A_c contains all letters that can only occur in transitions pushing some symbol onto the stack (call symbols), A_r those forcing the automaton to pop a symbol from the stack (return symbols), and A_i those leaving the stack unchanged (internal symbols). Furthermore, DVPAs do not have ε-transitions. We also adopt the general convention that VPAs do not consider the top-most stack symbol in their transitions. This simplifies several arguments. We can make this assumption without loss of generality, because it is possible to always keep track of the top-most stack symbol in the control state.

Formally, a deterministic visibly pushdown machine over the partitioned alphabet $A = A_c \cup A_i \cup A_r$ is of the form $\mathscr{M} = (Q, A, \Gamma, \delta, q_0, \bot)$, where δ consists of three transition functions

$$\begin{aligned}\delta_c &: Q \times A_c \to Q \times \Gamma\\ \delta_r &: Q \times \Gamma \times A_r \to Q\\ \delta_i &: Q \times A_i \to Q\end{aligned}$$

Instead of defining the semantics of these transitions directly, we simply describe how the corresponding transitions in a standard DPDA would look like. A call transition $\delta_c(q,c) = (p,Z)$ corresponds to a set of transitions $\delta(q,Y,c) = (p,ZY)$ for each $Y \in \Gamma_\bot$. A return transition $\delta_r(q,Z,r) = p$ corresponds to the transition $\delta_r(q,Z,r) = (p,\varepsilon)$, and an internal transition $\delta_i(q,i) = p$ to a set of transitions $\delta(q,Y,i) = (p,Y)$ for each $Y \in \Gamma_\bot$. Note that this definition does not admit transitions for return symbols on the empty stack. In [1] such transitions are possible, but we prefer to use the simpler model here to ease the presentation.

By adding an acceptance condition, we obtain DVPAs as in the general case. As for ω-DPDAs, we are interested in ω-DVPAs with Büchi or parity condition. However, we also consider a variant of the parity condition referred to as stair parity condition [11]. The condition is specified in the same way as before, however, it is evaluated only on a subsequence of the run, namely on the sequence of steps, as defined below.

A configuration $q\sigma$ in a run of a DVPA $\mathscr{A}$ is called a step if the stack height of all configurations $q'\sigma'$ that come later in the run is bigger than the stack height of $q\sigma$, i.e., $|\sigma| \leq |\sigma'|$. Note that the positions of the steps do not depend on the automaton, but only on the input word, because the type of the stack operation is determined for each input symbol. We can now define stair visibly pushdown automata. The only difference to visibly pushdown automata is that they evaluate the acceptance condition only for the subsequence of the run containing consisting of the steps.

In other words, a stair parity DVPA has the same components as a parity DVPA. An input is accepted if in the run on this input the maximal priority that occurs infinitely often on a step is even. In the same way we obtain stair Büchi DVPAs, which accept if an accepting state occurs on infinitely many steps.

We end this section by introducing some more terminology for visibly pushdown automata that is used in Sections 4 and 5.

The set of well matched words over $A = A_c \cup A_i \cup A_r$ is, intuitively speaking, the set of well-balanced words in which for each position with a call symbol there is a later position at which this call is "closed" by some return symbol (and vice versa, each return position has a corresponding previous call position). Formally, the set is defined inductively as follows:

- Each $a \in A_i$ is a well matched word.
- If u and v are well-matched words, then uv is a well matched word.
- If w is a well matched word, then cwr is a well-matched word for each $c \in A_c$ and each $r \in A_r$.

The words that are created by the last rule are referred to as minimally well-matched words. Let L_{mwm} denote this set, i.e., the words of the form cwr with a call c, a return r, and a well-matched word w.

The canonical language that can be accepted by a stair Büchi DVPA but by no parity DVPA is the language L_{su} of strictly unbounded words, containing all words over $\langle \{c\}, \emptyset, \{r\} \rangle$ with an infinite number of unmatched calls. More formally, an infinite word is in L_{su} if it is of the form $w_1 c w_2 c w_3 c \cdots$ for well-matched words w_i. In [1] it is shown that L_{su} cannot be accepted by a parity DVPA. But it is easy to construct a stair Büchi DVPA $\mathscr{A}$ for L_{su} using only a single stack symbol and one accepting and one non-accepting state (see [11]), where $\mathscr{A}$ moves into the accepting state for each c, and into the non-accepting state for each r. Note that the position after reading a c is a step in the run iff this c does not have a matching return. Thus, there are infinitely many unmatched calls iff there are infinitely many accepting states on steps.

3 Regularity and Equivalence

In this section we summarize results from [12] that show how to solve the regularity problem and the equivalence problem for weak ω-DPDAs. The proof uses a reduction to the corresponding problems for DPDAs on finite words. More details on these results can be found in [12] and in [14].

The regularity problem for DPDA is the problem of deciding for a given DPDA whether it accepts a regular language. It has been shown to be decidable in [17] and the complexity has been improved in [20].

Theorem 1 ([17]). *The regularity problem for DPDAs is decidable.*

The rough idea of the proof is as follows. Assuming that the language of the given DPDA is regular, one shows that for each configuration above a certain height (depending on the size of the DPDA), there is an equivalent configuration of smaller height. A finite state machine can then be constructed by redirecting the transitions into higher configurations to their equivalent smaller counterparts. Here, two configurations are considered to be equivalent if they define the same language when considered as initial configuration of the DPDA. The decision method for the regularity problem is then based on the characterization of the regular languages in terms of the Myhill/Nerode equivalence. For a language $L \subseteq A^*$, the Myhill/Nerode equivalence is defined as follows for words $u, v \in A^*$:

$$u \sim_L v \text{ iff } \forall w \in A^* : uw \in L \Leftrightarrow vw \in L.$$

A language of finite words is regular if, and only if, it has finitely many Myhill/Nerode equivalence classes, and these classes can be used as states for a canonical finite automaton for the language.

Unfortunately, a corresponding result is not true for ω-regular languages, in general. However, the subclass of weak ω-regular languages possesses a similar characterization in terms of an equivalence [16]. This similarity raises the question whether the decidability results for DPDAs on finite words can be lifted to weak DPDAs on infinite words.

In [12] it is shown that this is indeed possible. In fact, it is even possible to reduce questions for weak ω-DPDAs to DPDAs on finite words. To establish such a connection, we associate a language $L_*(\mathscr{A})$ of finite words to a weak ω-DPDA $\mathscr{A}$, which is obtained by viewing $\mathscr{A}$ as a DPDA on finite words and taking the set of states with an even priority as the set of final states.

The first attempt for reducing the regularity problem for weak ω-DPDAs to the regularity problem for DPDAs would be to test $L_*(\mathscr{A})$ for regularity, where $\mathscr{A}$ is the given weak ω-DPDA. This approach is sound because regularity of $L_*(\mathscr{A})$ implies ω-regularity of $L_\omega(\mathscr{A})$: a finite deterministic automaton for $L_*(\mathscr{A})$ viewed as a Büchi automaton defines $L_\omega(\mathscr{A})$ because it visits final states at the same positions as $\mathscr{A}$.

That the approach is not complete is illustrated by the following simple example. Consider the alphabet $\{a,b\}$ and the ω-language a^*b^ω of words starting with a finite sequence of a followed by an infinite sequence of b. Obviously, this language is regular. A weak ω-DPDA $\mathscr{A}$ could proceed as follows to accept this language. It starts by pushing a symbol onto the stack for each a. When the first b comes in the input, it changes its state and starts popping the stack symbols again. Once the bottom of the stack is reached, it changes to an accepting state and remains there as long as it reads further b (if another a comes, then the input is rejected). Since the finite a-sequence is followed by infinitely many b, it is guaranteed that $\mathscr{A}$ reaches the accepting state if the input is from a^*b^ω. Note that this is a weak ω-DPDA because it can change once from non-accepting to accepting states, and once more back to non-accepting states. The language $L_*(\mathscr{A})$ of this weak ω-DPDA is the set of all finite words of the form $a^m b^n$ with $n \geq m$ because $\mathscr{A}$ reaches the accepting state only after it has read as many b as a. Thus, $L_*(\mathscr{A})$ is non-regular although $L_\omega(\mathscr{A})$ is.

For this example, the problem would be solved if $\mathscr{A}$ switches to an accepting state as soon as the first b is read (instead of deferring this change to the stack bottom). In general, one can show that each weak ω-DPDA can be transformed in such a way that the above reduction to the regularity test for $L_*(\mathscr{A})$, as shown be the following theorem.

Theorem 2 ([12])**.** *There is a normal form for weak ω-DPDAs with the following properties:*

1. *For a weak ω-DPDA $\mathscr{A}$ in normal form, the language $L_\omega(\mathscr{A})$ is ω-regular if, and only if, $L_*(\mathscr{A})$ is regular.*
2. *Given two weak ω-DPDAs $\mathscr{A}$ and $\mathscr{B}$ in normal form, $L_\omega(\mathscr{A}) = L_\omega(\mathscr{B})$ if, and only if, $L_*(\mathscr{A}) = L_*(\mathscr{B})$.*

Combining the first part of Theorem 2 with Theorem 1, we get the decidability of the regularity problem for weak ω-DPDAs.

Corollary 1 ([12])**.** *The regularity problem for weak ω-DPDAs is decidable.*

The second part of the theorem can be used to show the decidability of the equivalence problem for weak ω-DPDAs, based on the corresponding deep result for DPDAs.

Theorem 3 ([15])**.** *The equivalence problem for DPDAs is decidable.*

Corollary 2 ([12])**.** *The equivalence problem for weak ω-DPDAs is decidable.*

The two problems for the full class of ω-DPDAs remain open. In [14] a congruence for ω-languages is identified that characterizes regularity within the class of ω-DPDA recognizable languages (a language accepted by an ω-DPDA is regular if, and only if, this congruence has finitely many equivalence classes). This might be step towards a solution for the regularity problem. However, the decidability of characterizing criterion remains open.

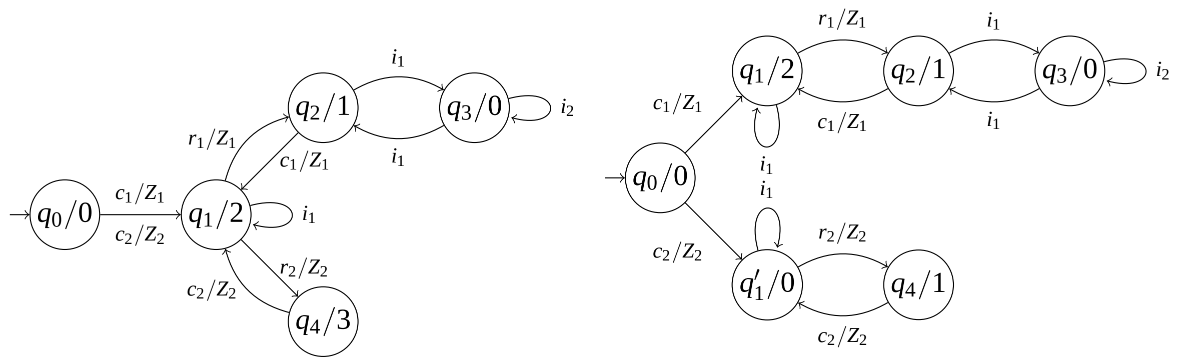


Figure 1: On the left-hand side: DVPA with minimal number of priorities for the given transition structure; on the right-hand side: equivalent DVPA with less priorities

4 The Parity Index Problem

In this section we are interested in the problem of reducing the number of priorities used in a parity condition. Formally, we consider the following problem. Given a parity DPDA (or stair parity DVPA) $\mathscr{A}$, compute the smallest number of priorities required for accepting $L_\omega(\mathscr{A})$ with a parity DPDA (or stair parity DVPA). We refer to these two variants of the problem as the parity index problem for DPDAs, and the stair parity index problem for stair parity DVPAs.

For finite parity automata, it suffices to change the priority assignment, in order to obtain an equivalent automaton with the fewest number of priorities, and this modified priority function can be computed in polynomial time [4].

For parity DPDAs the situation is different, as illustrated by the example in Figure 1 (taken from [18]). We use a DVPA in the example, where c_1, c_2 are calls, r_1, r_2 are returns, i_1, i_2 are internals, and Z_1, Z_2 are stack symbols. The transitions on call symbols are annotated with the stack symbol to be pushed, and for the return symbols with the stack symbol to be popped. The priority function of the DVPA on the left-hand side of Figure 1 (indicated as labels of the states) is minimal for the state set and the transition structure. The problem is caused by the state q_1, which is part of the loop in the upper and the lower branch. However, there is no run of the automaton that traverses both the upper and the lower branch. If the first symbol in the input is c_1, then the automaton stores Z_1 on the stack. Whenever the automaton reaches q_1 in the future, Z_1 will be on top of the stack and the automaton can only use the top branch. For the lower branch and c_2 as the first input symbol the situation is similar.

Splitting q_1 into two copies as done in the DVPA on the right-hand side of the figure, makes it possible to reassign priorities without using priority 3.

The example illustrates that we need to take a different approach for computing the parity index of pushdown automata. This approach is also described in [14].

Let $P \subset \mathbb{N}$ be a finite set of priorities. A parity DPDA using only priorities from P is referred to as a P-parity DPDA. To decide whether a given parity DPDA $\mathscr{A}$ has an equivalent P-parity DPDA, consider the following game. There are two players, referred to as Automaton and Classifier. Automaton starts in the initial configuration of $\mathscr{A}$ and plays transitions of $\mathscr{A}$. After each move of Automaton, Classifier chooses one priority from P. The idea is that the classifier wants to prove that there is a P-parity DPDA that accepts $L_\omega(\mathscr{A})$. If Classifier chooses priority k in a move, this can be interpreted as "the parity DPDA that I have in mind would now be in a state with priority k".

This game can be formalized as a game over a pushdown graph (basically, the configuration graph of $\mathscr{A}$ enriched by the bounded number of choices for Classifier). The winning condition states that an infinite play is won by classifier if, and only if, the two priority sequences, one induced by the configurations chosen by Automaton, the other given by the choices of Classifier, are either both accepting or both rejecting. We refer to this game as the classification game for $\mathscr{A}$ and P. The following result can be shown based on results for computing winning strategies in pushdown games [22].

Lemma 1. *Classifier has a winning strategy in the classification game for $\mathscr{A}$ and P if, and only if, there is P-parity DPDA accepting $L_\omega(\mathscr{A})$.*

For the proof it suffices to observe the following things. If there is a P-parity DPDA $\mathscr{B}$ accepting $L_\omega(\mathscr{A})$, then Classifier can simulate the run of $\mathscr{B}$ on the inputs played by Automaton, and always choose the priority of the current state of $\mathscr{B}$. This obviously defines a winning strategy because $\mathscr{A}$ and $\mathscr{B}$ accept the same language. For the other direction one uses the fact that a winning strategy for Classifier can be implemented by a pushdown automaton that reads the moves of Automaton and outputs the moves of Classifier [22, 7]. This pushdown automaton for the strategy can easily be converted into P-parity DPDA for $L_\omega(\mathscr{A})$.

For a given parity DPDA there are only finitely many sets P with less priorities than $\mathscr{A}$ uses. Since it is decidable which player has a winning strategy in the classification game [22], we obtain an algorithm for solving the parity index problem for DPDAs.

Theorem 4. *There is an algorithm solving the parity index problem for parity DPDAs.*

Stair Parity Index

We now turn to the stair parity index problem for stair parity DVPAs. In fact, it is possible to use the same game-based approach because pushdown games with stair conditions can be solved algorithmically [11]. However, for stair parity VPAs one can also adapt the much simpler solution for computing the parity index of finite parity automata. Note that in the example from Figure 1 the "critical" state q_1 can never occur on a step (moving out of q_1 requires to read a return and thus to pop a symbol). Thus, the priority of q_1 is not important in a stair parity acceptance condition. It turns out that this is not a coincidence. The result presented below has been obtained in collaboration with Philipp Stephan, see [18].

Consider the transformation graph of a stair parity DVPA $\mathscr{A}$ defined as follows. The vertices are the states of $\mathscr{A}$. An edge from q_1 to q_2 indicates that q_1 and q_2 can occur on successive steps in a run of $\mathscr{A}$. An input connecting two successive steps of a run is either an internal symbol or a minimally well-matched word. Therefore, this transformation graph can be computed inductively based on the definition of well-matched words from Section 2. One starts with the graph containing only the edges for the internal symbols. In each iteration one computes the transitive closure of the current graph. Denote this transitive closure by T. Then one checks whether there are transitions $\delta(q,c) = (q',Z)$ and $\delta(p',r,Z) = p$ for a call c, a return r, and a stack symbol Z, such that $(q',p') \in T$. In this case we add the edge (q,p) to the graph. We repeat this procedure until no more edges are added.

The paths through the transformation graph correspond to the possible sequences of states on steps in runs of $\mathscr{A}$. We now use the algorithm from [4] to compute the minimal number of priorities required on this transformation graph, simply by viewing it as the transition graph of a finite state deterministic parity automaton. The resulting assignment of priorities is then also minimal for the stair parity DVPA $\mathscr{A}$.

Theorem 5. *The stair parity index problem for stair parity DVPAs can be solved in polynomial time.*

5 Removing the Stair Condition

The goal is to decide for a given stair parity DVPA whether there is an equivalent parity DVPA and to construct one if it exists. We show how to decide this problem in general for stair Büchi DVPAs. We comment on the full class of stair parity DVPAs at the end of this section.

In Section 2 we described the language L_{su} of strictly unbounded words over $\langle\{c\},\emptyset,\{r\}\rangle$, containing all words with an infinite number of unmatched calls. This language can be accepted by a stair Büchi DVPA but not by a parity DVPA [1]. We show that a language L accepted by a stair Büchi DVPA can

- either be accepted by a parity DVPA, or
- L is at least as complex as L_{su}.

To formalize the notion of "as complex as L_{su}", we need to introduce some terminology and results concerning the topological complexity of ω-languages.

We can view A^ω as a topological space by equipping it with the Cantor topology, where the open sets are those of the form LA^ω for $L \subseteq A^*$. Starting from the open sets one defines the finite Borel hierarchy as a sequence $\Sigma_1, \Pi_1, \Sigma_2, \Pi_2, \ldots$ of classes of ω-languages as follows (we omit the finite and only refer to this hierarchy as Borel hierarchy in the following):

- Σ_1 consists of the open sets.
- Π_i consists of the complements of the languages in Σ_i.
- Σ_{i+1} consists of countable unions of languages in Π_i.

If we denote by $B(\Sigma_i)$ the closure of Σ_i under finite Boolean combinations, then we obtain the following relation between the classes of the Borel hierarchy, where an arrow indicates strict inclusion of the corresponding classes:

$$\begin{tikzcd}
\Sigma_1 \arrow[dr] & & \Sigma_2 \arrow[dr] & & \Sigma_3 \arrow[dr] & & {} & \\
& B(\Sigma_1) \arrow[ur] \arrow[dr] & & B(\Sigma_2) \arrow[ur] \arrow[dr] & & B(\Sigma_3) \arrow[ur] \arrow[dr] & & \cdots \\
\Pi_1 \arrow[ur] & & \Pi_2 \arrow[ur] & & \Pi_3 \arrow[ur] & & {} &
\end{tikzcd}$$

The above statement of a language L being at least as complex as L_{su} refers to the topological complexity. It is known that languages accepted by deterministic automata (independent of the specific automaton model) with a parity condition are included in $B(\Sigma_2)$, and in [11] it is shown that languages accepted by stair parity DVPAS are in $B(\Sigma_3)$. Furthermore, it is known that L_{su} is a true Σ_3-set (it is complete for Σ_3 for the reduction notion introduced below) [3]. In particular, it is not contained in $B(\Sigma_2)$.

In our decidability proof we show that specific patterns in a stair parity DVPA induce a high topological complexity of the accepted language (namely being at least as complex as L_{su}). On the other hand side, the absence of these patterns allows for the construction of an equivalent parity DVPA.

Before we introduce these patterns, we define the reducibility notion. Originally, it is defined using continuous functions. For our purposes it is easier to work with a different definition based on the Wadge game [21] (see also [3]).

Consider two alphabets A_1, A_2 and let $L_1 \subseteq A_1^\omega$ and $L_2 \subseteq A_2^\omega$. The Wadge game $W(L_1, L_2)$ is played between Players I and II as follows. In each round Player I plays an element of A_1 and Player II replies with a finite word from A_2^* (the empty word is also possible). In the limit, Player I plays an infinite word x over A_1, and Player II a finite or infinite word y over A_2. Player II wins if y is infinite and $x \in L_1$ iff $y \in L_2$.

We write $L_1 \leq_{\mathrm{W}} L_2$ if Player II has a winning strategy in $W(L_1, L_2)$. The following theorem is a consequence of basic properties of $\leq_{\mathrm{W}}$.

Theorem 6 ([21]). *If $L_1 \leq_W L_2$, then each class of the Borel hierarchy that contains L_2 also contains L_1.*

We use the following consequence of Theorem 6 and the properties of L_{su}.

Lemma 2. *If $L_{su} \leq_W L$, then L cannot be accepted by a parity DVPA.*

Proof. As mentioned above, the languages that can be accepted by parity DPDAs are contained in $B(\Sigma_2)$. We sketch the proof of this folklore result for completeness: We apply Theorem 6 using the following argument. Let $\mathscr{A}$ be a parity DPDA and let P be the set of priorities used by $\mathscr{A}$. Let $L_P \subseteq P^\omega$ be the sequences of priorities that satisfy the parity condition. Then $L_\omega(\mathscr{A}) \leq_W L_P$ because in the Wadge game Player II can simply keep track of the run of $\mathscr{A}$ on the word played by Player I, and play the corresponding priorities of the states of $\mathscr{A}$. Then clearly the word played by I is in $L_\omega(\mathscr{A})$ iff the priority sequence of II satisfies the parity condition. Now, L_P is easily seen to be a Boolean combination of Σ_2-sets.

Since L_{su} is not contained in $B(\Sigma_2)$ [3], we conclude from Theorem 6 that $L_{su} \leq_W L$ implies that L cannot be accepted by a parity DVPA. □

Forbidden patterns. Fix a stair Büchi DVPA $\mathscr{A} = (Q, A, \Gamma, q_0, \delta, F)$ and let $L = L_\omega(\mathscr{A})$. Recall that L does not contain words with unmatched returns. We assume that all states of $\mathscr{A}$ are reachable.

For an input word u, states q, q', and stack contents σ, σ' we write $(q,\sigma) \xrightarrow{u} (q',\sigma')$ if there is a run for the input u from (q,σ) to (q',σ'). The notation $(q,\sigma) \xrightarrow[F]{u} (q',\sigma')$ means that at least one state from F occurs on a step in this run (for steps to be defined we assume that all prefixes of u are of non-negative stack height). Dual to that we write $(q,\sigma) \xrightarrow[\notin F]{u} (q',\sigma')$ to indicate that no state from F occurs on a step in this run. If we omit the input word u then this means that there exists some input word.

It is not difficult to see that $L_{su} \leq_W L$ if there are words u and u', a stack content σ, and a state $q \in Q \setminus F$ such that

$$(q,\perp) \xrightarrow[F]{u} (q,\sigma) \xrightarrow{u'} (q,\perp)$$

and no final state occurs on steps in this run (in a run that starts and ends in the empty stack, the steps are the configurations with empty stack). To prove $L_{su} \leq_W L$, the corresponding winning strategy for Player II in the Wadge game is: $c \mapsto u$ and $r \mapsto u'$.

Unfortunately, the above condition is not necessary for $L_{su} \leq_W L$. Consider the stair Büchi DVPA $\mathscr{A}$ shown in Figure 2 with one call symbol c and two return symbols r_1, r_2 (the initial state does not matter). In this automaton the simple pattern described above cannot occur because the only non-final states are q and q'. For these two states, words u and u' as required in the pattern cannot exist for the following reasons:

- The state q can only be reached via calls and therefore $(q,\perp)$ is not reachable from $(q,\perp)$.
- From q' the symbol Z' is pushed onto the stack. But q' can only be reached on popping Z. Therefore $(q',\perp)$ is not reachable from $(q',\perp)$.

However, the example automaton $\mathscr{A}$ contains an extended pattern that guarantees that $L_{su} \leq_W L_\omega(\mathscr{A})$, as defined below and illustrated in Figure 3.

Formally, we call $q, q' \in Q \setminus F$, $q'' \in Q$, $u, v, w, x, y, z \in A^*$, and $\sigma, \sigma' \in \Gamma^*$ a forbidden pattern of $\mathscr{A}$ if $uvwxyz \in L_{mwm}$ and

$$\begin{aligned} &(q,\perp) \xrightarrow[F]{u} (q,\sigma), \quad &&(q,\perp) \xrightarrow[\notin F]{v} (q',\perp), \quad &&(q',\perp) \xrightarrow[\notin F]{w} (q,\sigma'), \\ &(q,\perp) \xrightarrow{x} (q'',\perp), \quad &&(q'',\sigma') \xrightarrow{y} (q'',\perp), \quad &&(q'',\sigma) \xrightarrow{z} (q',\perp). \end{aligned}$$

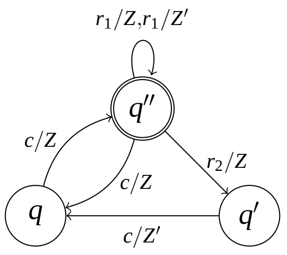


Figure 2: A stair Büchi DVPA illustrating the definition of forbidden pattern

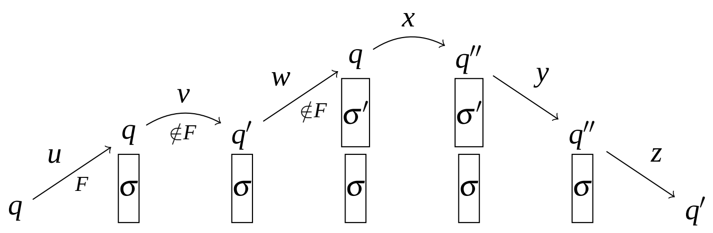


Figure 3: Forbidden pattern

Note that σ' might be empty. Since q is a non-final state, and we require that a final state is seen on a step on the path from q to q, the stack content σ cannot be empty. Further note that this pattern subsumes the first simple pattern: choose $q = q' = q''$, $v = w = x = y = \perp$, and $u' = z$.

The example automaton from Figure 2 contains such a pattern for q, q', q''. the words $u = cc$, $v = cr_2$, $w = c$, $x = cr_1$, $y = r_1$, $z = r_1r_2$, and the stack contents $\sigma = ZZ$, $\sigma' = Z'$.

Lemma 3. *If $\mathscr{A}$ has a forbidden pattern, then $L_{\mathrm{su}} \leq_{\mathrm{W}} L_\omega(\mathscr{A})$.*

Proof. We describe a winning strategy f for Player II in the Wadge game. The basic idea is to play u whenever Player I plays c, and to match the last open u with z whenever Player I plays r. However, after playing z, the automaton $\mathscr{A}$ is in state q' (compare Figure 3). Hence, to play u again, we first have to play w to reach q, producing a σ' on the stack. Therefore, it can happen that we first have remove these σ' from the stack before we can match the last open u with z. To keep track of this, we use words over $\{0,1\}$ as memory for f representing an abstraction of the stack of $\mathscr{A}$ (0 corresponds to σ and 1 corresponds to σ').

To simplify the description of f, we construct the moves such that $\mathscr{A}$ is always in q' after reading a finite word generated by f. We also assume that q' is the initial state of $\mathscr{A}$. If this is not the case, Player II can simply prepend to the first move a word leading $\mathscr{A}$ to state q'.

Let $\eta \in \{0,1\}^*$ be the current memory content (the initial content being ε). Then the strategy f works as follows:

- If Player I plays c, then play wuv and update the memory to 01η.
- If Player I plays r, then let $i \geq 0$ be such that η is of the form $1^i0\eta'$. In this case, play $wxyy^iz$ and update the memory to η'.

Let $|\eta|_0$ denote the number of 0 occurring in η and let k be the number of final states seen on steps in the run $(q,\perp) \xrightarrow{u} (q',\sigma)$. Note that $k \geq 1$ by definition of forbidden pattern. By induction one shows that

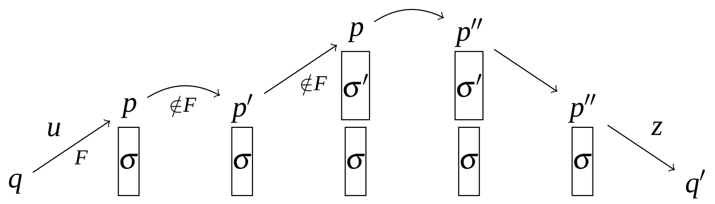


Figure 4: The relation $(p,p') \prec (q,q')$

1. after each move of Player II the number of open calls in the word played by Player I corresponds to $|\eta|_0$,
2. the number of final states seen on steps when $\mathscr{A}$ reads a finite word produced by f is $k \cdot |\eta|_0$.

This implies that $\mathscr{A}$ accepts the infinite word produced by Player II according to f iff the infinite word produced by Player I contains an unbounded number of unmatched calls. ☐

Complexity of state pairs. We now show that the absence of forbidden patterns allows to construct a parity DVPA $\mathscr{A}'$ that is equivalent to $\mathscr{A}$. In order to find an upper bound on the number of required priorities, we start by defining a measure for the complexity of pairs of non-final states. The pair (q,q') from Figure 3 would be of infinite complexity. If we now replace the states q and q' in the upper part of Figure 3 by states p and p', then this indicates that the possible runs between q and q' are at least as complex as those between p and p'. This situation is shown in Figure 4. Since q'' is just an auxiliary state and not of particular importance, we replaced it by p'' to obtain a more consistent naming scheme. We show that this relation indeed defines a strict partial order on pairs of non-final states in the case that $\mathscr{A}$ does not contain forbidden patterns.

For $p,p',q,q' \in Q \setminus F$ define $(p,p') \prec (q,q')$ iff there exists $p'' \in Q$ and stack contents σ,σ' such that (see Figure 4 for an illustration):

$$\begin{aligned}
&(q,\bot) \xrightarrow[F]{u} (p,\sigma), \quad && (p,\bot) \xrightarrow[\notin F]{} (p',\bot), \quad && (p',\bot) \xrightarrow[\notin F]{} (p,\sigma'),\\
&(p,\bot) \rightarrow (p'',\bot), && (p'',\sigma') \rightarrow (p'',\bot), && (p'',\sigma) \xrightarrow{z} (q',\bot),
\end{aligned}$$

and $uz \in L_{\text{mwm}}$. The words v,w,x,y from the definition of forbidden pattern are not made explicit in this definition because we never need to refer to them. As for forbidden patterns, σ' might be empty but σ must be non-empty.

Lemma 4. *If $\mathscr{A}$ does not have a forbidden pattern, then $\prec$ is a strict partial order on pairs of states.*

Proof. We have to show that $\prec$ is transitive and irreflexive (asymmetry follows from these two). The relation is obviously irreflexive because of the absence of forbidden patterns. Transitivity is illustrated in Figure 5 for $(r,r') \prec (p,p') \prec (q,q')$ (the stack contents are omitted). The shown pattern is obtained from $(r,r') \prec (p,p') \prec (q,q')$. The configurations with a frame lead to a pattern witnessing $(r,r') \prec (q,q')$. ☐

For $\mathscr{A}$ without forbidden patterns, we assign to each pair of states a number according to its height in the partial order, i.e., $ht : Q^2 \rightarrow \mathbb{N}$ is a mapping satisfying

$$ht(q,q') = \max(\{0\} \cup \{ht(p,p') \mid (p,p') \prec (q,q')\}) + 1.$$

We need the following simple observation.

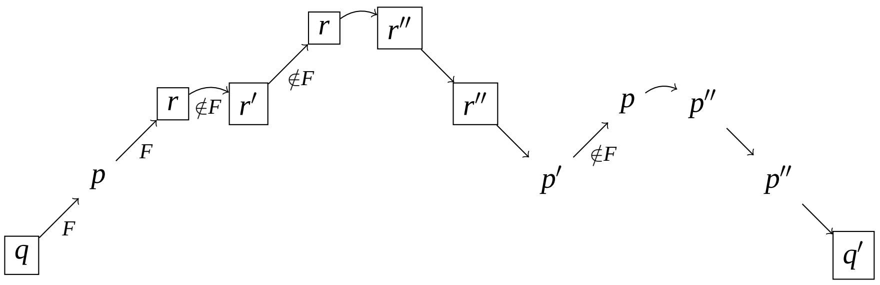


Figure 5: Transitivity of $\prec$

Lemma 5. *Let $q_1, q_1', q_2, q_2' \in Q \setminus F$. If there is a stack content σ such that $(q_2, \perp) \xrightarrow{u} (q_1, \sigma)$ and $(q_1', \sigma) \xrightarrow{v} (q_2', \perp)$ with $uv \in L_{\text{mwm}}$, then $ht(q_2, q_2') \geq ht(q_1, q_1')$.*

Proof. The condition $(q_2, \perp) \xrightarrow{u} (q_1, \sigma)$ and $(q_1', \sigma) \xrightarrow{v} (q_2', \perp)$ with $uv \in L_{\text{mwm}}$ implies that whenever $(q, q') \prec (q_1, q_1')$, then also $(q, q') \prec (q_2, q_2')$. Thus, $ht(q_2, q_2') \geq ht(q_1, q_1')$ by definition of ht. □

To make use of $\prec$ and ht in the construction of $\mathscr{A}'$ we need the following lemma. Note that this statement does not assume that $\mathscr{A}$ as no forbidden patterns.

Lemma 6. *The relation $\prec \subseteq (Q \setminus F)^2$ can be computed in time polynomial in the size of $\mathscr{A}$.*

Proof. In [6] it is shown that for a given configuration $p\sigma$ of $\mathscr{A}$ one can compute in polynomial time the set $pre^*(q\sigma)$ of configurations from which there is a run to $p\sigma$, and the set $post^*(q\sigma)$ of configurations that are reachable from $p\sigma$ by a run. These sets of configurations are sets of words over Γ, starting with a symbol from Q, and can be represented by finite automata.

The algorithms from [6] can be modified to consider only runs that either see a final state on a step or do not see a final state on a step, resulting in the sets $pre^*_F(q\sigma)$, $pre^*_{\notin F}(q\sigma)$, and similarly for *post*.

For checking whether $(p, p') \prec (q, q')$ it is sufficient to check for each p'' if there are runs as required in the definition of $\prec$. This can be done by a suitable combination of the above mentioned algorithms. For example, the stack content σ would be obtained by finding a σ such that $p\sigma \in post^*_F(q\perp)$, and $p''\sigma \in pre^*(q'\perp)$. Similarly for σ'.

All these computations can be done in polynomial time, and there are only polynomially many combinations of states that have to tested. □

Informal description of the parity DVPA. In a Büchi stair condition, a final state visited in a run is "erased" (in the sense that it is not considered for acceptance), if it is not on a step. If we construct a parity DVPA, then we cannot erase states like this. Instead, we use the mechanisms of different priorities to simulate erasing a state. Roughly, final states of the stair Büchi automaton are translated into even priorities. If a final state is erased, then this is compensated by visiting a higher odd priority. For the choice of the correct priorities we use the function ht.

In the description below, we use the terminology of "$\mathscr{A}$ closing a pair (q, q') of states". This means that $\mathscr{A}$ was in state q at some position and after reading a word L_{mwm} it reached state q', i.e., $\mathscr{A}$ was in state q before reading a call and reached q' after the matching return.

As mentioned above, we somehow need to determine a priority for the final states that are visited. Assume that the automaton is in configuration (q, β) and reads a word that increases the stack height

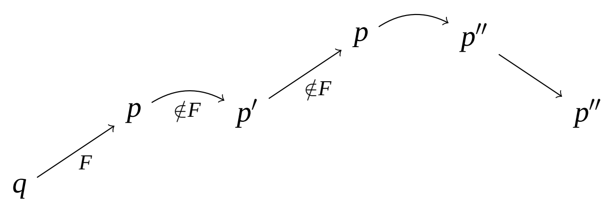


Figure 6: The pattern for determining the priority of the states with $ht(p,p') = i$

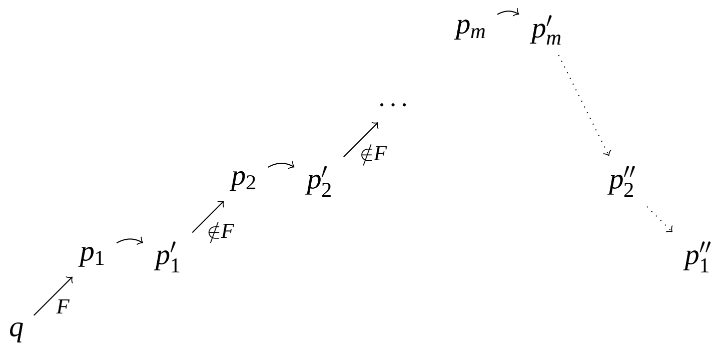


Figure 7: Detecting that each pair with q is of height at least i.

leading to some configuration $(p, \sigma\beta)$ and visiting some final states on steps during this run. We do not know if these final states remain on steps or will be erased at some point. But if we knew, e.g., that whenever we come back to the stack content β with, say, state q', that the pair (q,q') is of height at least i, then we could signal priority $2i$ for the final states that we have seen after (q,β) and signal priority $2i+1$ if we indeed close a pair (q,q') on the level of β, and thus erasing all the final states.

Assume that we have already seen the pattern shown in Figure 6, where (p,p') is a pair of height $i-1$. Then $ht(q,q') \geq i$ for every state q' that we could reach when coming back to the stack height of the configuration with q at the beginning of this pattern. In particular, if h is the maximal height of a pair of states, and (p,p') are of height h, then we know that the final states between q and p cannot all be deleted because this would require closing a pair of height $h+1$.

By a simple combinatorial argument, one can see that such a pattern as shown in Figure 6 must occur if $\mathscr{A}$, before returning to the stack height of q, has successively closed $m := |Q|^3 + 1$ pairs $(p_1.p'_1), \dots, (p_m.p'_m)$ of height $i-1$ without visiting final states on steps in between, as illustrated in Figure 7 (in the picture the pairs are closed on increasing stack levels, however, they can also be on the same stack level). If we denote by p''_i the states of $\mathscr{A}$ the next time it reaches the stack level of (p_i, p'_i) (indicated by the dotted line in the picture), then one such triple of states must occur twice, giving rise to a pattern witnessing that $ht(q,q') \geq i$.

To detect such situations, $\mathscr{A}'$ maintains a counter with range from 0 to m for each possible height of state pairs, and roughly behaves as follows:

- Whenever a pair of height i is closed by $\mathscr{A}$, then counter i is increased by one (and for technical reasons counter number 0 is increased whenever $\mathscr{A}$ visits a non-final state after reading a call or an internal symbol). To detect the closed pairs, $\mathscr{A}'$ stores the states of $\mathscr{A}$ on the stack, and the height of state pairs can be computed by Lemma 6.

- There is an additional flag for each $i \in \{0,\dots,h\}$ indicating whether counter number i was reset because a final state of $\mathscr{A}$ has been visited (the flag is set to 1), or because it reached its maximal value m (the flag is set to 0).
- When counter number i reaches value m (if several counters reach m at the same time we take the maximal such i), then the automaton signals priority $2i+2$ if the flag number i is set, and $2i+1$ if the flag is not set. In the next transition the counter is reset.

Formal description of the parity DVPA. Recall that $m := |Q|^3 + 1$ and that h is the maximal height of a pair of states from $Q \setminus F$.

- The states of $\mathscr{A}'$ are of the form (q,χ,f), where $q \in Q$ is a state of $\mathscr{A}$, $\chi : \{0,\dots,h\} \to \{0,\dots,m\}$ represents the counters mentioned above, and $f : \{0,\dots,h\} \to \{0,1\}$ represents the flag mentioned in the informal description.
- The stack symbols of $\mathscr{A}'$ are of the form $[Z,(q,\chi,f)]$, where Z is a stack symbol of $\mathscr{A}$ and (q,χ,f) is a state of $\mathscr{A}'$.
- We now define when $\mathscr{A}'$ can move from state (q,χ,f) to state (q',χ',f'), depending on whether it reads a call, an internal action, or a return. In all cases, q' is the next state of $\mathscr{A}$, i.e., $\mathscr{A}'$ simulates $\mathscr{A}$ in its first component. If $q' \in F$, then $\chi' = 0$ and $f' = 1$, i.e., the constant functions mapping everything to 0 and 1, respectively. The other cases for δ' are listed below:

  Call: $(q,\chi,f) \xrightarrow{c} \begin{array}{c}(q',\chi',f')\\ [Z,(q,\chi,f)]\end{array}$ if $\delta(q,c) = (Z,q')$, $q' \notin F$, and

  $$\chi'(i) = \begin{cases} (\chi(i) \mod m) + 1 \text{ if } i = 0, \\ (\chi(i) \mod m) \text{ otherwise,} \end{cases} \qquad f'(i) = \begin{cases} f(i) \text{ if } \chi(i) < m, \\ 0 \text{ otherwise.} \end{cases}$$

  Internal action: $(q,\chi,f) \xrightarrow{a} (q',\chi',f')$ if $\delta(q,a) = q'$, $q' \notin F$, and χ' and f' are as in the case of a call symbol.

  Return: $\begin{array}{c}(q,\chi,f)\\ [Z,(q'',\chi'',f'')]\end{array} \xrightarrow{r} (q',\chi',f')$ if $\delta(q,Z,r) = q'$, $q' \notin F$, and

  $$\chi'(i) = \begin{cases} (\chi''(i) \mod m) + 1 \text{ if } q'' \notin F \text{ and } i \leq ht(q'',q'), \\ (\chi''(i) \mod m) \text{ otherwise,} \end{cases}$$

  $$f'(i) = \begin{cases} f''(i) \text{ if } \chi''(i) < m, \\ 0 \text{ otherwise.} \end{cases}$$

- The priority function Ω' of $\mathscr{A}'$ is defined as follows

  $$\Omega'(q,\chi,f) = \begin{cases} 0 \text{ if } \chi(i) < m \text{ for all } i, \\ 2d + 1 + f(d) \text{ if } d = \max\{i \mid \chi(i) = m\}. \end{cases}$$

- The initial state is (q_0,χ_0,f_0) with $\chi_0 = 0$ and $f_0 = 1$.

Lemma 7. *The parity DVPA $\mathscr{A}'$ is equivalent to $\mathscr{A}$.*

Proof. We note the following helpful fact on reachable states (q,χ,f) of $\mathscr{A}'$:

(1) If $f(i) = 1$ for some i, then $f(j) = 1$ and $\chi(i) \geq \chi(j)$ for all $j \geq i$. The initial state satisfies this property, and if we apply the definition of the transition function to a state satisfying the property, then one can easily verify that the resulting state also satisfies it.

Now consider an accepting run of $\mathscr{A}$. We show that the corresponding run of $\mathscr{A}'$ is also accepting. Let the kth state in this run of $\mathscr{A}'$ be (q_k, χ_k, f_k).

If ℓ is a step in the run and q_ℓ is a final state of $\mathscr{A}$, then all flags are set to 1 at this point. From the definition of δ' follows that these flags can only be set to 0 if the corresponding counter reaches value m (we assume that the final state occurs on a step and therefore the run never accesses the stack symbols below). Now assume that $\mathscr{A}'$ signals some odd priority $2i+1$ at some position k after this final state. This means that i is maximal with $\chi_k(i) = m$, and furthermore $f_k(i) = 0$. But if $f_k(i) = 0$, then there must be some k' with $\ell < k' < k$ such that $f_{k'}(i) = 1$ and $\chi_{k'}(i) = m$ because this is the only situation in which the flag is set to 0.

From (1) we conclude that $f_{k'}(j) = 1$ for all $j \geq i$ and hence $\Omega'(q_{k'}, \chi_{k'}, f_{k'})$ is an even priority bigger than $2i+1$. Thus, for each odd priority occurring after a final state on a step there is a bigger even priority also occurring after this final state. Hence, the run of $\mathscr{A}'$ is also accepting.

For the other direction, consider a non-accepting run of $\mathscr{A}$ and as before let (q_k, χ_k, f_k) be the kth state in the corresponding run of $\mathscr{A}'$. There is a position such that after this position no final states of $\mathscr{A}$ occur on a step. From now on we only consider this part of the run.

Consider the sequence $k_1, k_2, k_3, \ldots$ of steps. As no final state occurs on a step we have the following relation between the counter values at two successive steps:

(i) If k_{j+1} was reached from k_j by reading a call or an internal symbol, then the only change of the counters is $\chi_{k_{j+1}}(0) = (\chi_{k_j}(0) \mod m) + 1$. The other values remain the same.

(ii) If k_{j+1} was reached from k_j by reading a minimally well-matched word, then the counters are updated as follows:

$$\chi_{k_{j+1}}(i) = \begin{cases} (\chi_{k_j}(i) \mod m) + 1 \text{ if } i \leq ht(q_{k_j}, q_{k_{j+1}}), \\ (\chi_{k_j}(i) \mod m) \text{ otherwise.} \end{cases}$$

The flags between two successive steps are updated as follows:

$$f_{k_{j+1}}(i) = \begin{cases} f_{k_j}(i) \text{ if } \chi_{k_j}(i) < m, \\ 0 \text{ otherwise.} \end{cases}$$

Now let d be the highest counter that is infinitely often increased on a step (such a counter exists because counter 0 is increased for each call and each internal symbol). Then the highest priority occurring on a step is obviously $2d+1$ because after the first reset of counter d to 0 the flag number d is 0 on all following steps.

We have to show that no even priority higher than $2d+1$ can occur infinitely often. Restrict the part of the run under consideration further to the suffix on which no counter higher than d is incremented on a step. We can conclude that for successive steps connected by a minimally well-matched word we have that $ht(q_{k_j}, q_{k_{j+1}}) \leq d$.

We first assume that $d > 0$. At the end of the proof we briefly explain the case $d = 0$.

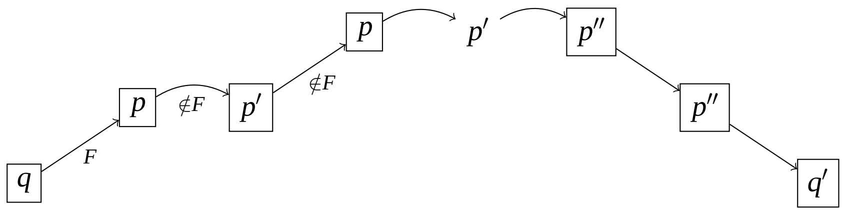
Pick j such that there is ℓ with $k_j < \ell < k_{j+1}$ and $\Omega'(q_\ell, \chi_\ell, f_\ell) = 2i+2$ (if no such position exists, then the run of $\mathscr{A}'$ is clearly rejecting). For simplicity let $(q_{k_j}, \chi_{k_j}, f_{k_j}) = (q, \chi, f)$ and $(q_{k_{j+1}}, \chi_{k_{j+1}}, f_{k_{j+1}}) = (q', \chi', f')$.

We now consider the part of the run from k_j to ℓ and show that $i < ht(q,q') \leq d$ and hence $2i+2 < 2d+1$.

Since $\Omega'(q_\ell, \chi_\ell, f_\ell) = 2i+2$ we know that $f_\ell(i) = 1$ and i is maximal with $\chi_\ell(i) = m$. If $i = 0$ we know that $i < d$ by our assumption $d > 0$. If $i > 0$, at position ℓ a pair of states of height i is closed. From Lemma 5 we obtain that $d \geq ht(q,q') \geq i$.

There are two cases to consider. If flag number i was already set to 1 at position k_j, i.e., $f(i) = 1$, then $i \neq d$ (as we only consider the part of the run where the flag for d remains 0 forever on the steps). Together with $d \geq i$ we get $d > i$.

If $f(i) = 0$, then it must be reset to 1 by visiting a final state. At the same time the counters are reset to 0. Then m pairs of height i have to be closed to reach the value $\chi_\ell(i) = m$. Furthermore, these pairs have to closed at positions that correspond to steps in the part of the run between k_j and ℓ (not steps in the whole run). Let these pairs be $(p_1,p_1'),(p_2,p_2'),\dots,(p_m,p_m')$ (see Figure 7) and the corresponding pairs of positions be $(\ell_1,\ell_1')\dots,(\ell_m,\ell_m')$. Now consider for each n the minimal position ℓ_n'' with $\ell \leq \ell_n'' \leq k_{j+1}$ such that the stack height at ℓ_n' and ℓ_n'' is the same. Let p_n'' denote the state at the corresponding position. By the choice of m we get that there are $n_1 \neq n_2$ such that $(p_{n_1}, p_{n_1}', p_{n_1}'') = (p_{n_2}, p_{n_2}', p_{n_2}'')$. Denote the corresponding triple by (p,p',p''). This triple witnesses that $ht(q,q') > ht(p,p') = i$ as illustrated in the following picture:



It remains to consider the case $d = 0$. Consider only the suffix of the run after the position where the flag for counter 0 remains 0 on all steps and no other counter is increased on a step anymore. Then all pairs closed on steps are of height 0 and by Lemma 5 pairs closed between two successive steps are also of height 0. So the maximal priority that we can see on this part of the run would be 2. For this to happen, the flag for counter 0 must be 1 and counter 0 must have value m. The flags are only set to 1 if a final state of $\mathscr{A}$ is reached, and at the same time the counters are set to 0. Let q,q' be the states at two successive steps, and assume that in between a final state is seen. Let p be the state after the symbol following the final state. If this symbol is a call or an internal, then $(p,p) \prec (q,q')$ (choosing $p'' = p$), contradicting $ht(q,q') = 0$. Thus, each final state of $\mathscr{A}$ is immediately followed by a return. Thus, whenever the flag is set to 1 by a final state, it is immediately reset to 0 in the next transition, and thus priority 2 never occurs (on the considered part of the run). ☐

Combining Lemmas 3 and 7 we obtain the following.

Theorem 7. *A stair Büchi DVPA $\mathscr{A}$ is equivalent to a parity DVPA if, and only if, it does not contain any forbidden patterns.*

The relation $\prec$ can be computed and checked for irreflexivity in polynomial time. Hence we get the following corollary.

Corollary 3. *For a stair Büchi DVPA $\mathscr{A}$ it is decidable in polynomial time if it is equivalent to some parity DVPA.*

A direct consequence of Lemma 7 is:

Theorem 8. *If a stair Büchi DVPA $\mathscr{A}$ is equivalent to some parity DVPA, then we can effectively construct such a parity DVPA.*

It seems possible to lift the methods presented in this section to decide for general stair parity DVPAs whether the stair condition is required. We have, however, not yet worked out the details. A simpler question can be solved using the game theoretic approach for deciding the parity index problem for DPDAs: Given a stair parity DVPA $\mathscr{A}$ and a set P of priorities, we can decide whether there is a parity DVPA using the priorities from P that accepts $L_\omega(\mathscr{A})$ by using the classification game. In this case, the classification game could be formalized using a combination of a classical parity and a stair parity condition. Pushdown games with such a winning condition can be solved with the methods from [11].

6 Conclusion

We have considered several decidability questions for ω-DPDAs. The regularity and equivalence problem are still open for the full class of ω-DPDAs. We have sketched some partial results from [12] showing the decidability for these two problems for the class of weak ω-DPDAs by a reduction to the corresponding problems for DPDAs on finite words. It seems that a decidability result for the full class of ω-DPDAs requires new ideas.

In the second part we have analyzed the problem of simplifying the acceptance condition of ω-DPDAs. We have shown that the smallest number of priorities required for accepting the language of a given parity DPDA can be computed. For the standard parity condition we have used a game approach. For stair parity DVPAs, this problem can be solved by a much simpler algorithm that uses a reduction to the computation of the parity index of a finite automaton.

We have also shown that for stair Büchi DVPAs it is decidable whether the stair condition is required or whether there exists an equivalent parity DVPA. It seems that the methods used in the proof can be generalized from stair Büchi conditions to arbitrary stair parity conditions but we have not worked out the details.

References

[1] Rajeev Alur & Parthasarathy Madhusudan (2004): *Visibly pushdown languages*. In: *STOC '04: Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, ACM Press, New York, NY, USA, pp. 202–211, doi:10.1145/1007352.1007390.

[2] Christel Baier & Joost-Pieter Katoen (2008): *Principles of Model Checking*. MIT Press.

[3] T. Cachat, J. Duparc & W. Thomas (2002): *Solving Pushdown Games with a Σ_3 Winning Condition*. In: *Proceedings of the 11th Annual Conference of the European Association for Computer Science Logic, CSL 2002, Lecture Notes in Computer Science* 2471, Springer, pp. 322–336, doi:10.1007/3-540-45793-3_22.

[4] Olivier Carton & Ramón Maceiras (1999): *Computing the Rabin Index of a Parity Automaton*. *ITA* 33(6), pp. 495–506, doi:10.1051/ita:1999129.

[5] Rina S. Cohen & Arie Y. Gold (1978): *Omega-Computations on Deterministic Pushdown Machines*. *JCSS* 16(3), pp. 275–300, doi:10.1016/0022-0000(78)90019-3.

[6] Javier Esparza, David Hansel, Peter Rossmanith & Stefan Schwoon (2000): *Efficient Algorithms for Model Checking Pushdown Systems*. In: *CAV*, pp. 232–247, doi:10.1007/10722167_20.

[7] W. Fridman (2010): *Formats of Winning Strategies for Six Types of Pushdown Games*. In A. Montanari, M. Napoli & M. Parente, editors: *Proceedings of the First Symposium on Games, Automata, Logic, and*

Formal Verification, GandALF 2010, 25, Electronic Proceedings in Theoretical Computer Science, pp. 132–145, doi:10.4204/EPTCS.25.14.

[8] Erich Grädel, Wolfgang Thomas & Thomas Wilke, editors (2002): *Automata, Logics, and Infinite Games: A Guide to Current Research [outcome of a Dagstuhl seminar, February 2001]*. *Lecture Notes in Computer Science* 2500, Springer, doi:10.1007/3-540-36387-4.

[9] John E. Hopcroft & Jeffrey D. Ullman (1979): *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley.

[10] Matti Linna (1977): *A Decidability Result for Deterministic omega-Context-Free Languages*. *Theor. Comput. Sci.* 4(1), pp. 83–98, doi:10.1016/0304-3975(77)90058-5.

[11] Christof Löding, Parthasarathy Madhusudan & Oliver Serre (2004): *Visibly pushdown games*. In: *FSTTCS 2004*, *Lecture Notes in Computer Science* 3328, Springer, pp. 408–420, doi:10.1007/978-3-540-30538-5_34.

[12] Christof Löding & Stefan Repke (2012): *Regularity Problems for Weak Pushdown ω-Automata and Games*. In: *Mathematical Foundations of Computer Science 2012*, *Lecture Notes in Computer Science* 7464, Springer Berlin / Heidelberg, pp. 764–776, doi:10.1007/978-3-642-32589-2_66.

[13] Dominique Perrin & Jean-Éric Pin (2004): *Infinite words*. *Pure and Applied Mathematics* 141, Elsevier.

[14] Stefan Repke (2014): *Simplification Problems for Automata and Games*. Ph.D. thesis, RWTH Aachen, Germany.

[15] Géraud Sénizergues (2001): *L(A)=L(B)? decidability results from complete formal systems*. *Theor. Comput. Sci.* 251(1-2), pp. 1–166, doi:10.1016/S0304-3975(00)00285-1.

[16] Ludwig Staiger (1983): *Finite-State ω-Languages*. *JCSS* 27(3), pp. 434–448. Available at `http://dx.doi.org/10.1016/0022-0000(83)90051-X`.

[17] Richard E. Stearns (1967): *A Regularity Test for Pushdown Machines*. *Information and Control* 11(3), pp. 323–340, doi:10.1016/S0019-9958(67)90591-8.

[18] Philipp Stephan (2006): *Deterministic Visibly Pushdown Automata over Infinite Words*. Diploma thesis, RWTH Aachen.

[19] Howard Straubing (1994): *Finite Automata, Formal Logic, and Circuit Complexity*. Birkhäuser, Basel, Switzerland, doi:10.1007/978-1-4612-0289-9.

[20] Leslie G. Valiant (1975): *Regularity and Related Problems for Deterministic Pushdown Automata*. *J. ACM* 22(1), pp. 1–10. Available at `http://doi.acm.org/10.1145/321864.321865`.

[21] William W. Wadge (1984): *Reducibility and Determinateness on the Baire Space*. Ph.D. thesis, University of California, Berkeley.

[22] Igor Walukiewicz (2001): *Pushdown Processes: Games and Model Checking*. *Information and Computation* 164(2), pp. 234–263, doi:10.1006/inco.2000.2894.

Equivalence Problems for Tree Transducers: A Brief Survey

Sebastian Maneth
School of Informatics
University of Edinburgh
smaneth@inf.ed.ac.uk

The decidability of equivalence for three important classes of tree transducers is discussed. Each class can be obtained as a natural restriction of deterministic macro tree transducers (MTTs): (1) no context parameters, i.e., top-down tree transducers, (2) linear size increase, i.e., MSO definable tree transducers, and (3) monadic input and output ranked alphabets. For the full class of MTTs, decidability of equivalence remains a long-standing open problem.

1 Introduction

The macro tree transducer (MTT) was invented independently by Engelfriet [20, 29] and Courcelle [13, 14] (see also [37]). As a model of syntax-directed translations, MTTs generalize the attribute grammars of Knuth [46]. Note that one (annoying) issue of attribute grammars is that they can be circular; MTTs always terminate. Macro tree transducers are a combination of context-free tree grammars, invented by Rounds [57] and also known as "macro tree grammars" [34], and the top-down tree transducer of Rounds and Thatcher [58, 67]: the derivation of the grammar is (top-down) controlled by a given input tree. In terms of a top-down transducer, the combination is obtained by allowing nesting of state calls in the rules (similar to the nesting of nonterminals in the productions of context-free tree grammars). Top-down tree transducers generalize to trees the finite state (string) transducers (also known as "generalized sequential machines", or GSMs, see [38, 8]). In terms of formal languages, compositions of MTTs give rise to a large hierarchy of string languages containing, e.g., the IO and OI hierarchies (at level one they include the indexed languages of Aho [1]), see [22]. MTTs can be applied in many scenarios, e.g., to type check XML transformations (they can simulate the *k*-pebble transducers of Milo, Suciu, and Vianu [51]), see [23, 49, 50], or to efficiently implement streaming XQuery transformations [42, 52]. In terms of functional programs, MTTs are particularly simple programs that do primitive recursion over an input tree and only produce trees as output. Applications in programming languages exist include [68, 69, 52].

Equivalence of nondeterministic transducers is undecidable, already for restricted string transducers [40]. We therefore only consider deterministic transducers. What is known about the equivalence problem for deterministic macro tree transducers? Unfortunately not much in the general case. Only a few subcases are known to be decidable. Here we describe three of them:

(1) top-down tree transducers

(2) linear size increase transducers

(3) monadic tree transducers.

The first one was solved long ago by Esik [30], but was revived through the "earliest canonical normal form" by Engelfriet, Maneth, and Seidl [26]. The latter implies PTIME equivalence check for total top-down tree transducers. The second one is solved by the decidability of equivalence for deterministic MSO tree transducers of Engelfriet and Maneth [25]. The same authors have shown that every MTT of linear size increase is effectively equal to an MSO transducer [24]. Hence, decidability of equivalence follows for MTTs of linear size increase. Here we give a direct proof using MTTs. The third result is about MTTs over monadic trees. These are string transducers with copying. The decidability of their equivalence problem follows through a relationship with L-systems [28]; in particular, with the sequence equivalence problem of HDT0L systems. The latter was first proved decidable by Culik II and Karumhäki [18].

2 Preliminaries

We deal with finite, ordered, ranked trees. In such a tree, each node is labeled by a symbol from a ranked alphabet such that the rank of the symbol is equal to the number of children of the node. Formally, a *ranked alphabet* consists of a finite set Σ together with a mapping $\mathsf{rank}_\Sigma : \Sigma \to \mathbb{N}$ associating to each symbol its rank. We write $\sigma^{(k)}$ to denote that the rank of σ is equal to k. By $\Sigma^{(k)}$ we denote the subset of symbols of Σ that have rank k. Let Σ be a ranked alphabet. The *set of all trees over* Σ, denoted T_Σ, is the smallest set of strings T such that if $k \geq 0$, $t_1,\dots,t_k \in T$, and $\sigma \in \Sigma^{(k)}$, then also $\sigma(t_1,\dots,t_k) \in T$. For a tree of the form $a()$ we simply write a. For a set A, we denote by $T_\Sigma(A)$ the set of all trees over $\Sigma \cup A$ such that the rank of each $a \in A$ is zero. Let $a_1,\dots,a_n$ be distinct symbols in $\Sigma^{(0)}$ and let $t_1,\dots,t_n \in T_\Sigma$ such that none of the leaves in t_j are labeled by a_i for $1 \leq i \leq n$. Then by $[a_i \leftarrow t_i \mid i \in \{1,\dots,n\}]$ we denote the *tree substitution* that replaces each leaf labeled a_i by the tree t_i. Thus $d(a,b,a)[a \leftarrow c(b), b \leftarrow a]$ denotes the tree $d(c(b),a,c(b))$.

Let $t \in T_\Sigma$ for some ranked alphabet Σ. We denote the *nodes of t* by their Dewey dotted decimal path and define the set $V(t)$ of nodes of t as $\{\varepsilon\} \cup \{i.u \mid 1 \leq i \leq k, u \in V(t_i)\}$ if $t = \sigma(t_1,\dots,t_k)$ with $k \geq 0$, $\sigma \in \Sigma^{(k)}$, and $t_1,\dots,t_k \in T_\Sigma$. Thus, ε denotes the root node, and $u.i$ denotes the ith child of the node u. For a node $u \in V(t)$ we denote by $t[u]$ its label, and, for a tree t' we denote by $t[u \leftarrow t']$ the tree obtained from t by replacing its subtree rooted at u by the tree t'. The *size* of t, denoted $|t|$, is its number $|V(t)|$ of nodes. The *height* of t, denoted $\mathsf{height}(t)$, is defined as $\mathsf{height}(\sigma(t_1,\dots,t_k)) = 1 + \max\{\mathsf{height}(t_i) \mid 1 \leq i \leq k\}$ for $k \geq 0$, $\sigma \in \Sigma^{(k)}$, and $t_1,\dots,t_k \in T_\Sigma$.

We fix the set of *input variables* $X = \{x_1,x_2,\dots\}$ and the set of *formal context parameters* $Y = \{y_1,y_2,\dots\}$. For $n \in \mathbb{N}$ we define $X_n = \{x_1,\dots,x_n\}$ and $Y_n = \{y_1,\dots,y_n\}$.

A *deterministic finite-state bottom-up tree automaton* is a tuple $A = (Q,\Sigma,\delta,Q_f)$ where Q is a finite set of states, $Q_f \subseteq Q$ is the set of final states, and for every $\sigma \in \Sigma^{(k)}$ and $k \geq 0$, δ_σ is a function from Q^k to Q. The transition function δ is extended to a mapping from T_Σ to Q in the obvious way, and the set of trees accepted by A is $L(A) = \{s \in T_\Sigma \mid \delta(s) \in Q_f\}$.

3 Macro Tree Transducers

Definition 1 A *(deterministic) macro tree transducer M* is a tuple (Q,Σ,Δ,q_0,R) such that Q is a ranked alphabet of states with $Q^{(0)} = \emptyset$, Σ and Δ are ranked alphabets of input and output symbols, respectively, $q_0 \in Q^{(1)}$ is the initial state, and for every $q \in Q^{(m+1)}$, $m \geq 0$, and $\sigma \in \Sigma^{(k)}$, $k \geq 0$, the set of rules R contains at most one rule of the form

$$q(\sigma(x_1,\dots,x_k),y_1,\dots,y_m) \to t$$

where $t \in T_{\Delta \cup Q}(X_k \cup Y_m)$ such that if a node in t has its label in Q, then its first child has its label in X_k. A rule as above is called a (q,σ)-rule and its right-hand side is denoted by $\mathsf{rhs}(q,\sigma)$.

The translation $\tau_M : T_\Sigma \to T_\Delta$ (often just denoted M) realized by an MTT M is a partial function recursively defined as follows. For each state q of rank $m+1$, $M_q : T_\Sigma \to T_\Delta(Y_m)$ is the translation of M starting in state q, i.e., "starting" with a tree $q(s, y_1, \dots, y_m)$ where $s \in T_\Sigma$. For instance, for $a \in \Sigma^{(0)}$, $M_q(a)$ simply equals $\mathsf{rhs}(q,a)$. In general, for an input tree $s = \sigma(s_1, \dots, s_k)$, $M_q(s)$ is obtained from $\mathsf{rhs}(q,\sigma)$ by repeatedly replacing a subtree of the form $q'(x_i, t_1, \dots, t_n)$ with $t_1, \dots t_n \in T_\Delta(Y_m)$ by the tree $M_{q'}(s_i)[y_j \leftarrow t_j \mid 1 \leq j \leq n]$. The latter tree will also be written $M_{q'}(s_i, t_1, \dots, t_n)$. We define $\tau_M = M_{q_0}$.

By the definition above, all our MTTs are deterministic and we refer to them as "macro tree transducers" and denote their class of translations by MTT. An MTT is *total* if there is exactly one rule of the above form. If each state of an MTT is of rank one, then it is a *top-down tree transducer*. The class of translations realized by (deterministic) top-down tree transducers is denoted by T. In Lemmas 3 and 5 we make use of nondeterministic top-down tree transducers and mark the corresponding class there by the letter "N". A transducer is nondeterministic if there are several (q,σ)-rules in R. An MTT is *monadic* if its input and output ranked alphabets are monadic, i.e., only contain symbols of rank 1 and 0. An MTT M is of *linear size increase* if there is a constant c such that $|\tau_M(s)| \leq c \cdot |s|$ for every $s \in T_\Sigma$.

As an example, consider the transducer M consisting of the rules in Figure 1. The alphabets of this

$$\begin{array}{lcl}
q_0(d(x_1,x_2)) & \to & d(q(x_1, 1(e)), q(x_2, 2(e))) \\
q_0(a) & \to & a(e) \\
q(d(x_1,x_2), y_1) & \to & d(q(x_1, 1(y_1)), q(x_2, 2(y_1))) \\
q(a, y_1) & \to & a(y_1)
\end{array}$$

Figure 1: The macro tree transducer M adds reverse Dewey paths to leaves

transducer are $Q = \{q_0^{(1)}, q^{(2)}\}$, $\Sigma = \{d^{(2)}, a^{(0)}\}$, and $\Delta = \{d^{(2)}, a^{(1)}, 1^{(1)}, 2^{(1)}, e^{(0)}\}$. The MTT M translates a binary tree into the same tree, but additionally adds under each leaf the reverse (Dewey) path of the node. For instance, $s = d(d(a,a),a)$ is translated into $d(d(a(1(1(e))), a(2(1(e)))), a(2(e)))$ as can be verified in this computation of M

$$\begin{array}{l}
M(d(d(a,a),a)) = \\
d(M_q(d(a,a), 1(e)), M_q(a, 2(e))) = \\
d(d(M_q(a, 1(1(e))), M_q(a, 2(1(e)))), a(2(e))) = \\
d(d(a(1(1(e))), a(2(1(e)))), a(2(e))).
\end{array}$$

Note that M is *not* of linear size increase. Hence, it is not MSO definable. Since the translation is neither top-down nor monadic, it falls into a class of MTTs for which we do not know a procedure to decide equivalence. As an exercise, the reader may wonder whether there is an MTT that is similar to M, but outputs Dewey paths below leaves (instead of their *reverses*). In contrast, consider input trees with only exactly one a-leaf (and all other leaves labeled differently). Then an MTT of linear size increase can output under the unique a-leaf its reverse Dewey path, i.e., this translation is MSO definable. Is it now possible with an MTT to output the non-reversed Dewey path?

One of the most useful properties of MTTs is the effective preservation of regular tree languages by their inverses. This is used inside the proofs of several results presented here. For instance, to prove that for every MTT there is an equivalent one which is nondeleting in the parameters (= strict), one can use the above property as follows: given a state q of rank $m+1$ and a subset $A \subseteq Y_m$, the language $T_\Delta(A)$ is regular, and hence $M_q^{-1}(T_\Delta(A))$ is the regular set of inputs for which q outputs only parameters in the set A. Thus, by using regular look-ahead we can determine which parameters are used, and can change the rules to call an appropriate state q' which is only provided the parameters in A which it uses. Regular look-ahead is explained in Section 4.1.

The following result is stated in Theorem 7.4 of [29]. A similar proof as below is given at the end of [23]. A slightly simpler proof, for a slightly larger class, is presented by Perst and Seidl for macro forest transducers [54].

Lemma 1 Let M be an MTT with output alphabet Δ and let $R \subseteq T_\Delta$ be a regular tree language (given by a bottom-up tree automaton B). Then $\tau_M^{-1}(R)$ is effectively regular. In particular, the domain $\mathsf{dom}(\tau_M)$ is effectively regular.

Proof. Let $M = (Q, \Sigma, \Delta, q_0, R)$ and $B = (P, \Delta, \delta, P_f)$. We construct the new automaton $A = (S, \Sigma, \delta', S_f)$. The states of A are mappings α that associate with each state $q \in Q^{(m+1)}$ a mapping $\alpha(q) : P^m \to P$. The set S_f consists of all $\alpha \in S$ such that $\alpha(q_0) \in P_f$. For $a \in \Sigma^{(0)}$ we define $\delta'_a() = \alpha$ such that for every $q \in Q^{(m+1)}$ and $p_1, \ldots, p_m$, $(\alpha(q))(p_1, \ldots, p_m) = \delta^*(\mathsf{rhs}(q,a)[y_j \leftarrow p_j \mid j \in [m]])$. Here, δ^* is the extension of δ to trees in $T_\Delta(P)$ by the rule $\delta(p) = p$ for all $p \in P$. Now let $b \in \Sigma^{(k)}$ with $k \geq 1$. For $\alpha_1, \ldots, \alpha_k \in S$ we define $\delta'_b(\alpha_1, \ldots, \alpha_k) = \alpha$ where for every $q \in Q^{(m+1)}$ and $p_1, \ldots, p_m$, $(\alpha(q))(p_1, \ldots, p_m) = \underline{\delta}^*(\mathsf{rhs}(q,b)[y_j \leftarrow p_j \mid j \in [m]])$. Now, $\underline{\delta}^*$ is the extension of δ^* of above to symbols $q' \in Q^{(n+1)}$ by $\delta(q'(x_i, p'_1, \ldots, p'_n)) = (\alpha_i(q'))(p'_1, \ldots, p'_n)$ for every $p'_1, \ldots, p'_n \in P$. $\square$

3.1 Bounded Balance

Many algorithms for deciding equivalence of transducers are based on the notion of bounded balance. Intuitively, two transducers have bounded balance if the difference of their outputs on any "partial" input is bounded by a constant. For instance, for two D0L systems $G_i = (\Sigma, h_i, \sigma)$ with $i = 1, 2$, Culik II defines [16] the balance of a string $w \in \Sigma^*$ as the difference of the lengths of $h_1(w)$ and $h_2(w)$. He shows that equivalence is decidable for D0L systems that have bounded difference. In a subsequent article [17], Culik II and Fris show that any two equivalent D0L systems in normal form have bounded difference, thus giving the first solution to the famous D0L equivalence problem.

For a tree transducer M a partial input is an input tree which contains exactly on distinguished leaf labeled x, and x is a fresh symbol not in Σ. More precisely, a partial input is a tree $p = s[u \leftarrow x]$ where s is in the domain of M and u is a node of s. Since the transducer has no rules for x, the computation on the input p "blocks" at the x-labeled node. Thus, the tree $M(p)$ may contain subtrees of the form $q(x, t_1, \ldots, t_m)$ where q is a state of rank $m+1$. For instance, consider the transducer M shown in the left of Figure 2 and consider the partial input tree $s = a(a(x))$. Then

$$M(a(a(x))) = d(M(a(x)), M(a(x))) = d(d(q_0(x), q_0(x)), d(q_0(x), q_0(x))).$$

It should be clear that if we replace each $q_0(x)$ by $M_{q_0}(s')$ so that $s' \in T_\Sigma$ is a tree with $\hat{s} = s[x \leftarrow s'] \in \mathsf{dom}(M)$, then we obtain the output $M(\hat{s})$ of M on input tree $\hat{s}$. For instance, we may pick $s' = e$ above; then we obtain $d(d(e,e), d(e,e))$ which indeed equals $M(a(a(e)))$.

Consider the trees $M_1(p)$ and $M_2(p)$ for two MTTs M_1 and M_2. What is the balance of these two trees? There are two natural notions of balance: either we compare the sizes of $M_i(p)$, or we compare their heights. Given two trees t_1, t_2 we define their *size-balance* (for short, *s-balance*) as $||t_1| - |t_2||$ and their *height-balance* (for short, *h-balance*) as $|\mathsf{height}(s_1) - \mathsf{height}(s_2)|$. Two transducers M_1, M_2 have *bounded s-balance* (resp. h-balance) if there exists a $c > 0$ such that for any partial input p the s-balance (resp. h-balance) of $M_1(p)$ and $M_2(p)$ is at most c. Obviously, bounded h-balance implies bounded s-balance, but not vice versa.

Let M and N be equivalent MTTs. Do M and N have bounded size-balance? To see that this is in general *not* the case, it suffices to consider two simple top-down tree transducers: M translates a monadic partial input of the form $a^n(x)$ into the full binary tree of height n containing 2^n occurrences of the subtree $q_0(x)$. The transducer N translates $a^n(x)$ into the full binary tree of height $n-1$ with 2^{n-1} occurrences of the subtree $p(x)$. The rules of M and N are shown in Figure 2. Clearly, M and N are of bounded

$$\begin{array}{lll} M: & q_0(a(x_1)) & \to & d(q_0(x_1), q_0(x_1)) \\ & q_0(e) & \to & e \end{array} \qquad \begin{array}{llll} N: & p_0(a(x_1)) & \to & p(x_1) \\ & p_0(e) & \to & e \\ & p(a(x_1)) & \to & d(p(x_1), p(x_1)) \\ & p(e) & \to & d(e,e) \end{array}$$

Figure 2: Equivalent top-down tree transducers M and N with unbounded size-balance

height-balance (with constant $c = 1$). But, their size-balance is not bounded.

In a similar way it can be seen that equivalent MTTs need not have bounded height-balance. This even holds for monadic MTTs: Consider the transducers M' and N' with rules shown in Figure 3. Their

$$\begin{array}{llll} M': & q_0(a(x_1)) & \to & q(x_1, q_0(x_1)) \\ & q_0(e) & \to & e \\ & q(a(x_1), y_1) & \to & q(x_1, y_1) \\ & q(e, y_1) & \to & a(a(y_1)) \end{array} \quad \begin{array}{llll} N': & p_0(a(x_1)) & \to & p(x_1, p(x_1, p_0(x_1))) \\ & p_0(e) & \to & e \\ & p(a(x_1), y_1) & \to & p(x_1, y_1) \\ & p(e, y_1) & \to & a(y_1) \end{array}$$

Figure 3: Equivalent monadic macro tree transducers M' and N' with unbounded height-balance

height-balance on input $a^n(x)$ is equal to n; for instance, for the tree $a(a(x))$ we obtain

$$\begin{array}{ll} M'(a(a(x))) = & N'(a(a(x))) = \\ M'_q(a(x), M'(a(x))) = & N'_p(a(x), N'_p(a(x), N'(a(x)))) = \\ q(x, q(x, q_0(x))) & p(x, p(x, p(x, p(x, p_0(x))))) \end{array}$$

which are trees of height 3 and 5, respectively. We may verify that replacing x by e results in equal trees:

$$M'_q(e, M'_q(e, M(e))) = M'_q(e, M'_q(e, e)) = M'_q(e, a(a(e))) = a(a(a(a(e)))).$$

And for N' we obtain that

$$N'_p(e, N'_p(e, N'_p(e, N'_p(e, N(e))))) = N'_p(e, N'_p(e, N'_p(e, N'_p(e, e)))) = N'_p(e, N'_p(e, N'_p(e, a(e)))) = \\ N'_p(e, N'_p(e, a(a(e)))) = N'_p(e, a(a(a(e)))) = a(a(a(a(e)))).$$

Let us now show that equivalent top-down tree transducers have bounded height-balance.

Lemma 2 Equivalent top-down tree transducers effectively have bounded height-balance.

Proof. Let M_1, M_2 be equivalent top-down tree transducers with sets of states Q_1, Q_2, respectively. Note that they have the same domain D. Let $s \in D$ and $u \in V(s)$. Consider the two trees $\xi_i = M_i(s[u \leftarrow x])$ for $i = 1, 2$. Let s' be a smallest input tree such that $s[u \leftarrow s'] \in D$. It should be clear that the height of s' is bounded by some constant d. In fact, let d be the height of a smallest tree in the set $(\bigcap_{q \in Q} \mathsf{dom}(M_{1,q}) \cap (\bigcap_{q' \in Q'} \mathsf{dom}(M_{2,q'}))$, for any subsets $Q \subseteq Q_1$ and $Q' \subseteq Q_2$. Since s' is in such a set, its height is at most d.

This bound d can be computed because by Lemma 1 the sets $\mathsf{dom}(M_{1,q})$ and $\mathsf{dom}(M_{2,q'})$ are effectively regular, and regular tree languages are effectively closed under intersection [11]. In fact, it is not difficult to see that we can choose $d = 2^{|Q_1|+|Q_2|}$. Hence, there is a constant c such that $\mathsf{height}(M_{i,q_i}(s')) < c$ for any $q_i \in Q_i$ appearing in ξ_i. Clearly we can take $c = d \cdot h$, where h is the maximal height of the right-hand side of any rule of M_1 and M_2. This means that $|\mathsf{height}(\xi_1) - \mathsf{height}(\xi_2)| \leq c$ because $\xi_1\Theta_1 = \xi_2\Theta_2$ and the substitutions $\Theta_i = [q(x) \leftarrow M_{i,q}(s') \mid q \in Q_i]$ increase the height of ξ_i by at most c. □

If the transducers M_1, M_2 of Lemma 2 are total, then $d = 1$ and c is the maximal size of the right-hand side of any rule for an input leaf symbol, i.e., $c = \max\{\mathsf{height}(\mathsf{rhs}(M_i, q_i, a)) \mid i \in \{1,2\}, q_i \in Q_i, a \in \Sigma^{(0)}\}$.

Let us consider an example of two equivalent top-down tree transducers M and N with output paths

$$
\begin{array}{llll@{\qquad}llll}
M: & q_0(a(x_1)) & \to & d(q(x_1), q_0(x_1)) & N: & p_0(a(x_1)) & \to & p(x_1) \\
 & q_0(e) & \to & e & & p_0(e) & \to & e \\
 & q(a(x_1)) & \to & q'(x_1) & & p(a(x_1)) & \to & d(a(p'(x_1)), p(x_1)) \\
 & q(e) & \to & e & & p(e) & \to & d(e,e) \\
 & q'(a(x_1)) & \to & a(a(q(x_1))) & & p'(a(x_1)) & \to & a(p'(x_1)) \\
 & q'(e) & \to & a(e) & & p'(e) & \to & e
\end{array}
$$

Figure 4: Equivalent top-down tree transducers M and N

of different height. The rules of M and N are given in Figure 4. Let us consider the input tree $s = aaaa(x)$. We omit some parentheses in monadic input trees. We obtain

$$
\begin{aligned}
M(s) = d(M_q(aaax), M(aaax)) = d(M_{q'}(aax), d(M_q(aax), M(aax))) = \\
d(a(a(M_q(ax))), d(M_{q'}(ax), d(M_q(ax), M(ax)))) = \\
d(a(a(q'(x))), d(a(a(q(x))), d(q'(x), d(q(x), q_0(x)))))
\end{aligned}
$$

Similarly, for the transducer N we obtain

$$
\begin{aligned}
N(s) = N_p(aaax) = d(a(N_{p'}(aax)), N_p(aax)) = d(a(a(N_{p'}(ax))), d(a(N_{p'}(ax)), N_p(ax))) = \\
d(a(a(a(p'(x)))), d(a(a(p'(x))), d(a(p'(x)), p(x)))).
\end{aligned}
$$

As the reader may verify, if x is replaced by the leaf e, then indeed the output trees $M(aaaae)$ and $N(aaaae)$ are the same, i.e., the transducers are equivalent. Let us compare the trees $M(aaaax)$ and $N(aaaax)$. On the one hand, the transducer M is "ahead" of the transducer N in the output branch 2.2.2. It has already produced a d-node at that position, while N has not (and is in state p at that position). On the other hand, N is ahead of M at two other positions in the output: at the node 1.1.1 the transducer N has produced an a-node already, while M at that node is in state q', and, at node 2.2.1 the transducer M has output an a-node, while also here M is in state q'.

4 Decidable Equivalence Problems

4.1 Top-Down Tree Transducers

It was shown by Ésik [30] that the bounded height-difference of top-down tree transducers can be used to decide equivalence.

Theorem 1 ([30]) Equivalence of top-down tree transducers is decidable.

Proof. We follow the version of the proof given by Engelfriet [20]. Consider two equivalent top-

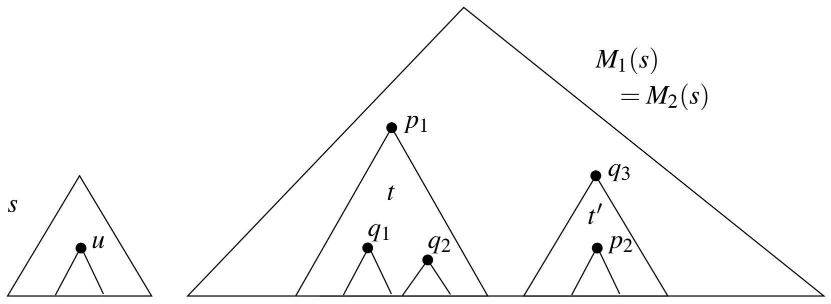


Figure 5: Two equivalent top-down tree transducers

down tree transducers M_1 and M_2. By Lemma 2 they have bounded height-balance by some constant c. Consider the trees $M_1(s[u \leftarrow x])$ and $M_2(s[u \leftarrow x])$. An "overlay" of these two trees is shown in Figure 5 (this is a copy of Figure 10 of [20]). At the node where M_2 is in state p_1, the transducer M_1 has already produced the tree t, i.e., at this node M_1 is "ahead" of M_2 by the amount t. Similarly, at the q_3-labeled node, M_2 is ahead of M_1 by the amount t'. Clearly, the height of t and t' is bounded by c. Hence, there are only finitely many such trees t and t'. We can construct a top-down tree automaton A which in its states keeps track of all such "difference trees" t and t', while simulating the runs of M_1 and M_2. It checks if the outputs are consistent, and rejects if either the outputs are different or if the height of a difference tree is too large. Finally, we check if A accepts the language $D = \mathsf{dom}(M_1) = \mathsf{dom}(M_2)$; this is decidable because D is regular by Lemma 1, and equivalence of regular tree languages is decidable (see [11]). $\square$

Note that Ésik [30] shows that even for single-valued (i.e., functional) nondeterministic top-down tree transducers, equivalence is decidable. It is open whether or not equivalence is decidable for k-valued nondeterministic top-down tree transducers (but believed to be decidable along the same lines as for bottom-up tree transducers [63], cf. the text below Theorem 5). A top-down tree transducer is letter-to-letter if the right-hand side of each rule contains exactly one output symbol in Δ. It was shown by Andre and Bossut [6] that equivalence is decidable for nondeleting *nondeterministic* letter-to-letter top-down tree transducers. An interesting generalization of Theorem 1 is given by Courcelle and Franchi-Zannettacci [15]. They show that equivalence is decidable for "separated" attribute grammars which are evaluated in two independent phases: first a phase that computes all inherited attributes, followed by a phase that computes all synthesized attributes (top-down tree transducers are the special case of synthesized attributes only).

Top-down Tree Transducers with Regular Look-Ahead. Regular look-ahead means that the transducer (MTT or T) comes with a (complete) deterministic bottom-up automaton (without final states), called the "look-ahead automaton of M". A rule of the look-ahead transducer is of the form

$$q(\sigma(x_1,\dots,x_k),\dots) \to t \quad \langle p_1,\dots,p_k\rangle$$

and is applicable to an input tree $\sigma(s_1,\dots,s_k)$ only if the look-ahead automaton recognizes s_i in state p_i for all $1 \le i \le k$. Given two top-down tree transducers with regular look-ahead M_1,M_2, we can transform them into ordinary transducers (without look-ahead) N_1,N_2 such that the resulting transducers are equivalent if and only if the original ones are. This is done by changing the input alphabet so that for every original input symbol $\sigma \in \Sigma$ of rank k, it now contains the symbols $\langle \sigma, p_1, \ldots, p_k, q_1, \ldots, q_k \rangle$ for all possible look-ahead states p_i of M_1 and q_i of M_2. Thus, for every $\sigma \in \Sigma^{(k)}$, the new input alphabet has $|P|^k|Q|^k$-many symbols. It is easy to see that Lemma 2 also holds for transducers with look-ahead, by additionally requiring that the domain D of the N_i is intersected with all input trees that represent correct runs of the look-ahead automata. Thus, Theorem 1 also holds for top-down tree transducers with look-ahead.

Theorem 2 Equivalence of top-down tree transducers with regular look-ahead is decidable.

Canonical Normal Form. Consider two equivalent top-down tree transducers M_1, M_2 and let D be their domain. As the example transducers M and N with rules in Figure 4 show, for a partial input tree $s[u \leftarrow x]$, there may be positions in the output trees where M_1 is ahead of M_2, and other positions where M_2 is ahead of M_1. Such a scenario is also depicted in Figure 5.

We say that M_1 is *earlier* than M_2, if for every $s \in D$ and $u \in V(s)$, the tree $M_2(s[u \leftarrow x])$ is a prefix of the tree $M_1(s[u \leftarrow x])$. A tree t is a prefix of a tree t' if for every $u \in V(t)$ with $t[u] \in \Delta$ it holds that $t'[u] = t[u]$. The question arises whether for every top-down translation there is an equivalent unique earliest transducer M such that $M_q \neq M_{q'}$ for $q \neq q'$; we call such a transducer a *canonical transducer*. The question was answered affirmative by Engelfriet, Maneth, and Seidl [26]. We only state this result for total transducers.

Theorem 3 Let M be a total top-down tree transducer. An equivalent canonical transducer can be constructed in polynomial time.

Proof. The canonical transducers are top-down tree transducers without an initial state, but with an *axiom tree* $A \in T_{\Delta \cup Q}(\{x_0\})$. This means that the translation on input tree $s \in T_\Sigma$ starts with the tree $A[x_0 \leftarrow s]$ (instead of $q_0(s)$ for ordinary transducers).

Starting with M, we define its axiom $A = q_0(x_0)$. In a first step, an earliest transducer is constructed: if there is a state q and an output symbol δ (of rank k) such that $\mathsf{rhs}(q,\sigma)[\varepsilon] = \delta$ for every input symbol σ, then M is *not* earliest. Intuitively, the symbol δ should be produced earlier, at each call of the state q. Thus, the construction replaces $q(x_i)$ in all right-hand sides (and in the axiom A) by $\delta(\langle q,1\rangle(x_i), \ldots, \langle q,k\rangle(x_i))$ where the $\langle q, j\rangle$ are new states. For every σ, $\mathsf{rhs}(\langle q, j\rangle, \sigma)$ is defined as the j-th subtree of the root of $\mathsf{rhs}(q,\sigma)$ – beware, this right-hand side may have changed due to the replacement above. Finally we remove q and its rules. This step is repeated until it cannot be applied anymore. In this case M has become earliest and no q and δ exists such that $M_q(s) = \delta(\ldots)$ for all input trees s of q. It should be clear that the earliest step can be carried out in polynomial time. In the second step, equivalent states are merged to obtain the canonical transducer; the corresponding equivalence relation on states is computed using fixed point iteration in cubic time (with respect to the size of M). It is computed in such a way that if $q \neq q'$, then $M_q \neq M_{q'}$. □

Note that the availability of a canonical ("minimal") transducer has many advantages. For instance, it makes possible to formulate a Myhill-Nerode like theorem which, in turn, makes possible Gold-style learning of top-down tree transducers (in polynomial time), as shown by Lemay, Maneth, and Niehren [47].

Consider the two transducers M and N with the rules given in Figure 4. To construct a canonical equivalent transducer for M according to Theorem 3, we observe that state q' of M is *not* earliest: the root equals a for the right-hand sides of all q-rules. We replace $q'(x_1)$ by $a(\langle q',1\rangle(x_1))$ in the (q,a)-rule, and introduce the two rules $\langle q',1\rangle(a(x_1)) \to a(q(x_1))$ and $\langle q',1\rangle(e) \to e$. We remove q' and have obtained an earliest transducer. The canonical transducer is constructed by realizing that the states $\langle q',1\rangle$ and q are equivalent and hence can be merged. The rules of the canonical transducer can thus be given as

$$\begin{array}{lll} q_0(a(x_1)) & \to & d(q(x_1),q_0(x_1)) \\ q_0(e) & \to & e \\ q(a(x_1)) & \to & a(q(x_1)) \\ q(e) & \to & e. \end{array}$$

To construct the canonical transducer for N, we observe that state p is not earliest: the root equals d in both rules. We thus replace $p(x_1)$ everywhere by $d(p_1(x_1),p_2(x_1))$ where p_1,p_2 are new states. After this replacement, the current (p,a)-rule is:

$$p(a(x_1)) \to d(a(p'(x_1)),d(p_1(x_1),p_2(x_1))).$$

Thus, new a-rules are $p_1(a(x_1)) \to a(p'(x_1))$ and $p_2(a(x_1)) \to d(p_1(x_1),p_2(x_1))$. The e-rules are $p_1(e) \to e$ and $p_2(e) \to e$. The resulting transducer is earliest. We now compute that $p_1 \equiv p'$ and that $p_2 \equiv p_0$. We merge these pairs of states and obtain the same transducer (up to renaming of states) as the canonical one of M above. Hence, M and N are equivalent.

As a consequence of Theorem 3 we obtain that equivalence of total top-down tree transducers can be decided in polynomial time.

Theorem 4 Equivalence of total top-down tree transducers can be decided in polynomial time.

The earliest normal form has also certain "disadvantages". For instance, it does not preserve linearity (or nondeletingness) of the transducer. Consider for $\Sigma = \{a^{(2)},e^{(0)}\}$ the rules $q(a(x_1,x_2)) \to d(q(x_1),q(x_2))$ and $q(e) \to d(e,e)$. When making earliest, these rules are removed and a rule such as $q_1(f(x_1)) \to q(x_1)$ is replaced by $q_1(f(x_1)) \to d(\langle q,1\rangle(x_1),\langle q,2\rangle(x_1))$ which is non-linear. It is also deleting: $\langle q,1\rangle(a(x_1,x_2)) \to q(x_1)$.

4.2 Bottom-Up Tree Transducers

As a corollary of Theorem 2 we obtain that also for deterministic bottom-up tree transducers, equivalence is decidable. This follows from the fact that every deterministic bottom-up tree transducer can be transformed into an equivalent deterministic top-down tree transducer with regular look-ahead [19].

Theorem 5 Equivalence of deterministic bottom-up tree transducers is decidable.

Proof. A deterministic bottom-up tree transducer is a tuple $B = (\Sigma,\Delta,Q,Q_f,R)$ where $Q_f \subseteq Q$ is the set of final states and R contains for every $k \geq 0$, $\sigma \in \Sigma^{(k)}$, and $q_1,\dots,q_k \in Q$ at most one rule of the form $\sigma(q_1(x_1),\dots,q_k(x_k)) \to q(t)$ where t is a tree in $T_\Delta(X_k)$. We construct in linear time a deterministic bottom-up tree automaton which for every rule as above has the transition $\delta_\sigma(q_1,\dots,q_k) \to q$. This automaton serves as the look-ahead automaton of a top-down tree transducer with the unique state p. For a rule as above, the transducer has the rule

$$p(\sigma(x_1,\dots,x_k)) \to t[x_i \leftarrow p(x_i) \mid i \in [k]] \quad \langle q_1,\dots,q_k\rangle.$$

It should be clear that the resulting top-down tree transducer with look-ahead T (which has only the single state p) is equivalent to the given bottom-up tree transducer B. □

The equivalence problem for bottom-up tree transducers was first solved by Zachar [70]. It was shown by Seidl [60] that equivalence can be decided in polynomial time for single-valued (i.e., functional) nondeterministic bottom-up tree transducers. Note that this also follows from Theorem 8 and the (polynomial time) construction in the proof of Theorem 5. This result was extended to finite-valued nondeterministic bottom-up tree transducers by Seidl [61]. For nondeterministic letter-to-letter bottom-up tree transducers, equivalence was shown decidable by Andre and Bossut [5]; such transducers contain exactly one output symbol in the right-hand side of each rule. They reduce the problem to the equivalence of bottom-up relabelings which was solved by Bozapalidis [9]. For deterministic bottom-up tree transducers the effective existence of a canonical normal form, similar in spirit to the earliest normal form of top-down tree transducers, was shown by Friese, Seidl, and Maneth [36]. They show that this normal form can be constructed in polynomial time, if each state of the given transducer produces either none or infinitely many outputs; hence, equivalence is decidable in polynomial time for such transducers. Friese presents in her PhD thesis [35] a Myhill-Nerode theorem for bottom-up tree transducers.

4.3 Linear Size Increase mtts

It was shown by Engelfriet and Maneth [24] that total deterministic mtts of linear size increase characterize the total deterministic MSO definable tree translations. In fact, even any composition of total deterministic mtts, when restricted to linear size increase, is equal to an MSO definable translation, as shown by Maneth [48]. The MSO definable tree translations are a special instance of the MSO definable graph translations, introduced by Courcelle and Engelfriet, see [12]. Decidability of equivalence for deterministic MSO graph-to-string translations on a context-free graph language was proved by Engelfriet and Maneth [25]. It implies decidable equivalence also for MSO tree translations. We present a proof of the latter here that only uses MTTs and avoids going through MSO.

The idea of the proof stems from Gurari's proof [41] of the decidability of equivalence for 2DGSM. In a nutshell: the ranges of all the above translations are Parikh. A language is *Parikh* if its set of Parikh vectors is equal to the set of Parikh vectors of a regular language. Let $\Sigma = \{a_1, \ldots, a_m\}$ be an alphabet. The *Parikh vector* of a string $w \in \Sigma^*$ is the n-tuple $(i_1, \ldots, i_m)$ of natural numbers i_j such that for $1 \leq j \leq m$, i_j equals the number of occurrences of a_j in w. For a language that is Parikh, it is decidable whether or not it contains a string with Parikh vector $(n, n, \ldots, n)$ for some natural number n. This property is used to prove equivalence as follows. Given two tree-to-string transducers M_1, M_2 we first change M_i to produce a new end marker \$ at the end of each output string. Then, given the regular domain language D of M_1 and M_2, and two distinct output letters a, b we construct the Parikh language

$$L^{a,b} = \{a^m b^n \mid \exists s \in D : M_1(s)/m = a, M_2(s)/n = b\}.$$

Here w/m denotes the m-th letter in the string w. We now decide if there is an n such that $a^n b^n \in L^{a,b}$, using the fact that $L^{a,b}$ is Parikh. If such an n exists, then the transducers M_1, M_2 are *not* equivalent. If, for all possible a, b, no such n exists, then we know that the transducers M_1, M_2 are equivalent.

It was shown by Engelfriet, Rozenberg, and Slutzki in Corollary 3.2.7 of [28] that ranges of *nondeterministic* finite-copying top-down tree transducers with regular look-ahead (for short, N-$\text{T}^{\text{R}}_{\text{fc}}$s) possess the Parikh property. The nondeterminism of this result is useful for defining the language $L^{a,b}$, because we need to nondeterministically choose a and b positions m and n of the output strings. A nondeterministic top-down tree transducer M is *finite-copying* if there is number c such that for every $s \in T_\Sigma$ and $u \in V(s)$, the number of occurrences of states (more precisely, subtrees $q(x)$ such that q is a state of M) in the tree $M(s[u \leftarrow x])$ is $\leq c$. We denote the class of translations of nondeterministic finite-copying top-down tree transducers by N-$\text{T}^{\text{R}}_{\text{fc}}$.

For a tree t we denote by yt its *yield*, i.e., the string of its leaf labels from left to right. For a class X of tree translations we denote by yX the corresponding class of *tree-to-yield* translations. The tree-to-yield translations of top-down tree transducers can be obtained by top-down tree-to-*string* transducers which have strings over output symbols and state calls $q(x_i)$ in the right-hand sides of their rules. We repeat the argument given in [28]. By REGT we denote the class of regular tree languages, i.e., those languages recognized by (deterministic) finite-state bottom-up tree automata.

Lemma 3 Languages in $y\text{N-T}^{\text{R}}_{\text{fc}}(\text{REGT})$ are Parikh.

Proof. Let M be a $y\text{N-T}^{\text{R}}_{\text{fc}}$ transducer and let $R \in \text{REGT}$. A top-down transducer is *linear* if no x_i appears more than once in any of the right-hand sides of its rules. We construct a linear transducer M' such that $\mathsf{dom}(M') = \mathsf{dom}(M)$ and the string $M'(s)$ is a permutation of the string $M(s)$, for every $s \in \mathsf{dom}(M)$. The new transducer computes in its states the state sequences of M, i.e., the sequence of states that are translating the current input node. Since M is finite-copying, there effectively exists a bound c on the length of the state sequences. For a new state $\langle q_1, \ldots, q_n \rangle$ with $n \leq c$ the right-hand side of a rule is obtained by simply concatenating the right-hand sides of the corresponding rules for q_i. It is well known that linear top-down tree transducers preserve regularity and hence the language $M'(R)$ is in $y\text{REGT}$, i.e., it is the yield language of a regular tree language. The latter is obviously a context-free language (cf. Theorem 3.8 of [67]) which is Parikh by Parikh's theorem [53]. □

A macro tree transducer M is *finite-copying* if there exist constants k and n such that

(1) for every input tree $s' = s[u \leftarrow x]$ with $s \in T_\Sigma$ and $u \in V(s)$, the number of occurrences of states in $M(s')$ is $\leq k$ and

(2) for every state q of rank $m+1$, $1 \leq j \leq m$, and $s \in T_\Sigma$, the number of occurrences of y_j in $M_q(s)$ is $\leq n$.

Recall that an MTT M is of linear size increase if there is a constant c such that $|\tau_M(s)| \leq c \cdot |s|$ for every $s \in T_\Sigma$. We denote the class of translations realized by MTTs of linear size increase by MTT_{lsi}.

Lemma 4 $(y\text{MTT}_{\text{lsi}}) \subseteq y\text{T}^{\text{R}}_{\text{fc}}$.

Proof. It was shown in [24] how to construct a finite-copying macro tree transducer with look-ahead, for a given macro tree transducer of linear size increase. The construction goes through several normal forms which make sure that the transducer generates only finitely many copies; most essentially, the "proper" normal form: each state produces infinitely many output trees, and, each parameter is instantiated by infinitely many trees. By using regular look-ahead finitely many different trees can be determined and outputted directly. The idea of the proper normal form was used already by Aho and Ullmann for top-down tree transducers [2].

It was shown in Lemmas 6.3 and 6.6 of [21] that M can be changed into an equivalent transducer which is "special in the parameters". This means that it is linear and nondeleting in the parameters, i.e., each parameter y_j of a state q appears *exactly once* in the right-hand side of each (q,σ)-rule. The idea is to simply provide multiple parameters, whenever parameters are copied, and to use regular look-ahead in order to determine which parameters are deleted. This was mentioned above Lemma 1 already. For a $y\text{MTT}^{\text{R}}$ transducer that is special in the parameters, it was shown in Lemma 13 of [22] how to construct an equivalent $y\text{T}^{\text{R}}$ transducer. The parameters of the yMTT can be removed by outputting the strings between them directly. Since each parameter appears once, the final string $M_q(s)$ is divided into $m+1$ string chunks w_j (where $m+1$ is the rank of q): $w_0, y_1 w_1, \ldots, y_m w_m$. We leave further details as an exercise, and suggest to start with the case that all y_j appear in strictly increasing order at the leaves of any $M_q(s)$. It is not difficult to see that the construction preserves finite-copying. □

Lemma 5 Let M_1, M_2 be $y\text{T}^{\text{R}}_{\text{fc}}$ transducers with input and output alphabets Σ and Δ, and let $a,b \in \Delta$ with $a \neq b$. Let $D \subseteq T_\Sigma$ be a regular tree language. The language $L^{a,b} = \{a^m \# b^n \mid \exists s \in D : M_1(s)/m = a, M_2(s)/n = b\}$ is Parikh.

Proof. Let us assume that the state sets Q_1, Q_2 of the transducers M_1, M_2 are disjoint. The initial state of M_1, M_2 is q_0 and p_0, respectively. We first construct a $y\text{N-T}^{\text{R}}_{\text{fc}}$ transducer M_1' such that

$$M_1'(s) = \{ua \mid u \in \Delta^*, \exists v \in \Delta^* : uav = M_1(s)\}.$$

Its state set is $Q_0 = Q_1 \cup \{q_a \mid q \in Q_1\}$ and its initial state is $q_{0,a}$. It has all rules of M_1 and, moreover, for every rule $q(\sigma(x_1,\dots,x_k)) \to w \quad \langle\cdots\rangle$ of M_1, whenever $w = uav$ it has the rule $q_a(\sigma(x_1,\dots,x_k)) \to ua \quad \langle\cdots\rangle$, and whenever $w = uq'(x_i)v$ it has the rule $q_a(\sigma(x_1,\dots,x_k)) \to uq_a'(x_i) \quad \langle\cdots\rangle$. From M_1' one obtains a $y\text{N-T}^{\text{R}}_{\text{fc}}$ transducer M_1'' such that

$$M_1''(s) = \{a^m \mid M_1(s)/m = a\}$$

by simply changing all symbols of Δ into a in the rules of M_1'. Similarly, one obtains a transducer M_2'' such that $M_2''(s) = \{b^n \mid M_2(s)/n = b\}$. Finally, a $y\text{N-T}^{\text{R}}_{\text{fc}}$ transducer M is defined such that $M(s) = \{a^m \# b^n \mid M_1(s)/m = a, M_2(s)/n = b\}$. Its state set is $\{r_0\} \cup Q_1' \cup Q_2'$ with initial state r_0. The look-ahead automaton of M is the product automaton of the look-ahead automata of M_1 and M_2. The set of rules of M is the union of those of M_1'' and M_2'', adapted to the new look-ahead appropriately. Moreover, for $\sigma \in \Sigma^{(k)}$, $k \geq 0$, and rules $q_{0,a}(\sigma(x_1,\dots,x_k)) \to u \quad \langle q_1',\dots,q_k'\rangle$ and $p_{0,b}(\sigma(x_1,\dots,x_k)) \to w \quad \langle p_1',\dots,p_k'\rangle$, we let

$$r_0(\sigma(x_1,\dots,x_k)) \to u\#w \quad \langle (q_1',p_1'),\dots,(q_k',p_k')\rangle$$

be a rule of M. Obviously, $M(s)$ equals the concatenation $M_1''(s)\#M_2''(s)$, and is finite-copying. Since $M(D) = L^{a,b}$ it follows by Lemma 3 that $L^{a,b}$ is Parikh. □

Theorem 6 *Equivalence of deterministic macro tree transducers of linear size increase is decidable.*

Proof. Let M_1, M_2 be MTT transducers of linear size increase. We first check that the domains of M_i coincide. This is decidable because $\mathsf{dom}(M_i)$ is effectively regular by Lemma 1. If not then the transducers are not equivalent and we are finished. Otherwise, let D be their domain. We may consider M_i as tree-to-string transducers, by considering the tree in the right-hand side of each rule as a string (which uses additional terminals symbols for denoting the tree structure such as opening and closing parentheses and commas). Thus, by Lemma 4 (which is effective) we may in fact assume that M_1 and M_2 are $y\text{T}^{\text{R}}_{\text{fc}}$ transducers. Let Δ be the output alphabet of M_i and let \$ be a new symbol not in Δ. We change M_i so that each output string is followed by the \$ symbol. This can easily be done by first splitting the initial state q_0 so that it appears in the right-hand side of no rule, and then adding \$ to the end of each q_0-rule. It now holds that M_1 and M_2 are *not equivalent* if and only if there exist $a,b \in \Delta$ with $a \neq b$, $s \in D$, and a number n such that $M_1(s)/n = a$ and $M_2(s)/n = b$. The latter holds if the intersection of $L^{a,b}$ of Lemma 5 with the language $E = \{a^n \# b^n \mid n \in \mathbb{N}\}$ is nonempty. Since $L^{a,b}$ is Parikh by Lemma 5, we obtain decidability because semilinear sets are closed under intersection [39, 38] and have decidable emptiness. But, there is a much easier proof: $E \cap L$ is context-free, because E is (by the well-known "triple construction", see, e.g., Theorem 6.5 of [44]), where L is a regular language with the same Parikh vectors as $L^{a,b}$. The result follows since context-free grammars have decidable emptiness. □

4.4 Monadic mtts

Recall that a macro tree transducer is monadic if both its input and output alphabet are monadic, i.e., consist of symbols of rank one and rank zero only. We will reduce the equivalence problem for monadic MTT transducers to the sequence equivalence problem of HDT0L systems. An MTT is *nondeleting* if for every state q of rank $m+1$, $1 \leq j \leq m$, and input symbol σ, the parameter y_j occurs in $\mathsf{rhs}(q,\sigma)$. A monadic MTT $M = (Q, \Sigma, \Delta, q_0, R)$ is *normalized* if

(N0) it is nondeleting

(N1) each state is of rank two or one, i.e., $Q = Q^{(2)} \cup Q^{(1)}$ and

(N2) there is only one input and output symbol of rank zero, i.e., $\Sigma^{(0)} = \Delta^{(0)} = \{\perp\}$

Note that for total transducers (N1) is a consequence of (N0) because a $(q,\perp)$-rule can only contain at most one parameter occurrence.

HDT0L systems An instance of the HDT0L sequence equivalence problem consists of finite alphabets Σ and Δ, two strings $w_1, w_2 \in \Sigma^*$, homomorphisms $h_j, g_j : \Sigma^* \to \Sigma^*$, $1 \leq j \leq n$, and homomorphisms $h, g : \Sigma^* \to \Delta^*$. To solve the problem we have to determine whether or not

$$h(h_{i_k}(\cdots h_{i_1}(w_1)\cdots)) = g(g_{i_k}(\cdots g_{i_1}(w_2)\cdots))$$

holds true for all $k \geq 0$, $1 \leq i_1, \ldots, i_k \leq n$. This problem is known to be decidable. It was first proven by Culik II and Karhumäki [18], using Ehrenfeucht's Conjecture and Makanin's algorithm. A later proof of Ruohonen [59] is based on the theory of metabelian groups. Yet another, very short, proof was given by Honkala [43] which only relies on Hilbert's Basis Theorem. We now show that the equivalence problem for total monadic MTT transducers can be reduced to the sequence equivalence problem for HDT0L systems. For a monadic tree $s = a_1(\cdots a_n(e)\cdots)$ we denote by $\mathsf{strip}(s)$ the string $a_1 \cdots a_n$.

Lemma 6 *Equivalence of total monadic normalized* MTT*s on a regular input language is decidable.*

Proof. We first solve the problem without a given input tree language. Let $M_1 = (Q_1, \Gamma, \Pi, q_0, R_1)$ and $M_2 = (Q_2, \Gamma, \Pi, p_0, R_2)$ be total monadic normalized macro tree transducers such that Q_1 is disjoint from Q_2. Let $Q = Q_1 \cup Q_2$. We define an instance of the HDT0L sequence equivalence problem. The string alphabets Σ, Δ are defined as $\Sigma = \Pi^{(1)} \cup Q$ and $\Delta = \Pi^{(1)}$. We define homomorphisms h_a, g_a for every input symbol $a \in \Gamma^{(1)}$. For $\pi \in \Pi^{(1)}$ let $h_a(\pi) = g_a(\pi) = \pi$. Let $q \in Q$. If $q \in Q_1$ then let $h_a(q) = \mathsf{strip}(\mathsf{rhs}_{M_1}(q,a))$, and otherwise let $h_a(q) = q$. If $q \in Q_2$ then let $g_a(q) = \mathsf{strip}(\mathsf{rhs}_{M_2}(q,a))$, and otherwise let $g_a(q) = q$. For trees $t \in T_{\Pi \cup Q}(X_k \cup Y_m)$ we define the mapping strip by $\mathsf{strip}(\pi(t)) = \pi \cdot \mathsf{strip}(t)$ for $\pi \in \Gamma^{(1)}$, $\mathsf{strip}(q(x_1,t)) = q \cdot \mathsf{strip}(t)$ for $q \in Q^{(2)}$, $\mathsf{strip}(q(x_1)) = q$ for $q \in Q^{(1)}$, and $\mathsf{strip}(\perp) = \mathsf{strip}(y_1) = \varepsilon$, where "$\cdot$" denotes string concatenation. The final homomorphisms h, g are defined as $h(q) = \mathsf{strip}(\mathsf{rhs}_{M_1}(q,\perp))$ if $q \in Q_1$, and otherwise $h(q) = q$, and $g(q) = \mathsf{strip}(\mathsf{rhs}_{M_2}(q,\perp))$ if $q \in Q_2$, and otherwise $g(q) = q$. Last but not least, let $w_1 = q_0$ and $w_2 = p_0$. This ends the construction of the HDT0L instance. Consider an input tree $s = a_1(\cdots a_n(\perp)\cdots) \in T_\Gamma$. It should be clear that $h(h_{a_n}(\cdots h_{a_1}(w_1)\cdots)) = \mathsf{strip}(M_1(s))$ and that $g(g_{a_n}(\cdots g_{a_1}(w_2)\cdots)) = \mathsf{strip}(M_2(s))$. Thus, this instance of the HDT0L sequence equivalence problem solves the equivalence problem of the two transducers M_1 and M_2.

Let $D \subseteq T_\Gamma$ be a regular input tree language. We wish to decide whether $M_1(s) = M_2(s)$ for every $s \in D$. We assume that D is given by a deterministic finite-state automaton A that runs top-down on the unary symbols in $\Gamma^{(1)}$. We further assume that $A = (R, \Gamma^{(1)}, r_0, \delta, R_f)$ is complete, i.e., for every state $r \in R$ and every symbol $a \in \Gamma^{(1)}$, $\delta(r,a)$ is defined (and in R). Note that r_0 is the initial state and

$R_f \subseteq R$ is the set of final states. Let $\Sigma = \Pi^{(1)} \cup Q$ as before and define $\Sigma' = \{\langle r,b\rangle \mid r \in R, b \in \Sigma\}$ and $\Delta = \Pi^{(1)}$. Our HDT0L instance is over Σ' and Δ. Let $a \in \Gamma^{(1)}$, $r \in R$, and $r' = \delta(r,a)$. For $\pi \in \Pi^{(1)}$ let $h_a(\langle r,\pi\rangle) = g_a(\langle r,\pi\rangle) = \langle r',\pi\rangle$. Let $q \in Q_1$ and $p \in Q_2$. Define

$$\begin{aligned} h_a(\langle r,q\rangle) &= \mathsf{strip}(\mathsf{rhs}_{M_1}(q,a))[b \leftarrow \langle r',b\rangle \mid b \in \Sigma] \\ g_a(\langle r,p\rangle) &= \mathsf{strip}(\mathsf{rhs}_{M_2}(p,a))[b \leftarrow \langle r',b\rangle \mid b \in \Sigma]. \end{aligned}$$

Let $h_a(\langle r,p\rangle) = \langle r,p\rangle$ and $g_a(\langle r,q\rangle) = \langle r,q\rangle$. The final homomorphisms g,h are defined as follows. If $r \in R_f$ then let $h(\langle r,q\rangle) = \mathsf{strip}(\mathsf{rhs}_{M_1}(q,\perp))$, $g(\langle r,p\rangle) = \mathsf{strip}(\mathsf{rhs}_{M_2}(p,\perp))$, and let $h(\langle r,b\rangle) = b$ and $g(\langle r,b\rangle) = b$ for the remaining cases. If $r \notin R_f$ then let $h(\langle r,b\rangle) = g(\langle r,b\rangle) = \varepsilon$ for every $b \in \Sigma$. The initial strings are defined as $w_1 = \langle r_0,q_0\rangle$ and $w_2 = \langle r_0,p_0\rangle$.

Consider an input tree $s = a_1(\cdots a_n(\perp)\cdots) \in T_\Gamma$ and let $1 \le j \le n$. It should be clear that if $\delta^*(r_0,a_1\cdots a_j) = r$, i.e., A arrives in state r after reading the prefix $a_1\cdots a_j$, then

$$h_{a_j}(\cdots h_{a_1}(w_1)\cdots) = \mathsf{strip}(M_1(a_1\cdots a_j(x)))[\pi \leftarrow \langle r,\pi\rangle \mid \pi \in \Delta][q \leftarrow \langle r,q\rangle \mid q \in Q_1]$$

and similarly for g and M_2. Thus each and every symbol of a sentential form is labeled by the current state of the automaton A. Hence, if $s \notin D$, then every symbol in $u_1 = h_{a_n}(\cdots h_{a_1}(w_1)\cdots)$ and in $u_2 = g_{a_n}(\cdots g_{a_1}(w_2)\cdots)$ is labeled by some state $r \notin R_f$. This implies that $h(u_1) = g(u_2) = \varepsilon$, i.e., the final strings are equal whenever $s \notin D$. If on the contrary $s \in D$ then every symbol in u_i is labeled by a final state and therefore $h(u_i) = \mathsf{strip}(M_i(s))$ as before. □

Input and output symbols of rank zero of a given transducer become symbols of rank one in the corresponding normalized transducer. For a monadic tree $t = a_1(\cdots a_n(e)\cdots)$ we denote by $\mathsf{expand}(t)$ the tree $a_1(\cdots a_n(e(\perp))\cdots)$.

Lemma 7 *For every monadic* MTT^R *transducer M a normalized* MTT^R *transducer N can be constructed such that $\tau_N = \{(\mathsf{expand}(s),\mathsf{expand}(t)) \mid (s,t) \in \tau_M\}$.*

Proof. Using regular look-ahead we first make M nondeleting. As mentioned in the proof of Lemma 4, this construction was given in the proof of Lemma 6.6 of [21]. Now, every parameter that appears in the left-hand side of a rule, also appears in the right-hand side. Since the final output tree is monadic, the resulting transducer satisfies (N1) above. Finally, we define the MTT^R transducer N which has input and output alphabets $\Sigma' = \Sigma^{(1)} \cup \{a'^{(1)} \mid a \in \Sigma^{(0)}\}$ and $\Delta' = \Delta^{(1)} \cup \{a'^{(1)} \mid a \in \Delta^{(0)}\}$. For input symbols in $\Sigma^{(1)}$ the transducer N has exactly the same rules as M. Let $q \in Q$ and $a \in \Sigma^{(0)}$ such that $\mathsf{rhs}(q,a)$ is defined. Then we let

$$q(a'(x_1)) \to \mathsf{rhs}(q,a)[b \leftarrow b'(\perp) \mid b \in \Delta^{(0)}].$$

be a rule of N. Regular look-ahead can be used to ensure that only trees of the form $\mathsf{expand}(s)$ are in the domain of N. □

Obviously, two monadic MTT transducers are equivalent if and only if their normalized versions are equivalent. Hence, it suffices to consider the equivalence problem of normalized monadic MTT transducers.

Theorem 7 *Equivalence of monadic macro tree transducers with regular look-ahead is decidable.*

Proof. Let M_1,M_2 be monadic macro tree transducers with regular look-ahead and let Σ be their input alphabet. Let A_1,A_2 be the look-ahead automata of M_1,M_2. By Lemma 7 we may assume that

M_1 and M_2 are normalized. We first check if the domains of M_1 and M_2 coincide. If not then the transducers are not equivalent and we are finished. Otherwise, let D be their domain. We define two total monadic MTTs N_1, N_2 *without* look-ahead. Let P_1, P_2 be the sets of states of A_1, A_2, respectively. The input alphabet of N_i is defined as $\Sigma' = \{\langle\sigma, p_1, p_2\rangle \mid \sigma \in \Sigma^{(1)}, p_1 \in P_1, p_2 \in P_2\}$. An input symbol $\langle\sigma, p_1, p_2\rangle$ denotes that the look-ahead automata at the child of the current node are in states p_1 and p_2, respectively. Thus, the $(q, \langle\sigma, p_1, p_2\rangle)$-rule of N_1 is defined as the (q, σ)-rule with look-ahead $\langle p_1\rangle$ of M_1, and the $(q, \langle\sigma, p_1, p_2\rangle)$-rule of N_2 is defined as the (q, σ)-rule with look-ahead $\langle p_2\rangle$ of M_2. Finally, we make N_1 and N_2 total (in some arbitrary way).

For a tree t in $T_{\Sigma'}$ we denote by $\gamma(t)$ the tree in T_Σ obtained by changing every label $\langle\sigma, p, p'\rangle$ into the label σ. Let $E \subseteq T_{\Sigma'}$ be the regular tree language consisting of all trees t such that

(1) $s = \gamma(t)$ is in D,

(2) the second components of the labels in t constitute a correct run of A_1 on s, and

(3) the third components of the labels in t constitute a correct run of A_2 on s.

Clearly, for the resulting transducers N_i it holds that N_1 and N_2 are equivalent on E if and only if M_1 is equivalent to M_2. Hence decidability of equivalence follows from Lemma 6. □

Note that macro tree transducers with monadic output alphabet are essentially the same as top-down tree-to-string transducers (see Lemma 7.6 of [21]). For the latter, the equivalence problem was stated already in 1980 by Engelfriet [20] as a big open problem. This problem remains open, but, as this section has shown, at least for the restricted case of monadic input, we obtain decidability. Note further that the connection between L-systems and tree transducers is well known and was studied extensively in [28].

5 Complexity

In Section 4 we already mentioned one complexity result, viz. Theorem 4, which states that equivalence can be decided in polynomial time for total top-down tree transducers. How about top-down tree transducers (TS) in general? It was mentioned in the Conclusions of [7] that checking equivalence of TS can be done in double exponential time, using the procedure of [26].

Without giving details we now present a proof that strengthens both results above (and which also works for transducers with look-ahead). We show that equivalence for TS can be decided in EXPSPACE, and for total TS in NLOGSPACE. For a top-down tree transducer M and trees s, t with $t = M(s)$, it holds that each node v in the output tree t is produced by one particular node u in s. The latter is called v's origin. It means that $M(s[u \leftarrow x])$ does not have a Δ-node v, while v is a Δ node in $M[s \leftarrow a(x, \ldots, x)]$ where $a = s[u]$.

Theorem 8 Equivalence of top-down tree transducers with regular look-ahead is decidable in EXPSPACE, and for total transducers in NLOGSPACE.

Proof. We sketch the proof for transducers without look-ahead. Since both complexity classes are closed under complement, it suffices to consider nonequivalence. Consider two top-down tree transducers M_1 and M_2. The idea (as in the finite-copying case) is to guess (part of) an input tree s and a node v of the output trees $t_1 = M_1(s)$ and $t_2 = M_2(s)$ such that $t_1[v] \neq t_2[v]$. It suffices to guess the two *origins* of v with respect to M_1 and M_2: nodes u_1 and u_2 of s, respectively. More precisely, it suffices to guess the paths from the root of s to u_1 and u_2, and the path from the root of t_1 and t_2 to v, where we may assume that all proper ancestors of v have the same label in t_1 and t_2. When guessing the path from the root of s to the least common ancestor of u_1 and u_2, the path in t_1 can be ahead of the path in t_2, or vice versa, so the difference between these paths must be stored. But it suffices to keep the length of this difference to be at most exponential in the sizes of M_1 and M_2, due to the bounded height-balance of M_1 and M_2 in case they are equivalent. In the proof of Lemma 2 the height of the smallest tree s' is at most exponential, and hence the height of its translation is at most exponential. Hence the difference between the paths in t_1 and t_2 can be stored in exponential space.

If M_1 and M_2 are total, then the difference between the paths in t_1 and t_2 is at most a path in a right-hand side of a rule, which can be kept in logarithmic space. Logarithmic space is also needed to do all the guesses, of course. The same proof as above also holds for transducers with regular look-ahead. □

Theorem 9 *Equivalence of top-down tree transducers is* EXPTIME*-hard.*

Proof. It is well known that testing intersection emptiness of n deterministic top-down tree automata $A_1,\dots,A_n$ is EXPTIME-complete. This was shown by Seidl [62], cf. also [11]. Let Σ be the ranked alphabet of the A_i. We define the top-down tree transducer $M_1 = (\{q_0,\dots,q_n\},\Sigma,\Sigma\cup\{\delta^{(n)}\},q_0,R)$. We consider each A_i as a partial identity transducer with start state q_i, and add the corresponding rules to R. Thus, $M_{1,q_i} = \{(s,s) \mid s \in L(A_i)\}$. Let $\sigma \in \Sigma$ be an arbitrary symbol of rank ≥ 1, and let e be an arbitrary symbol in $\Sigma^{(0)}$. We add these two rules to R:

$$\begin{array}{lll} q_0(\sigma(x_1,\dots)) & \to & \delta(q_1(x_1),q_2(x_1),\dots,q_n(x_1)) \\ q_0(e) & \to & e \end{array}$$

The transducer $M_2 = (\{p\},\Sigma,\Sigma,p,\{p(e)\to e\})$ realizes the translation $\tau_{M_2} = \{(e,e)\}$. If the intersection of the $L(A_i)$ is empty, then there is no tree $s \in T_\Sigma$ such that $M_{q_i}(s)$ is defined for all $i \in \{1,\dots,n\}$, i.e., the first rule displayed above is never applicable. Hence, in this case also $\tau_{M_1} = \{(e,e)\}$, i.e., the transducers M_1,M_2 are equivalent. If the intersection is non-empty, then there is an input tree s such that $M(s) = \delta(s,s,\dots,s)$. Thus, M_1 is equivalent to M_2 if and only if the intersection of the $L(A_i)$ is empty. □

5.1 Streaming Tree Transducers

The (deterministic) streaming tree transducers of Alur and d'Antoni are a new model with the same expressive power as deterministic MSO tree translations which in turn realize the same translations as deterministic macro tree translations of linear size increase. The idea of the model is to use a finite set of variables which hold partial outputs. These variables are updated during a single depth-first left-to-right traversal of the input tree. It is stated in Theorem 20 of [3] that equivalence of streaming tree transducers can be decided in exponential time. The idea of the proof is the same as the one in Theorem 6: construct a context-free language $L^{a,b}$ and use its Parikhness to check if a^nb^n is in the language. For them, $L^{a,b}$ is represented by a pushdown automaton A, the number of states of which is exponential in the number of variables of the given streaming tree transducer. They mention that checking if a^nb^n is in $L(A)$ can be done in NPTIME using [31, 64].

Theorem 10 ([3]) *Equivalence of streaming tree transducers is decidable in* CO-NEXPTIME.

For the transducers that map strings to nested strings, that is, for streaming string-to-tree transducers their construction yields a PSPACE bound (Theorem 21 of [3]).

Theorem 11 ([3]) *Equivalence of streaming string-to-tree transducers is decidable in* PSPACE.

5.2 Visibly Pushdown Transducers

Visibly pushdown languages were defined by Alur and Madhusudan [4] as a particular subclass of the context-free languages. In fact, they are just regular tree languages in disguise. Visibly pushdown transducers were introduced by Raskin and Servais [56]. They translate well-nested input strings into strings, during one left-to-right traversal of the input. If the output strings are nested as well, then they describe tree transformations. The expressive power of the resulting tree transformations is investigated by Caralp, Filiot, Reynier, Servais, and Talbot [10]. Such transducers cannot copy nor swap the order of input trees. Thus, they are MSO definable. But they are incomparable to the top-down or bottom-up tree translations, because they can translate a tree into its yield (string of leaf labels from left to right).

Theorem 12 ([32]) *Equivalence of functional visibly pushdown transducers is* EXPTIME-*complete. For total such transducers the problem is in* PTIME.

The EXPTIME-completeness result extends to the case of regular look-ahead, as shown in Section 8.4 of [33, 65]. Staworko, Laurence, Lemay, and Niehren [66] have considered the equivalence problem for deterministic visibly pushdown transducers and show that it can be reduced in PTIME to the homomorphic equivalence problem on context-free grammars. The latter was shown by Plandowski [45, 55] to be solvable in PTIME. They show in [66] that for several related classes the problem is in PTIME, for instance, linear and order-preserving deterministic top-down and bottom-up tree transducers.

Theorem 13 ([66]) *Equivalence of deterministic visibly pushdown transducers is decidable in* PTIME.

6 Conclusion

We discussed the decidability of equivalence for three incomparable subclasses of deterministic macro tree transducers: top-down tree transducers, linear size increase MTTs, and monadic MTTs. For top-down tree transducers the proof either uses its bounded height-balance property and constructs an automaton that keeps track of the balance. Alternatively, such transducers may be transformed into their canonical normal form and then be checked for isomorphism. For these decision procedures it is not "harmful" that a top-down tree transducer can copy a lot and be of exponential size increase, because the multiple copies of equivalent transducers must be well-nested into each other (cf. Figure 5). This nesting property is not present for MTTs, and in particular the bounded height-balance does not hold for MTTs, even not for monadic ones. Thus, other techniques are needed in these two cases. For the linear size increase subclass of MTTs we may use the Parikh property of the corresponding output languages: the two transducers are merged ("twinned") to output $a^m b^n$ if, on the same input, one transducer produces at position m of its output the letter a while the other transducer produces at position n the letter b. Since this output language is Parikh, we may decide if it contains $a^n b^n$ which implies that the transducers are not equivalent (because $a \neq b$). For monadic MTTs we use yet another technique: we simulate the transducers by HDT0L sequences. Since the sequence equivalence problem for HDT0L systems is decidable (not detailed here), the result follows. It remains a deep open problem whether or not equivalence is decidable for arbitrary deterministic macro tree transducers. Even for MTTs with monadic output, which are the same as deterministic top-down tree-to-string transducers, it is open whether or not equivalence is decidable. Note that the availability of a canonical normal form is a much stronger result than the decidability of equivalence: for instance, equivalence is easily decided for top-down transducers with look-ahead, but, for such transducers we only know a canonical normal form in the total case for a fixed look-ahead automaton [27]. In fact, even to decide whether or not a given T$^{\mathrm{R}}$ is equivalent to a T is a difficult open problem; it was solved recently for a subclass of T$^{\mathrm{R}}$s [27].

References

[1] A. V. Aho (1968): *Indexed Grammars - An Extension of Context-Free Grammars*. J. ACM 15(4), pp. 647–671, doi:10.1145/321479.321488.

[2] A. V. Aho & J. D. Ullman (1971): *Translations on a Context-Free Grammar*. Information and Control 19(5), pp. 439–475, doi:10.1016/S0019-9958(71)90706-6.

[3] R. Alur & L. D'Antoni (2011): *Streaming Tree Transducers*. CoRR abs/1104.2599.

[4] R. Alur & P. Madhusudan (2004): *Visibly pushdown languages*. In: STOC, pp. 202–211, doi:10.1145/1007352.1007390.

[5] Y. Andre & F. Bossut (1995): *The Equivalence Problem for Letter-to-Letter Bottom-up Tree Transducers is Solvable*. In: TAPSOFT, pp. 155–171, doi:10.1007/3-540-59293-8_193.

[6] Y. Andre & F. Bossut (1998): *On the Equivalence Problem for Letter-to-Letter Top-Down Tree Transducers*. Theor. Comput. Sci. 205(1-2), pp. 207–229, doi:10.1016/S0304-3975(97)00080-7.

[7] M. Benedikt, J. Engelfriet & S. Maneth (2013): *Determinacy and Rewriting of Top-Down and MSO Tree Transformations*. In: MFCS, pp. 146–158, doi:10.1007/978-3-642-40313-2_15.

[8] J. Berstel (1979): *Transductions and context-free languages*. Teubner, Stuttgart.

[9] S. Bozapalidis (1992): *Alphabetic Tree Relations*. Theor. Comput. Sci. 99(2), pp. 177–211, doi:10.1016/0304-3975(92)90348-J.

[10] M. Caralp, E. Filiot, P.-A. Reynier, F. Servais & J.-M. Talbot (2013): *Expressiveness of Visibly Pushdown Transducers*. In: TTATT, pp. 17–26, doi:10.4204/EPTCS.134.3.

[11] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, C. Löding, D. Lugiez, S. Tison & M. Tommasi (2007): *Tree Automata Techniques and Applications*. Available at: http://www.grappa.univ-lille3.fr/tata.

[12] B. Courcelle & J. Engelfriet (2012): *Graph Structure and Monadic Second-Order Logic - A Language-Theoretic Approach*. Encyclopedia of mathematics and its applications 138, Cambridge University Press, doi:10.1017/CBO9780511977619.

[13] B. Courcelle & P. Franchi-Zannettacci (1982): *Attribute Grammars and Recursive Program Schemes I*. Theor. Comput. Sci. 17, pp. 163–191, doi:10.1016/0304-3975(82)90003-2.

[14] B. Courcelle & P. Franchi-Zannettacci (1982): *Attribute Grammars and Recursive Program Schemes II*. Theor. Comput. Sci. 17, pp. 235–257, doi:10.1016/0304-3975(82)90024-X.

[15] B. Courcelle & P. Franchi-Zannettacci (1982): *On the Equivalence Problem for Attribute Systems*. Information and Control 52(3), pp. 275–305, doi:10.1016/S0019-9958(82)90786-0.

[16] K. Culik II (1976): *On the Decidability of the Sequence Equivalence Problem for D0L-Systems*. Theor. Comput. Sci. 3(1), pp. 75–84, doi:10.1016/0304-3975(76)90066-9.

[17] K. Culik II & I. Fris (1977): *The Decidability of the Equivalence Problem for D0L-Systems*. Information and Control 35(1), pp. 20–39, doi:10.1016/S0019-9958(77)90512-5.

[18] K. Culik II & J. Karhumäki (1986): *A new proof for the D0L Sequence Equivalence Problem and its implications*, doi:10.1007/978-3-642-95486-3_5. In G. Rozenberg & A. Salomaa, editors: The book of L, Springer, Berlin, pp. 63–74.

[19] J. Engelfriet (1977): *Top-down Tree Transducers with Regular Look-ahead*. Mathematical Systems Theory 10, pp. 289–303, doi:10.1007/BF01683280.

[20] J. Engelfriet (1980): *Some open questions and recent results on tree transducers and tree languages*. In R. V. Book, editor: Formal Language Theory; Perspectives and Open Problems, Academic Press, New York.

[21] J. Engelfriet & S. Maneth (1999): *Macro Tree Transducers, Attribute Grammars, and MSO Definable Tree Translations*. Inf. Comput. 154(1), pp. 34–91, doi:10.1137/S0097539701394511.

[22] J. Engelfriet & S. Maneth (2002): *Output String Languages of Compositions of Deterministic Macro Tree Transducers*. J. Comput. Syst. Sci. 64(2), pp. 350–395, doi:10.1006/jcss.2001.1816.

[23] J. Engelfriet & S. Maneth (2003): *A comparison of pebble tree transducers with macro tree transducers*. Acta Inf. 39(9), pp. 613–698, doi:10.1007/s00236-003-0120-0.

[24] J. Engelfriet & S. Maneth (2003): *Macro Tree Translations of Linear Size Increase are MSO Definable*. SIAM J. Comput. 32(4), pp. 950–1006, doi:10.1137/S0097539701394511.

[25] J. Engelfriet & S. Maneth (2006): *The equivalence problem for deterministic MSO tree transducers is decidable*. Inf. Process. Lett. 100(5), pp. 206–212, doi:10.1016/j.ipl.2006.05.015.

[26] J. Engelfriet, S. Maneth & H. Seidl (2009): *Deciding equivalence of top-down XML transformations in polynomial time*. J. Comput. Syst. Sci. 75(5), pp. 271–286, doi:10.1016/j.jcss.2009.01.001.

[27] J. Engelfriet, S. Maneth & H. Seidl (2013): *Look-Ahead Removal for Top-Down Tree Transducers*. CoRR abs/1311.2400.

[28] J. Engelfriet, G. Rozenberg & G. Slutzki (1980): *Tree Transducers, L Systems, and Two-Way Machines*. J. Comput. Syst. Sci. 20(2), pp. 150–202, doi:10.1016/0022-0000(80)90058-6.

[29] J. Engelfriet & H. Vogler (1985): *Macro Tree Transducers*. J. Comput. Syst. Sci. 31(1), pp. 71–146, doi:10.1016/0022-0000(85)90066-2.

[30] Z. Ésik (1981): *Decidability results concerning tree transducers I*. Acta Cybern. 5(1), pp. 1–20.

[31] J. Esparza (1997): *Petri Nets, Commutative Context-Free Grammars, and Basic Parallel Processes*. Fundam. Inform. 31(1), pp. 13–25, doi:10.3233/FI-1997-3112.

[32] E. Filiot, J.-F. Raskin, P.-A. Reynier, F. Servais & J.-M. Talbot (2010): *Properties of Visibly Pushdown Transducers*. In: MFCS, pp. 355–367, doi:10.1007/978-3-642-15155-2_32.

[33] E. Filiot & F. Servais (2012): *Visibly Pushdown Transducers with Look-Ahead*. In: SOFSEM, pp. 251–263, doi:10.1007/978-3-642-27660-6_21.

[34] M. J. Fischer (1968): *Grammars with Marcro-like Productions*. Ph.D. thesis, Harvard University.

[35] S. Friese (2011): *On Normalization and Type Checking for Tree Transducers*. Ph.D. thesis, Institut für Informatik, Technische Universität München. Available at http://mediatum.ub.tum.de/doc/1078090/1078090.pdf.

[36] S. Friese, H. Seidl & S. Maneth (2011): *Earliest Normal Form and Minimization for Bottom-up Tree Transducers*. Int. J. Found. Comput. Sci. 22(7), pp. 1607–1623, doi:10.1142/S012905411100891X.

[37] Z. Fülöp & H. Vogler (1998): *Syntax-Directed Semantics - Formal Models Based on Tree Transducers*. Monographs in Theoretical Computer Science. An EATCS Series, Springer, doi:10.1007/978-3-642-72248-6.

[38] S. Ginsburg (1966): *The Mathematical Theory of Context-Free Languages*. McGraw-Hill.

[39] S. Ginsburg & E. H. Spanier (1964): *Bounded ALGOL-like languages*. Trans. Amer. Math. Soc 113, pp. 333–368, doi:10.2307/1994067.

[40] T. V. Griffiths (1968): *The Unsolvability of the Equivalence Problem for Lambda-Free Nondeterministic Generalized Machines*. J. ACM 15(3), pp. 409–413, doi:10.1145/321466.321473.

[41] E. M. Gurari (1982): *The Equivalence Problem for Deterministic Two-Way Sequential Transducers is Decidable*. SIAM J. Comput. 11(3), pp. 448–452, doi:10.1137/0211035.

[42] S. Hakuta, S. Maneth, K. Nakano & H. Iwasaki (2014): *XQuery Streaming by Forest Transducers*. In: ICDE, pp. 417–428.

[43] J. Honkala (2000): *A short solution for the HDT0L sequence equivalence problem*. Theor. Comput. Sci. 244(1-2), pp. 267–270, doi:10.1016/S0304-3975(00)00158-4.

[44] J. E. Hopcroft & J. D. Ullman (1979): *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley.

[45] J. Karhumäki, W. Plandowski & W. Rytter (1995): *Polynomial Size Test Sets for Context-Free Languages*. J. Comput. Syst. Sci. 50(1), pp. 11–19, doi:10.1006/jcss.1995.1002.

[46] D. E. Knuth (1968): *Semantics of Context-Free Languages*. Mathematical Systems Theory 2(2), pp. 127–145, doi:10.1007/BF01692511.

[47] A. Lemay, S. Maneth & J. Niehren (2010): *A learning algorithm for top-down XML transformations*. In: *PODS*, pp. 285–296, doi:10.1145/1807085.1807122.

[48] S. Maneth (2003): *The Macro Tree Transducer Hierarchy Collapses for Functions of Linear Size Increase*. In: *FSTTCS*, pp. 326–337, doi:10.1007/978-3-540-24597-1_28.

[49] S. Maneth, A. Berlea, T. Perst & H. Seidl (2005): *XML type checking with macro tree transducers*. In: *PODS*, pp. 283–294, doi:10.1145/1065167.1065203.

[50] S. Maneth, T. Perst & H. Seidl (2007): *Exact XML Type Checking in Polynomial Time*. In: *ICDT*, pp. 254–268, doi:10.1007/11965893_18.

[51] T. Milo, D. Suciu & V. Vianu (2003): *Typechecking for XML transformers*. *J. Comput. Syst. Sci.* 66(1), pp. 66–97, doi:10.1016/S0022-0000(02)00030-2.

[52] K. Nakano & S.-C. Mu (2006): *A Pushdown Machine for Recursive XML Processing*. In: *APLAS*, pp. 340–356, doi:10.1007/11924661_21.

[53] R. Parikh (1966): *On Context-Free Languages*. *J. ACM* 13(4), pp. 570–581, doi:10.1145/321356.321364.

[54] T. Perst & H. Seidl (2004): *Macro forest transducers*. *Inf. Process. Lett.* 89(3), pp. 141–149, doi:10.1016/j.ipl.2003.05.001.

[55] W. Plandowski (1994): *Testing Equivalence of Morphisms on Context-Free Languages*. In: *ESA*, pp. 460–470.

[56] J.-F. Raskin & F. Servais (2008): *Visibly Pushdown Transducers*. In: *ICALP (2)*, pp. 386–397, doi:10.1007/978-3-540-70583-3_32.

[57] W. C. Rounds (1969): *Context-Free Grammars on Trees*. In: *STOC*, pp. 143–148, doi:10.1145/800169.805428.

[58] W. C. Rounds (1970): *Mappings and Grammars on Trees*. *Mathematical Systems Theory* 4(3), pp. 257–287, doi:10.1007/BF01695769.

[59] K. Ruohonen (1986): *Equivalence problems for regular sets of word morphisms*, doi:10.1007/978-3-642-95486-3_33. In G. Rozenberg & A. Salomaa, editors: *The book of L*, Springer, Berlin, pp. 393–401.

[60] H. Seidl (1992): *Single-Valuedness of Tree Transducers is Decidable in Polynomial Time*. *Theor. Comput. Sci.* 106(1), pp. 135–181, doi:10.1016/0304-3975(92)90281-J.

[61] H. Seidl (1994): *Equivalence of Finite-Valued Tree Transducers Is Decidable*. *Mathematical Systems Theory* 27(4), pp. 285–346, doi:10.1007/BF01192143.

[62] H. Seidl (1994): *Haskell Overloading is DEXPTIME-Complete*. *Inf. Process. Lett.* 52(2), pp. 57–60, doi:10.1016/0020-0190(94)00130-8.

[63] H. Seidl (2014): *Private Communication*.

[64] H. Seidl, T. Schwentick, A. Muscholl & P. Habermehl (2004): *Counting in Trees for Free*. In: *ICALP*, pp. 1136–1149, doi:10.1007/978-3-540-27836-8_94.

[65] F. Servais (2011): *Visibly Pushdown Transducers*. Ph.D. thesis, Université Libre de Bruxelles.

[66] S. Staworko, G. Laurence, A. Lemay & J. Niehren (2009): *Equivalence of Deterministic Nested Word to Word Transducers*. In: *FCT*, pp. 310–322, doi:10.1007/978-3-642-03409-1_28.

[67] J. W. Thatcher (1970): *Generalized Sequential Machine Maps*. *J. Comput. Syst. Sci.* 4(4), pp. 339–367, doi:10.1016/S0022-0000(70)80017-4.

[68] H. Vogler (1991): *Functional Description of the Contextual Analysis in Block-Structured Programming Languages: A Case Study of Tree Transducers*. *Sci. Comput. Program.* 16(3), pp. 251–275, doi:10.1016/0167-6423(91)90009-M.

[69] J. Voigtländer (2005): *Tree transducer composition as program transformation*. Ph.D. thesis, Technical University Dresden.

[70] Z. Zachar (1979): *The solvability of the equivalence problem for deterministic frontier-to-root tree transducers*. *Acta Cybern.* 4(2), pp. 167–177.

Grammars with two-sided contexts[*]

Mikhail Barash
Department of Mathematics and Statistics, University of Turku, Turku FI-20014, Finland
Turku Centre for Computer Science, Turku FI-20520, Finland
mikbar@utu.fi

Alexander Okhotin
Department of Mathematics and Statistics, University of Turku, Turku FI-20014, Finland
alexander.okhotin@utu.fi

In a recent paper (M. Barash, A. Okhotin, "Defining contexts in context-free grammars", LATA 2012), the authors introduced an extension of the context-free grammars equipped with an operator for referring to the left context of the substring being defined. This paper proposes a more general model, in which context specifications may be two-sided, that is, both the left and the right contexts can be specified by the corresponding operators. The paper gives the definitions and establishes the basic theory of such grammars, leading to a normal form and a parsing algorithm working in time $\mathcal{O}(n^4)$, where n is the length of the input string.

1 Introduction

The context-free grammars are a logic for representing the syntax of languages, in which the properties of longer strings are defined by concatenating shorter strings with known properties. Disjunction of syntactic conditions is represented in this logic as multiple alternative rules defining a single symbol. One can further augment this logic with conjunction and negation operations, leading to *conjunctive grammars* [13] and *Boolean grammars* [15]. These grammars are context-free in the general sense of the word, as they define the properties of each substring independently of the context, in which it occurs. Furthermore, most of the practically important features of ordinary context-free grammars, such as efficient parsing algorithms, are preserved in their conjunctive and Boolean variants [15, 18]. These grammar models have been a subject of recent theoretical studies [1, 8, 10, 12, 24].

Not long ago, the authors [3, 4] proposed an extension of the context-free grammars with special operators for expressing the form of the *left context*, in which the substring occurs. For example, a rule $A \to BC \,\&\, \lhd D$ asserts that every string representable as BC in a left context of the form described by D therefore has the property A. These grammars were motivated by Chomsky's [6, p. 142] well-known idea of a phrase-structure rule applicable only in some particular contexts. Chomsky's own attempt to implement this idea by string rewriting resulted in a model equivalent to linear-space Turing machines, in which the "nonterminal symbols", meant to represent syntactic categories, could be freely manipulated as tape symbols. In spite of the name "context-sensitive grammars", the resulting model was unsuitable for describing the syntax of languages, and thus failed to represent the idea of a rule applicable in a context.

Taking a new start with this idea, the authors [4] defined *grammars with one-sided contexts*, following the logical outlook on grammars, featured in the work of Kowalski [11, Ch. 3] and of Pereira and

[*]Supported by the Academy of Finland under grant 257857.

Warren [19], and later systematically developed by Rounds [21]. A grammar defines the truth value of statements of the form "a certain string has a certain property", and these statements are deduced from each other according to the rules of the grammar. The resulting definition maintains the underlying logic of the context-free grammars, and many crucial properties of grammars are preserved: grammars with one-sided contexts have parse trees, can be transformed to a normal form and have a cubic-time parsing algorithm [4]. However, the model allowed specifying contexts only on one side, and thus it implemented, so to say, only one half of Chomsky's idea.

This paper continues the development of formal grammars with context specifications by allowing contexts in both directions. The proposed *grammars with two-sided contexts* may contain such rules as $A \to BC \mathbin{\&} \lhd D \mathbin{\&} \rhd E$, which define any substring of the form BC preceded by a substring of the form D and followed by a substring of the form E. If the grammar contains additional rules $B \to b$, $C \to c$, $D \to d$ and $E \to e$, then the above rule for A asserts that a substring bc of a string $w = dbce$ has the property A. However, this rule will not produce the same substring bc occurring in another string $w' = dbcd$, because its right context does not satisfy the conjunct $\rhd E$. Furthermore, the grammars allow expressing the so-called *extended right context* ($\trianglerighteq\alpha$), which defines the form of the current substring concatenated with its right context, as well as the symmetrically defined *extended left context* ($\trianglelefteq\alpha$).

In Section 2, this intuitive definition is formalized by deduction of propositions of the form $A\big(u\langle w\rangle v\big)$, which states that the substring w occurring in the context between u and v has the property A, where A is a syntactic category defined by the grammar ("nonterminal symbol" in Chomsky's terminology). Then, each rule of the grammar becomes a schema for deduction rules, and a string w is generated by the grammar, if there is a proof of the proposition $S\big(\varepsilon\langle w\rangle\varepsilon\big)$. A standard proof tree of such a deduction constitutes a parse tree of the string w.

The next Section 3 presents basic examples of grammars with two-sided contexts. These examples model several types of cross-references, such as declaration of identifiers before or after their use.

The paper then proceeds with developing a normal form for these grammars, which generalizes the Chomsky normal form for ordinary context-free grammars. In the normal form, every rule is a conjunction of one or more *base conjuncts* describing the form of the current substring (either as a concatenation of the form BC or as a single symbol a), with any context specifications ($\lhd D$, $\trianglelefteq E$, $\trianglerighteq F$, $\rhd H$). The transformation to the normal form, presented in Section 4, proceeds in three steps. First, all rules generating the empty string in any contexts are eliminated. Second, all rules with an explicit empty context specification ($\lhd\varepsilon$, $\rhd\varepsilon$) are also eliminated. The final step is elimination of any rules of the form $A \to B \mathbin{\&} \ldots$, where the dependency of A on B potentially causes cycles in the definition.

Once the normal form is established, a simple parsing algorithm for grammars with two-sided contexts with the running time $\mathcal{O}(n^4)$ is presented in Section 5. While this paper has been under preparation, Rabkin [20] has developed a more efficient and more sophisticated parsing algorithm for grammars with two-sided contexts, with the running time $\mathcal{O}(n^3)$.

2 Definition

Ordinary context-free grammars allow using the concatenation operation to express the form of a string, and disjunction to define alternative forms. In conjunctive grammars, the conjunction operation may be used to assert that a substring being defined must conform to several conditions at the same time. The grammars studied in this paper further allow operators for expressing the form of the left context ($\lhd$, $\trianglelefteq$) and the right context ($\rhd$, $\trianglerighteq$) of a substring being defined.

Definition 1. *A grammar with two-sided contexts is a quadruple $G = (\Sigma, N, R, S)$, where*

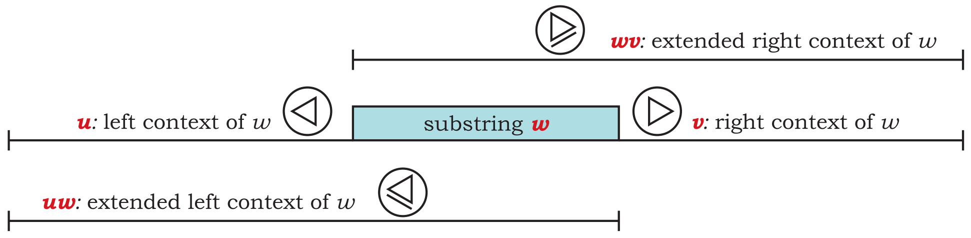

wv: extended right context of *w*
u: left context of *w*
substring *w*
v: right context of *w*
uw: extended left context of *w*

Figure 1: A substring w of a string uwv: four types of contexts.

- *Σ is the alphabet of the language being defined;*
- *N is a finite set of auxiliary symbols ("nonterminal symbols" in Chomsky's terminology), which denote the properties of strings defined in the grammar;*
- *R is a finite set of grammar rules, each of the form*

$$A \to \alpha_1 \,\&\, \ldots \,\&\, \alpha_k \,\&\, \lhd\beta_1 \,\&\, \ldots \,\&\, \lhd\beta_m \,\&\, \unlhd\gamma_1 \,\&\, \ldots \,\&\, \unlhd\gamma_n \,\&\, \\ \&\, \unrhd\kappa_1 \,\&\, \ldots \,\&\, \unrhd\kappa_{m'} \,\&\, \rhd\delta_1 \,\&\, \ldots \,\&\, \rhd\delta_{n'}, \tag{1}$$

  with $A \in N$, $k \geqslant 1$, $m,n,m',n' \geqslant 0$ and $\alpha_i, \beta_i, \gamma_i, \kappa_i, \delta_i \in (\Sigma \cup N)^*$;
- *$S \in N$ is a symbol representing well-formed sentences of the language.*

If all rules in a grammar have only left contexts (that is, if $m' = n' = 0$), then this is a grammar with one-sided contexts [4]. If no context operators are ever used ($m = n = m' = n' = 0$), this is a conjunctive grammar, and if the conjunction is also never used ($k = 1$), this is an ordinary context-free grammar.

For each rule (1), each term α_i, $\lhd\beta_i$, $\unlhd\gamma_i$, $\unrhd\kappa_i$ and $\rhd\delta_i$ is called a *conjunct*. Denote by $u\langle w\rangle v$ a substring $w \in \Sigma^*$, which is preceded by $u \in \Sigma^*$ and followed by $v \in \Sigma^*$, as illustrated in Figure 1. Intuitively, such a substring is generated by a rule (1), if

- each *base conjunct* $\alpha_i = X_1 \ldots X_\ell$ gives a representation of w as a concatenation of shorter substrings described by $X_1, \ldots, X_\ell$, as in context-free grammars;
- each conjunct $\lhd\beta_i$ similarly describes the form of the *left context u*;
- each conjunct $\unlhd\gamma_i$ describes the form of the *extended left context uw*;
- each conjunct $\unrhd\kappa_i$ describes the *extended right context wv*;
- each conjunct $\rhd\delta_i$ describes the *right context v*.

The semantics of grammars with two-sided contexts are defined by a deduction system of elementary propositions (items) of the form "a string $w \in \Sigma^*$ written in a left context $u \in \Sigma^*$ and in a right context $v \in \Sigma^*$ has the property $X \in \Sigma \cup N$", denoted by $X\big(u\langle w\rangle v\big)$. The deduction begins with axioms: any symbol $a \in \Sigma$ written in any context has the property a, denoted by $a\big(u\langle a\rangle v\big)$ for all $u, v \in \Sigma^*$. Each rule in R is then regarded as a schema for deduction rules. For example, a rule $A \to BC$ allows making deductions of the form

$$B\big(u\langle w\rangle w'v\big), C\big(uw\langle w'\rangle v\big) \vdash_G A\big(u\langle ww'\rangle v\big) \qquad \text{(for all } u, w, w', v \in \Sigma^*\text{)},$$

which is essentially a concatenation of w and w' that respects the contexts. If the rule is of the form $A \to BC \,\&\, \lhd D$, this deduction requires an extra premise:

$$B\big(u\langle w\rangle w'v\big), C\big(uw\langle w'\rangle v\big), D\big(\varepsilon\langle u\rangle ww'v\big) \vdash_G A\big(u\langle ww'\rangle v\big).$$

And if the rule is $A \to BC \,\&\, \trianglerighteq F$, the deduction proceeds as follows:

$$B\big(u\langle w\rangle w'v\big), C\big(uw\langle w'\rangle v\big), F\,\big(u\langle ww'v\rangle \varepsilon\big) \vdash_G A\big(u\langle ww'\rangle v\big).$$

The general form of deduction schemata induced by a rule in R is defined below.

Definition 2. *Let $G = (\Sigma, N, R, S)$ be a grammar with two-sided contexts. Define the following deduction system of items of the form $X\,\big(u\langle w\rangle v\big)$, with $X \in \Sigma \cup N$ and $u, w, v \in \Sigma^*$. There is a single axiom scheme $\vdash_G a\big(u\langle a\rangle v\big)$, for all $a \in \Sigma$ and $u, v \in \Sigma^*$. Each rule (1) in R defines the following scheme for deduction rules:*

$$I \vdash_G A\big(u\langle w\rangle v\big),$$

for all $u, w, v \in \Sigma^*$ and for every set of items I satisfying the below properties:

- *For every base conjunct $\alpha_i = X_1 \dots X_\ell$, with $\ell \geqslant 0$ and $X_j \in \Sigma \cup N$, there should exist a partition $w = w_1 \dots w_\ell$ with $X_j\big(uw_1 \dots w_{j-1}\langle w_j\rangle w_{j+1} \dots w_\ell v\big) \in I$ for all $j \in \{1, \dots, \ell\}$.*
- *For every conjunct $\lhd\beta_i = \lhd X_1 \dots X_\ell$ there should be such a partition $u = u_1 \dots u_\ell$, that $X_j\big(u_1 \dots u_{j-1}\langle u_j\rangle u_{j+1} \dots u_\ell wv\big) \in I$ for all $j \in \{1, \dots, \ell\}$.*
- *Every conjunct $\trianglelefteq\gamma_i = \trianglelefteq X_1 \dots X_\ell$ should have a corresponding partition $uw = x_1 \dots x_\ell$ with $X_j\big(x_1 \dots x_{j-1}\langle x_j\rangle x_{j+1} \dots x_\ell v\big) \in I$ for all $j \in \{1, \dots, \ell\}$.*
- *For every conjunct $\rhd\delta_i$ and $\trianglerighteq\kappa_i$, the condition is defined symmetrically.*

Then the language generated by a symbol $A \in N$ is defined as

$$L_G(A) = \{\, u\langle w\rangle v \mid u, w, v \in \Sigma^*, \vdash_G A\big(u\langle w\rangle v\big) \,\}.$$

The language generated by the grammar G is the set of all strings with empty left and right contexts generated by S: $L(G) = \{\, w \mid w \in \Sigma^*, \vdash_G S\big(\varepsilon\langle w\rangle \varepsilon\big) \,\}$.

The following trivial example of a grammar is given to illustrate the definitions.

Example 1. Consider the grammar with two-sided contexts that defines the singleton language $\{abca\}$:

$$\begin{array}{rcl} S & \to & aS \mid Sa \mid BC \\ A & \to & a \\ B & \to & b \,\&\, \lhd A \\ C & \to & c \,\&\, \rhd A \end{array}$$

The deduction given below proves that the string $abca$ has the property S.

$$\begin{array}{rll}
\vdash a\big(\varepsilon\langle a\rangle bca\big) & & (axiom) \\
\vdash b\big(a\langle b\rangle ca\big) & & (axiom) \\
\vdash c\big(ab\langle c\rangle a\big) & & (axiom) \\
\vdash a\big(abc\langle a\rangle \varepsilon\big) & & (axiom) \\
a\big(\varepsilon\langle a\rangle bca\big) \vdash A\big(\varepsilon\langle a\rangle bca\big) & & (A \to a) \\
b\big(a\langle b\rangle ca\big), A\big(\varepsilon\langle a\rangle bca\big) \vdash B\big(a\langle b\rangle ca\big) & & (B \to b \,\&\, \lhd A) \\
a\big(abc\langle a\rangle \varepsilon\big) \vdash A\big(abc\langle a\rangle \varepsilon\big) & & (A \to a) \\
c\big(ab\langle c\rangle a\big), A\big(abc\langle a\rangle \varepsilon\big) \vdash C\big(ab\langle c\rangle a\big) & & (C \to c \,\&\, \rhd A) \\
B\big(a\langle b\rangle ca\big), C\big(ab\langle c\rangle a\big) \vdash S\big(a\langle bc\rangle a\big) & & (S \to BC) \\
a\big(\varepsilon\langle a\rangle bca\big), S\big(a\langle bc\rangle a\big) \vdash S\big(\varepsilon\langle abc\rangle a\big) & & (S \to aS) \\
S\big(\varepsilon\langle abc\rangle a\big), a\big(abc\langle a\rangle \varepsilon\big) \vdash S\big(\varepsilon\langle abca\rangle \varepsilon\big) & & (S \to Sa)
\end{array}$$

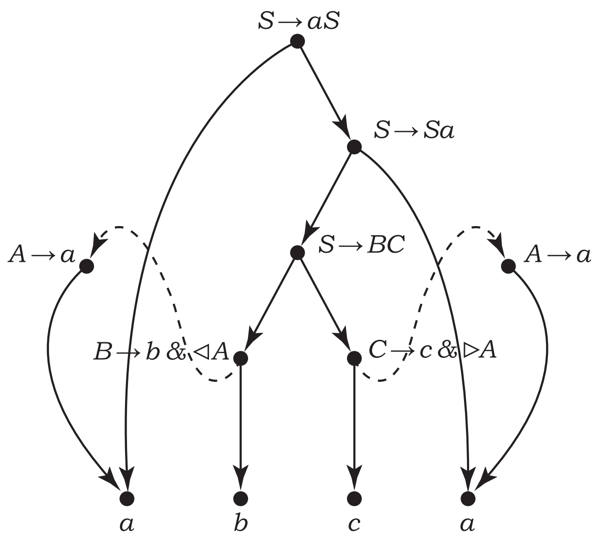


Figure 2: A parse tree of the string *abca* according to the grammar in Example 1.

Another possible definition of grammars with contexts is by directly expressing them in first-order logic over positions in a string [21]. Nonterminal symbols become *binary predicates*, with the arguments referring to positions in the string. Each predicate $A(x,y)$ is defined by a formula $\varphi_A(x,y)$ that states the condition of a substring delimilited by positions x and y having the property A. There are built-in unary predicates $a(x)$, for each $a \in \Sigma$, which assert that the symbol in position x in the string is a, and binary predicates $x < y$ and $x = y$ for comparing positions. Arguments to predicates are given as *terms*, which are either variables ($t = x$) or constants referring to the first and the last positions ($t = \underline{\text{begin}}$, $t = \underline{\text{end}}$), and which may be incremented ($t+1$) or decremented ($t-1$). Each formula is constructed from predicates using conjunction, disjunction and first-order existential quantification.

Example 2. The grammar from Example 1 is expressed by the following formulae defining predicates $S(x,y)$, $B(x,y)$, $A(x,y)$ and $C(x,y)$.

$$\begin{aligned} S(x,y) &= (a(x) \wedge S(x+1,y)) \vee (S(x,y-1) \wedge a(y)) \vee (\exists z\,(x < z < y \wedge B(x,z) \wedge C(z,y))) \\ A(x,y) &= a(x) \wedge x+1 = y \\ B(x,y) &= b(x) \wedge x+1 = y \wedge A(\underline{\text{begin}},x) \\ C(x,y) &= c(x) \wedge x+1 = y \wedge A(y,\underline{\text{end}}) \end{aligned}$$

The membership of a string w is expressed by the statement $S(\underline{\text{begin}},\underline{\text{end}})$, which may be true of false.

3 Examples

This section presents several examples of grammars with two-sided contexts generating important syntactic constructs. All examples use ordinary context-free elements, such as a grammar for $\{\, a^n b^n \mid n \geqslant 0 \,\}$, and combine these elements using the new context operators. This leads to natural specifications of languages in the style of classical formal grammars.

Consider the problem of checking declaration of identifiers before their use: this construct can be found in all kinds of languages, and it can be expressed by a conjunctive grammar [16, Ex. 3]. The variant of this problem, in which the identifiers may be declared *before or after* their use, is also fairly common: consider, for instance, the declaration of classes in C++, where an earlier defined method can refer to a class member defined later. However, no conjunctive grammar expressing this construct is known.

A grammar with one-sided contexts for declarations before or after use has recently been constructed by the authors [4]. That grammar used context specifications, along with iterated conjunction, to express what would be more naturally expressed in terms of two-sided contexts. In the model proposed in this paper, the same language can be defined in a much more natural way.

Example 3 (cf. grammar with one-sided contexts [4, Ex. 4])**.** Consider the language

$$\{\, u_1 \dots u_n \mid \text{for every } u_i, \textbf{ either } u_i \in a^*c, \textbf{ or } u_i = b^k c \text{ and there exists } j \in \{1, \dots, n\} \text{ with } u_j = a^k c \,\}.$$

Substrings of the form $a^k c$ represent declarations, while every substring of the form $b^k c$ is a reference to a declaration of the form $a^k c$.

This language is generated by the following grammar.

$$\begin{array}{rcl@{\qquad\qquad}rcl}
S & \to & AS \mid CS \mid DS \mid \varepsilon & C & \to & B \,\&\, \lhd EFc \\
A & \to & aA \mid c & D & \to & B \,\&\, \unrhd HcE \\
B & \to & bB \mid c & F & \to & aFb \mid cE \\
E & \to & AE \mid BE \mid \varepsilon & H & \to & bHa \mid cE
\end{array}$$

The idea of the grammar is that S should generate a substring $u_1 \dots u_\ell \langle u_{\ell+1} \dots u_n \rangle \varepsilon$, with $0 \leqslant \ell \leqslant n$ and $u_i \in a^*c \cup b^*c$, if and only if every reference in $u_{\ell+1} \dots u_n$ has a corresponding declaration somewhere in the whole string $u_1 \dots u_n$. The rules for S define all substrings satisfying this condition inductively on their length, until the entire string $\varepsilon \langle u_1 \dots u_n \rangle \varepsilon$ is defined. The rule $S \to \varepsilon$ defines the base case: the string $u_1 \dots u_n \langle \varepsilon \rangle \varepsilon$ has the desired property. The rule $S \to CS$ appends a reference of the form b^*c, restricted by an extended left context $\lhd EFc$, which ensures that this reference has a matching *earlier* declaration; here E represents the prefix of the string up to that earlier declaration, while F matches the symbols a in the declaration to the symbols b in the reference. The possibility of a *later* declaration is checked by another rule $S \to DS$, which adds a reference of the form b^*c with an extended right context $\unrhd HcE$, where H is used to match the bs forming this reference to the as in the later declaration.

The next example abstracts another syntactic mechanism—*function prototypes*—found in the C programming language and, under the name of *forward declarations*, in the programming language Pascal.

Example 4. Consider the language

$$\Big\{ u_1 \dots u_n \,\Big|\, \text{for every } u_i, \textbf{ either } u_i = a^k c \text{ and there exists } j > i, \text{ such that } u_j = d^k c, \tag{2a}$$

$$\textbf{or } u_i = b^k c \text{ and there exists } j < i, \text{ for which } u_j = a^k c \Big\}. \tag{2b}$$

A substring of the form $a^k c$ represents a function prototype and a substring $d^k c$ represents its body. Calls to functions are expressed as substrings $b^k c$. Condition (2a) means that every prototype must be followed by its body, and restriction (2b) requires that references are only allowed to declared prototypes.

This language can be generated by the following grammar with two-sided contexts.

$$\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl}
S & \to & US \mid VS \mid DS \mid \varepsilon & D & \to & dD \mid c & E & \to & AE \mid BE \mid DE \mid \varepsilon \\
A & \to & aA \mid c & U & \to & A \,\&\, \unrhd HcE & H & \to & aHd \mid cE \\
B & \to & bB \mid c & V & \to & B \,\&\, \lhd EFc & F & \to & aFb \mid cE
\end{array}$$

The rules $S \to US$ and $U \to A \,\&\, \trianglerighteq HcE$ append a prototype $a^k c$ and the extended right context of the form $a^k c \dots d^k c \dots$ ensures that this prototype has a matching body somewhere *later* within the string. The rules $S \to VS$ and $V \to B \,\&\, \trianglelefteq EFc$ append a reference $b^k c$, and the context specification $\dots a^k c \dots b^k c$ checks that it has a matching prototype *ealier* in the string. Function bodies $d^k c$ are added by the rule $S \to DS$. Using these rules, S generates substrings of the form $u_1 \dots u_\ell \langle u_{\ell+1} \dots u_n \rangle \varepsilon$, with $0 \leqslant \ell \leqslant n$ and $u_i \in a^* c \cup b^* c \cup d^* c$, such that every prototype $u_i = a^k c$ in $u_{\ell+1} \dots u_n$ has a corresponding body $d^k c$ in $u_{i+1} \dots u_n$ and every reference $u_i = b^k c$ in $u_{\ell+1} \dots u_n$ has a corresponding prototype $a^k c$ in $u_1 \dots u_{i-1}$.

The next example gives a grammar with contexts that defines reachability on graphs. Sudborough [22] defined a linear context-free grammar for a special encoding of the graph reachability problem on acyclic graphs, in which every arc goes from a lower-numbered vertex to a higher-numbered vertex. The grammar presented below allows any graphs and uses a direct encoding. This example illustrates the ability of grammars with contexts to define various kinds of cross-references.

Example 5. Consider encodings of directed graphs as strings of the form $b^s a^{i_1} b^{j_1} a^{i_2} b^{j_2} \dots a^{i_n} b^{j_n} a^t$, with $s, t \geqslant 1$, $n \geqslant 0$, $i_k, j_k \geqslant 1$, where each block $a^i b^j$ denotes an arc from vertex number i to vertex number j, while the prefix b^s and the suffix a^t mark s as the source vertex and t as the target. Then the following grammar defines all graphs with a path from s to t.

$$\begin{array}{rcl@{\qquad}rcl}
S & \to & FDCA \mid F & & & \\
A & \to & aA \mid c & D & \to & B \,\&\, \trianglelefteq BCE \mid B \,\&\, \trianglerighteq FDCA \mid B \,\&\, \trianglerighteq F \\
B & \to & bB \mid c & E & \to & aEb \mid DCA \\
C & \to & ABC \mid \varepsilon & F & \to & bFa \mid bCa
\end{array}$$

The grammar is centered around the nonterminal D, which generates all such substrings $b^s a^{i_1} b^{j_1} \dots a^{i_k} \langle b^{j_k} \rangle a^{i_{k+1}} b^{j_{k+1}} \dots a^{i_n} b^{j_n} a^t$ that there is a path from j_k to t in the graph. If this path is empty, then $j_k = t$. Otherwise, the first arc in the path can be listed either to the left or to the right of b^k. These three cases are handled by the three rules for D. Each of these rules generates b^{j_k} by the base conjunct B, and then uses an extended left or right context operator to match b^{j_k} to the tail of the next arc or to a^t.

The rule $D \to B \,\&\, \trianglelefteq BCE$ considers the case when the next arc in the path is located to the left of b^{j_k}. Let this arc be $a^{i_\ell} b^{j_\ell}$, for some $\ell < k$. Then the extended left context BCE covers the substring $b^s a^{i_1} b^{j_1} \dots a^{i_\ell} b^{j_\ell} \dots a^{i_k} b^{j_k}$. The concatenation BC skips the prefix $b^s a^{i_1} b^{j_1} \dots a^{i_{\ell-1}} b^{j_{\ell-1}}$, and then the nonterminal E matches a^{i_ℓ} to b^{j_k}, verifying that $i_\ell = j_k$. After this, the rule $E \to DCA$ ensures that the substring b^{j_ℓ} is generated by D, that is, that there is a path from j_ℓ to t. The concatenation CA skips the inner substring $a^{i_{\ell+1}} b^{j_{\ell+1}} \dots a^{i_k}$.

The second rule $D \to B \,\&\, \trianglerighteq FDCA$ searches for the next arc to the right of b^{j_k}. Let this be an ℓ-th arc in the list, with $\ell > k$. The extended right context $FDCA$ should generate the suffix $b^{j_k} \dots a^{i_\ell} b^{j_\ell} \dots a^{i_n} b^{j_n} a^t$. The symbol F covers the substring $b^{j_k} \dots a^{i_\ell}$, matching b^{j_k} to a^{i_ℓ}. Then, D generates the substring b^{j_ℓ}, checking that there is a path from j_ℓ to t. The concatenation CA skips the rest of the suffix.

Finally, if the path is of length zero, that is, $j_k = t$, then the rule $D \to B \,\&\, \trianglerighteq F$ uses F to match b^{j_k} to the suffix a^t in the end of the string.

Once the symbol D checks the path from any vertex to the vertex t, for the initial symbol S, it is sufficient to match b^s in the beginning of the string to any arc $a^{j_k} b^{j_k}$, with $j_k = s$. This is done by the rule $S \to FDCA$, which operates in the same way as the second rule for D. The case of s and t being the same node is handled by the rule $S \to F$.

All the above examples use identifiers given in unary, which are matched by rules of the same kind as the rules defining the language $\{ a^n b^n \mid n \geqslant 0 \}$. These examples can be extended to use identifiers over an arbitrary alphabet Σ, owing to the fact that there is a conjunctive grammar generating the language $\{ w\#w \mid w \in \Sigma^* \}$, for some separator $\# \notin \Sigma$ [13, 16].

4 Normal form

An ordinary context-free grammar can be transformed to the Chomsky normal form, with the rules restricted to $A \to BC$ and $A \to a$, with $B, C \in N$ and $a \in \Sigma$. This form has the following generalization to grammars with contexts.

Definition 3. *A grammar with two-sided contexts $G = (\Sigma, N, R, S)$ is said to be in the binary normal form, if each rule in R is of one of the forms*

$$A \to B_1C_1 \,\&\, \dots \,\&\, B_kC_k \,\&\, \lhd D_1 \,\&\, \dots \,\&\, \lhd D_m \,\&\, \trianglelefteq E_1 \,\&\, \dots \,\&\, \trianglelefteq E_n \,\&\, \trianglerighteq F_1 \,\&\, \dots \,\&\, \trianglerighteq F_{n'} \,\&\, \rhd H_1 \,\&\, \dots \,\&\, \rhd H_{m'},$$
$$A \to a \,\&\, \lhd D_1 \,\&\, \dots \,\&\, \lhd D_m \,\&\, \trianglelefteq E_1 \,\&\, \dots \,\&\, \trianglelefteq E_n \,\&\, \trianglerighteq F_1 \,\&\, \dots \,\&\, \trianglerighteq F_{n'} \,\&\, \rhd H_1 \,\&\, \dots \,\&\, \rhd H_{m'},$$

where $k \geqslant 1$, $m, n, n', m' \geqslant 0$, $B_i, C_i, D_i, E_i, F_i, H_i \in N$, $a \in \Sigma$.

The transformation to the normal form consists of three stages: first, removing all *empty conjuncts* ε; secondly, eliminating *empty contexts* ($\lhd\varepsilon$, $\rhd\varepsilon$); finally, getting rid of *unit conjuncts* of the form B, with $B \in N$.

The first step is the removal of all rules of the form $A \to \varepsilon \,\&\, \dots$, so that no symbols generate ε, while all non-empty strings are generated as before. As generation of longer strings may depend on the generation of ε, already for ordinary context-free grammars, such a transformation requires adding extra rules that simulate the same dependence without actually generating any empty strings.

Example 6. Consider the following context-free grammar, which defines the language $\{abc, ab, ac, a, bcd, bd, cd, d\}$.

$$\begin{array}{rcl} S & \to & aA \mid Ad \\ A & \to & BC \\ B & \to & \varepsilon \mid b \\ C & \to & \varepsilon \mid c \end{array}$$

Since B generates the empty string, the rule $A \to BC$ can be used to generate just C; therefore, once the rule $B \to \varepsilon$ is removed, one should add a new rule $A \to C$, in which B is omitted. Similarly one can remove the rule $C \to \varepsilon$ and add a "compensatory" rule $A \to B$. Since both B and C generate ε, so does A by the rule $A \to BC$. Hence, extra rules $S \to a$ and $S \to d$, where A is omitted, have to be added.

An algorithm for carrying out such a transformation first calculates the set of nonterminals that generate the empty string, known as $\text{NULLABLE}(G) \subseteq N$, and then uses it to reconstruct the rules of the grammar.

This set is calculated as a least upper bound of an ascending sequence of sets $\text{NULLABLE}_i(G)$. The set $\text{NULLABLE}_1(G) = \{A \in N \mid A \to \varepsilon \in R\}$ contains all nonterminals which directly define the empty string. Every next set $\text{NULLABLE}_{i+1}(G) = \{A \in N \mid A \to \alpha \in R,\ \alpha \in \text{NULLABLE}_i^*(G)\}$ contains nonterminals that generate ε by the rules referring to other nullable nonterminals. This knowledge is given by the Kleene star of $\text{NULLABLE}_i(G)$.

For the grammar in Example 6, the calculation of the set $\text{NULLABLE}(G)$ proceeds as follows:

$$\begin{array}{rcl} \text{NULLABLE}_0(G) & = & \varnothing, \\ \text{NULLABLE}_1(G) & = & \{B, C\}, \\ \text{NULLABLE}_2(G) & = & \{B, C, A\}, \end{array}$$

and $\text{NULLABLE}(G) = \text{NULLABLE}_2(G)$.

The same idea works for conjunctive grammars as well [13]. For grammars with contexts [4], the generation of the empty string additionally depends on the left contexts, in which the string occurs. This requires an elaborated version of the set $\text{NULLABLE}(G)$, formed of nonterminals along with the information about the left contexts in which they may define ε.

In order to eliminate null conjuncts in case of grammars with two-sided contexts, one has to consider yet another variant of the set $\text{NULLABLE}(G)$, which respects both left and right contexts.

Example 7. Consider the following grammar with two-sided contexts, obtained by adding context restrictions to the grammar in Example 6; this grammar defines the language $L = \{abc, ac, bcd, bd\}$.

$$\begin{array}{rcl} S & \to & aA \mid Ad \\ A & \to & BC \\ B & \to & \varepsilon \,\&\, \lhd D \mid b \\ C & \to & \varepsilon \,\&\, \rhd E \mid c \\ D & \to & a \\ E & \to & d \end{array}$$

In this grammar, the nonterminal B generates the empty string only in a left context of the form defined by D, while C defines the empty string only in a right context of the form E. In those contexts where *both* B and C generate ε, so can A, by the rule $A \to BC$.

The information about the left and right contexts, in which a nonterminal generates the empty string, is to be stored in the set $\text{NULLABLE}(G)$, which is defined as a subset of $2^N \times N \times 2^N$. An element (U, A, V) of this set represents an intuitive idea that A defines ε in a left context of the form described by each nonterminal in U, and in a right context of the form given by nonterminals in V.

For the grammar in Example 7, such a set $\text{NULLABLE}(G)$ is constructed as follows.

$$\begin{array}{rcl} \text{NULLABLE}_0(G) & = & \varnothing \\ \text{NULLABLE}_1(G) & = & \big\{(\{D\}, B, \varnothing), (\varnothing, C, \{E\})\big\} \\ \text{NULLABLE}_2(G) & = & \big\{(\{D\}, B, \varnothing), (\varnothing, C, \{E\}), (\{D\}, A, \{E\})\big\} \end{array}$$

Then $\text{NULLABLE}(G) = \text{NULLABLE}_2(G)$. The elements $(\{D\}, B, \varnothing)$ and $(\varnothing, C, \{E\})$ are obtained directly from the rules of the grammar, and the element $(\{D\}, A, \{E\})$ represents the "concatenation" BC in the rule for A. Note the similarity of this construction to the one for the ordinary grammar in Example 6: the construction given here is different only in recording information about the contexts.

The above "concatenation" of triples $(\{D\}, B, \varnothing)$ and $(\varnothing, C, \{E\})$ should be defined to accumulate both left and right contexts. This can be regarded as a generalization of the Kleene star to sets of triples, denoted by $\text{NULLABLE}^\star(G)$. Formally, $\text{NULLABLE}^\star(G)$ is the set of all triples $(U_1 \cup \ldots \cup U_\ell,\, A_1 \ldots A_\ell,\, V_1 \cup \ldots \cup V_\ell)$ with $\ell \geqslant 0$ and $(U_i, A_i, V_i) \in \text{NULLABLE}(G)$. The symbols A_i are concatenated, while their left and right contexts are accumulated. In the special case when $\ell = 0$, the concatenation of zero symbols is the empty string, and thus $\varnothing^\star = \big\{(\varnothing, \varepsilon, \varnothing)\big\}$.

Before giving a formal definition of the set $\text{NULLABLE}(G)$, assume, for the sake of simplicity, that context operators are only applied to single nonterminal symbols, that is, every rule is of the form

$$A \to \alpha_1 \,\&\, \ldots \,\&\, \alpha_k \,\&\, \lhd D_1 \,\&\, \ldots \,\&\, \lhd D_m \,\&\, \trianglelefteq E_1 \,\&\, \ldots \,\&\, \trianglelefteq E_n \,\&\, \trianglerighteq F_1 \,\&\, \ldots \,\&\, \trianglerighteq F_{m'} \,\&\, \rhd H_1 \,\&\, \ldots \,\&\, \rhd H_{n'}, \quad (3)$$

with $A \in N$, $k \geqslant 1$, $m, n, m', n' \geqslant 0$, $\alpha_i \in (\Sigma \cup N)^*$ and $D_i, E_i, F_i, H_i \in N$. As will be shown in Lemma 3, there is no loss of generality in this assumption.

Definition 4. *Let $G = (\Sigma, N, R, S)$ be a grammar with two-sided contexts with all rules of the form (3). Construct the sequence of sets $\text{NULLABLE}_i(G) \subseteq 2^N \times N \times 2^N$, for $i \geqslant 0$, as follows.*

Let $\text{NULLABLE}_0(G) = \varnothing$. Every next set $\text{NULLABLE}_{i+1}(G)$ contains the following triples: for every rule (3) and for every k triples (U_1, α_1, V_1), $\ldots$, (U_k, α_k, V_k) in $\text{NULLABLE}_i^{\star}(G)$, the triple $\big(\{D_1, \ldots, D_m, E_1, \ldots, E_n\} \cup \{U_1, \ldots, U_k\},\ A,\ \{F_1, \ldots, F_{m'}, H_1, \ldots, H_{n'}\} \cup \{V_1, \ldots, V_k\}\big)$ is in $\text{NULLABLE}_{i+1}(G)$.

Finally, let $\text{NULLABLE}(G) = \bigcup_{i \geqslant 0} \text{NULLABLE}_i(G)$.

The next lemma explains how exactly the set $\text{NULLABLE}(G)$ represents the generation of the empty string by different nonterminals in different contexts.

Lemma 1. *Let $G = (\Sigma, N, R, S)$ be a grammar with contexts, let $A \in N$ and $u, v \in \Sigma^*$. Then, $u\langle\varepsilon\rangle v \in L_G(A)$ if and only if there is a triple $(\{J_1, \ldots, J_s\}, A, \{K_1, \ldots, K_t\})$ in $\text{NULLABLE}(G)$, such that $\varepsilon\langle u\rangle v \in L_G(J_i)$ for all i and $u\langle v\rangle\varepsilon \in L_G(K_j)$ for all j.*

The plan is to reconstruct the grammar, so that for every triple $(\{J_1, \ldots, J_s\}, A, \{K_1, \ldots, K_t\})$ in $\text{NULLABLE}(G)$, and for every occurrence of A in the right-hand side of any rule, the new grammar contains a companion rule, in which A is omitted and context operators for J_i and K_i are introduced.

The following case requires special handling in the new grammar. Assume that A generates ε in the empty left context (that is, $u = \varepsilon$ in Lemma 1). This is reflected by a triple $(\{J_1, \ldots, J_s\}, A, \{K_1, \ldots, K_t\})$ in $\text{NULLABLE}(G)$, in which all symbols J_i also generate ε in the left context ε. The latter generation may in turn involve some further right context operators. In the new grammar, the left context will be explicitly set to be empty ($\lhd\varepsilon$), whereas all those right contexts should be assembled together with the set $\{K_1, \ldots, K_t\}$, and used in the new rules, where A is omitted. This calculation of right contexts is done in the following special variant of the set NULLABLE.

Definition 5. *Let $G = (\Sigma, N, R, S)$ be a grammar. Define sets $\lhd\varepsilon\text{-NULLABLE}_i(G) \subseteq N \times 2^N$, with $i \geqslant 0$:*

$$\lhd\varepsilon\text{-NULLABLE}_0(G) = \{\, (A, V) \mid (\varnothing, A, V) \in \text{NULLABLE}(G) \,\},$$

$$\lhd\varepsilon\text{-NULLABLE}_{i+1}(G) = \Big\{ (A, V \cup V_1 \cup \ldots \cup V_s) \,\Big|\, (\{J_1, \ldots, J_s\}, A, V) \in \text{NULLABLE}(G), \\ \exists V_1, \ldots, V_s \subseteq N : (J_i, V_i) \in \lhd\varepsilon\text{-NULLABLE}_i(G) \Big\}.$$

Let $\lhd\varepsilon\text{-NULLABLE}(G) = \bigcup_{i \geqslant 0} \lhd\varepsilon\text{-NULLABLE}_i(G)$.

Lemma 2. *Let $G = (\Sigma, N, R, S)$ be a grammar, let $A \in N$ and $v \in \Sigma^*$. Then $\varepsilon\langle\varepsilon\rangle v \in L_G(A)$ if and only if there is a pair $(A, \{K_1, \ldots, K_t\})$ in $\lhd\varepsilon\text{-NULLABLE}(G)$, such that $\varepsilon\langle v\rangle\varepsilon \in L_G(K_i)$ for all i.*

There is a symmetrically defined set $\rhd\varepsilon\text{-NULLABLE}(G) \subseteq 2^N \times N$, which characterizes the generation of ε in an empty right context.

With the generation of the empty string represented in these three sets, a grammar with two-sided contexts is transformed to the normal form as follows. First, it is convenient to simplify the rules of the grammar, so that every concatenation is of the form BC, with $B, C \in N$, and the context operators are only applied to individual nonterminals. For this, base conjuncts α with $|\alpha| > 2$ and context operators $\lhd\alpha$, $\trianglelefteq\alpha$, $\trianglerighteq\alpha$ and $\rhd\alpha$ with $|\alpha| > 1$ are shortened by introducing new nonterminals.

Lemma 3. *For every grammar $G_0 = (\Sigma, N_0, R_0, S_0)$, there exists and can be effectively constructed another grammar $G = (\Sigma, N, R, S)$ generating the same language, with all rules of the form:*

$$A \to a \tag{4a}$$

$$A \to BC \tag{4b}$$

$$A \to B_1 \,\&\, \ldots \,\&\, B_k \,\&\, \lhd D_1 \,\&\, \ldots \,\&\, \lhd D_m \,\&\, \trianglelefteq E_1 \,\&\, \ldots \,\&\, \trianglelefteq E_n \,\&\, \trianglerighteq F_1 \,\&\, \ldots \,\&\, \trianglerighteq F_{m'} \,\&\, \rhd H_1 \,\&\, \ldots \,\&\, \rhd H_{n'} \tag{4c}$$

$$A \to \varepsilon, \tag{4d}$$

with $a \in \Sigma$ and $A, B, C, D_i, E_i, F_i, H_i \in N$.

Construction 1. Let $G = (\Sigma, N, R, S)$ be a grammar with two-sided contexts, with all rules of the form (4). Consider the sets $\textsc{Nullable}(G)$, $\lhd\varepsilon\text{-}\textsc{Nullable}(G)$ and $\rhd\varepsilon\text{-}\textsc{Nullable}(G)$, and construct another grammar with two-sided contexts $G' = (\Sigma, N, R', S)$, with the following rules.

1. All rules of the form (4a) in R are added to R'.
2. Every rule of the form (4b) in R is added to R', along with the following extra rules, where a nullable nonterminal is omitted and the fact that it generates ε is expressed by context operators.

$$\begin{aligned}
&A \to B \,\&\, \trianglelefteq J_1 \,\&\, \dots \,\&\, \trianglelefteq J_s \,\&\, \rhd K_1 \,\&\, \dots \,\&\, \rhd K_t, \quad \text{for } (\{J_1, \dots, J_s\}, C, \{K_1, \dots, K_t\}) \in \textsc{Nullable}(G)\\
&A \to B \,\&\, \trianglelefteq J_1 \,\&\, \dots \,\&\, \trianglelefteq J_s \,\&\, \rhd\varepsilon, \quad \text{for } (\{J_1, \dots, J_s\}, C) \in \rhd\varepsilon\text{-}\textsc{Nullable}(G) \text{ with } s \geqslant 1\\
&A \to C \,\&\, \lhd J_1 \,\&\, \dots \,\&\, \lhd J_s \,\&\, \trianglerighteq K_1 \,\&\, \dots \,\&\, \trianglerighteq K_t, \quad \text{for } (\{J_1, \dots, J_s\}, B, \{K_1, \dots, K_t\}) \in \textsc{Nullable}(G)\\
&A \to C \,\&\, \trianglerighteq K_1 \,\&\, \dots \,\&\, \trianglerighteq K_t \,\&\, \lhd\varepsilon, \quad \text{for } (B, \{K_1, \dots, K_t\}) \in \lhd\varepsilon\text{-}\textsc{Nullable}(G) \text{ with } t \geqslant 1
\end{aligned}$$

   In the first case, C defines ε in left contexts J_i and right contexts K_i, and this restriction is implemented by context operators in the new rule. Since the left context of C includes B, extended context operators ($\trianglelefteq J_i$) are used on the left, whereas the right context operators are proper ($\rhd K_i$).

   The second case considers the possibility of a nullable nonterminal C, which defines ε in an empty right context. This condition is simulated by the conjunct $\rhd\varepsilon$ and extended left contexts $\trianglelefteq J_i$.

   The two last rules handle symmetrical cases, when the nonterminal B defines the empty string.
3. Every rule of the form (4c) is preserved in R'. In the original grammar, this rule (4c) may generate strings in empty contexts, as long as symbols in the context operators ($\lhd D_i$, $\rhd H_i$) are nullable.

   For any collection of pairs $(D_1, V_1), \dots, (D_m, V_m) \in \lhd\varepsilon\text{-}\textsc{Nullable}(G)$, with $m \geqslant 1$, add the rule

$$A \to B_1 \,\&\, \dots \,\&\, B_k \,\&\, E_1 \,\&\, \dots \,\&\, E_n \,\&\, \trianglerighteq K_1 \,\&\, \dots \,\&\, \trianglerighteq K_t \,\&\, \trianglerighteq F_1 \,\&\, \dots \,\&\, \trianglerighteq F_{m'} \,\&\, \rhd H_1 \,\&\, \dots \,\&\, \rhd H_{n'} \,\&\, \lhd\varepsilon,$$

   where $\{K_1, \dots, K_t\} = \bigcup_{i=1}^{m} V_i$. Nonterminals $D_1, \dots, D_m$ define ε in the right contexts given in the set $\lhd\varepsilon\text{-}\textsc{Nullable}(G)$. This is represented by conjuncts $\lhd\varepsilon$ and $\trianglerighteq K_i$. Extended left contexts $\trianglelefteq E_i$ are replaced with base conjuncts E_i, because in the empty left context they have the same effect.

   Symmetrically, if $(U_1, H_1), \dots, (U_{n'}, H_{n'}) \in \rhd\varepsilon\text{-}\textsc{Nullable}(G)$, with $n' \geqslant 1$, then there is a rule

$$A \to B_1 \,\&\, \dots \,\&\, B_k \,\&\, F_1 \,\&\, \dots \,\&\, F_{m'} \,\&\, \lhd D_1 \,\&\, \dots \,\&\, \lhd D_m \,\&\, \trianglelefteq E_1 \,\&\, \dots \,\&\, \trianglelefteq E_n \,\&\, \trianglelefteq K_1 \,\&\, \dots \,\&\, \trianglelefteq K_t \,\&\, \rhd\varepsilon,$$

   where $\{K_1, \dots, K_t\} = \bigcup_{i=1}^{n'} U_i$.

   Finally, if with m, $n' \geqslant 1$ and $(D_1, V_1), \dots, (D_m, V_m) \in \lhd\varepsilon\text{-}\textsc{Nullable}(G)$, $(U_1, H_1), \dots, (U_{n'}, H_{n'}) \in \rhd\varepsilon\text{-}\textsc{Nullable}(G)$, then the set R' contains a rule

$$A \to B_1 \,\&\, \dots \,\&\, B_k \,\&\, E_1 \,\&\, \dots \,\&\, E_n \,\&\, F_1 \,\&\, \dots \,\&\, F_{m'} \,\&\, K_1 \,\&\, \dots \,\&\, K_t \,\&\, \lhd\varepsilon \,\&\, \rhd\varepsilon,$$

   where $\{K_1, \dots, K_t\} = \bigcup_{i=1}^{m} V_i \cup \bigcup_{j=1}^{n'} U_j$. In this case, both left and right contexts of a string are empty. All the symbols D_i and H_i define ε in the contexts specified in $\lhd\varepsilon\text{-}\textsc{Nullable}(G)$ and $\rhd\varepsilon\text{-}\textsc{Nullable}(G)$. These contexts apply to the entire string and are explicitly stated as $K_1 \,\&\, \dots \,\&\, K_t$ in the new rule. The null contexts $\lhd\varepsilon$, $\rhd\varepsilon$ limit the applicability of this rule to the whole string. Again, as in the two previous cases, the base conjuncts are used instead of extended context operators.

Lemma 4. *Let $G = (\Sigma, N, R, S)$ be a grammar with two-sided contexts. Then the grammar $G' = (\Sigma, N', R', S)$ obtained by Construction 1 generates the language $L(G') = L(G) \setminus \{\varepsilon\}$.*

The above construction eliminates the empty string in all base conjuncts, but the resulting grammar may still contain null context specifications ($\lhd\varepsilon$ and $\rhd\varepsilon$), which state that the current substring is a prefix or a suffix of the whole string. These operators are eliminated by the following simple transformation. First, define a new nonterminal symbol U that generates all non-empty strings in the empty left context. This is done by the following three rules:

$$\begin{aligned} &U \to Ua && \text{(for all } a \in \Sigma) \\ &U \to a \,\&\, \trianglelefteq X && \text{(for all } a \in \Sigma) \\ &X \to a && \text{(for all } a \in \Sigma) \end{aligned}$$

Another symbol V generates all non-empty strings in the empty right context; it is defined by symmetric rules. Then it remains to replace left and right null context operators ($\lhd\varepsilon$, $\rhd\varepsilon$) with U and V, respectively.

The third stage of the transformation to the normal form is removing the *unit conjuncts* in rules of the form $A \to B \,\&\, \ldots$ Already for conjunctive grammars [13], the only known transformation involves substituting all rules for B into all rules for A; in the worst case, this results in an exponential blowup. The same construction applies verbatim to grammars with contexts.

This three-stage transformation proves the following theorem.

Theorem 1. *For each grammar with two-sided contexts $G = (\Sigma, N, R, S)$ there exists and can be effectively constructed a grammar with two-sided contexts $G' = (\Sigma, N', R', S)$ in the binary normal form, such that $L(G) = L(G') \setminus \{\varepsilon\}$.*

5 Parsing algorithm

Let $G = (\Sigma, N, R, S)$ be a grammar with two-sided contexts in the binary normal form, and let $w = a_1 \ldots a_n \in \Sigma^+$, with $n \geqslant 1$ and $a_i \in \Sigma$, be an input string to be parsed. For every substring of w delimited by two positions i, j, with $0 \leqslant i < j \leqslant n$, consider the set of nonterminal symbols generating this substring.

$$T_{i,j} = \big\{ A \;\big|\; A \in N,\ a_1 \ldots a_i \langle a_{i+1} \ldots a_j \rangle a_{j+1} \ldots a_n \in L_G(A) \big\}$$

In particular, the whole string w is in $L(G)$ if and only if $S \in T_{0,n}$.

In ordinary context-free grammars, a substring $a_{i+1} \ldots a_j$ is generated by A if there is a rule $A \to BC$ and a partition of the substring into $a_{i+1} \ldots a_k$ generated by B and $a_{k+1} \ldots a_j$ generated by C, as illustrated in Figure 3(left). Accordingly, each set $T_{i,j}$ depends only on the sets $T_{i',j'}$ with $j' - i' < j - i$, and hence all these sets may be constructed inductively, beginning with shorter substrings and eventually reaching the set $T_{0,n}$: this is the Cocke–Kasami–Younger parsing algorithm. For conjunctive grammars, all dependencies are the same, and generally the same parsing algorithm applies [13]. In grammars with only left contexts, each set $T_{i,j}$ additionally depends on the sets $T_{0,i}$ and $T_{0,j}$ via the conjuncts of the form $\lhd D$ and $\trianglelefteq E$, respectively, which still allows constructing these sets progressively for $j = 1, \ldots, n$ [4].

The more complicated structure of logical dependencies in grammars with two-sided contexts is shown in Figure 3(right). The following example demonstrates how these dependencies may form circles.

Example 8. Consider the grammar with the rules

$$\begin{aligned} S &\to AB \\ A &\to a \,\&\, \rhd B \\ B &\to b \,\&\, \lhd C \\ C &\to a \end{aligned}$$

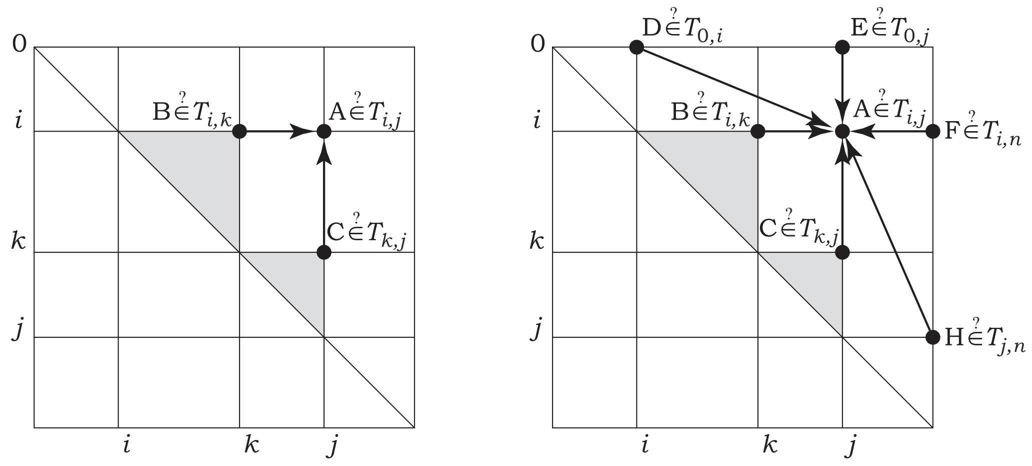

0
i
k
j
$B \overset{?}{\in} T_{i,k}$
$A \overset{?}{\in} T_{i,j}$
$C \overset{?}{\in} T_{k,j}$
i k j
$D \overset{?}{\in} T_{0,i}$
$E \overset{?}{\in} T_{0,j}$
$F \overset{?}{\in} T_{i,n}$
$H \overset{?}{\in} T_{j,n}$

Figure 3: How the membership of A in $T_{i,j}$ depends on other data, for rules (a) $A \to BC$ and (b) $A \to BC \,\&\, \lhd D \,\&\, \trianglelefteq E \,\&\, \trianglerighteq F \,\&\, \rhd H$.

and the input string $w = ab$. It is immediately seen that $C \in T_{0,1}$. From this, one can infer that $B \in T_{1,2}$, and that knowledge can in turn be used to show that $A \in T_{0,1}$. These data imply that $S \in T_{0,2}$. Thus, none of the sets $T_{0,1}$ and $T_{1,2}$ can be fully constructed before approaching the other.

The proposed algorithm for constructing the sets $T_{i,j}$ works as follows. At the first pass, it makes all deductions $\vdash_G A\big(a_1 \dots a_i \langle a_{i+1} \dots a_j \rangle a_{j+1} \dots a_n\big)$ that do not involve any contexts, and accordingly puts A to the corresponding $T_{i,j}$. This pass progressively considers longer and longer substrings, as done by the Cocke–Kasami–Younger algorithm for ordinary grammars. During this first pass, some symbols may be added to any sets $T_{0,j}$ and $T_{i,n}$, and thus it becomes known that some contexts are true. This triggers another pass over all entries $T_{i,j}$, from shorter substrings to longer ones, this time using the known true contexts in the deductions. This pass may result in adding more elements to $T_{0,j}$ and $T_{i,n}$, which will require yet another pass, and so on. Since a new pass is needed only if a new element is added to any of $2n-1$ subsets of N, the total number of passes is at most $(2n-1) \cdot |N| + 1$.

These calculations are implemented in Algorithm 1, which basically deduces all true statements about all substrings of the input string. For succinctness, the algorithm uses the following notation for multiple context operators. For a set $\mathscr{X} = \{X_1, \dots, X_\ell\}$, with $X_i \in N$, and for an operator $Q \in \{\lhd, \trianglelefteq, \trianglerighteq, \rhd\}$, denote $Q\mathscr{X} := QX_1 \,\&\, \dots \,\&\, QX_\ell$.

Theorem 2. *For every grammar with two-sided contexts G in the binary normal form, Algorithm 1, given an input string $w = a_1 \dots a_n$, constructs the sets $T_{i,j}$ and determines the membership of w in $L(G)$, and does so in time $\mathscr{O}(|G|^2 \cdot n^4)$, using space $\mathscr{O}(|G| \cdot n^2)$.*

While this paper was under preparation, Rabkin [20] developed a more efficient and more sophisticated parsing algorithm for grammars with two-sided contexts, with the running time $\mathscr{O}(|G| \cdot n^3)$, using space $\mathscr{O}(|G| \cdot n^2)$. Like Algorithm 1, Rabkin's algorithm works by proving all true statements about the substrings of the given string, but does so using the superior method of Dowling and Gallier [7]. Nevertheless, Algorithm 1 retains some value as the elementary parsing method for grammars with two-sided contexts—just like the Cocke–Kasami–Younger algorithm for ordinary grammars remains useful, in spite of the asymptotically superior Valiant's algorithm [23].

Algorithm 1. Let $G = (\Sigma, N, R, S)$ be a grammar with contexts in the binary normal form. Let $w = a_1 \dots a_n \in \Sigma^+$ (with $n \geqslant 1$ and $a_i \in \Sigma$) be the input string. Let $T_{i,j}$ with $0 \leqslant i < j \leqslant n$ be variables, each representing a subset of N, and let $T_{i,j} = \varnothing$ be their initial values.

```
1:  while any of $T_{0,j}$ ($1 \leqslant j \leqslant n$) or $T_{i,n}$ ($1 \leqslant i < n$) change do
2:      for $j = 1, \dots, n$ do
3:          for all $A \to a \,\&\, \lhd\mathscr{D} \,\&\, \trianglelefteq\mathscr{E} \,\&\, \trianglerighteq\mathscr{F} \,\&\, \rhd\mathscr{H} \in R$ do
4:              if $a_j = a \wedge \mathscr{D} \subseteq T_{0,j-1} \wedge \mathscr{E} \subseteq T_{0,j} \wedge \mathscr{F} \subseteq T_{j,n} \wedge \mathscr{H} \subseteq T_{i,n}$ then
5:                  $T_{j-1,j} = T_{j-1,j} \cup \{A\}$
6:          for $i = j-2$ to $0$ do
7:              let $P = \varnothing$ ($P \subseteq N \times N$)
8:              for $k = i+1$ to $j-1$ do
9:                  $P = P \cup (T_{i,k} \times T_{k,j})$
10:             for all $A \to B_1C_1 \,\&\, \dots \,\&\, B_mC_m \,\&\, \lhd\mathscr{D} \,\&\, \trianglelefteq\mathscr{E} \,\&\, \trianglerighteq\mathscr{F} \,\&\, \rhd\mathscr{H} \in R$ do
11:                 if $(B_1, C_1), \dots, (B_m, C_m) \in P \wedge \mathscr{D} \subseteq T_{0,i} \wedge \mathscr{E} \subseteq T_{0,j} \wedge \mathscr{F} \subseteq T_{j,n} \wedge \mathscr{H} \subseteq T_{i,n}$ then
12:                     $T_{i,j} = T_{i,j} \cup \{A\}$
13: accept if and only if $S \in T_{0,n}$
```

6 Conclusion

This paper has developed a formal representation for the idea of phrase-structure rules applicable in a context, featured in the early work of Chomsky [6]. This idea did not receive adequate treatment at the time, due to the unsuitable string-rewriting approach. The logical approach, adapted from Rounds [21] and his predecessors, brings it to life.

There are many theoretical questions to research about the new model: for instance, one can study the limitations of their expressive power, their closure properties, efficient parsing algorithms and subfamilies that admit more efficient parsing. Another possibility for further studies is investigating Boolean and stochastic variants of grammars with contexts, following the recent related work [8, 12, 24].

On a broader scope, there must have been other good ideas in the theory of formal grammars that were inadequately formalized before. They may be worth being re-investigated using the logical approach.

References

[1] T. Aizikowitz, M. Kaminski, “LR(0) conjunctive grammars and deterministic synchronized alternating pushdown automata”, *Computer Science in Russia* (CSR 2011, St. Petersburg, Russia, 14–18 June 2011), LNCS 6651, 345–358, DOI: `10.1007/978-3-642-20712-9_27`.

[2] M. Barash, “Programming language specification by a grammar with contexts”, In: S. Bensch, F. Drewes, R. Freund, F. Otto (Eds.), *Fifth Workshop on Non-Classical Models of Automata and Applications* (NCMA 2013, Umeå, Sweden, 13–14 August, 2013), books@ocg.at 294, Österreichische Computer Gesellschaft (2013), 51–67, `http://users.utu.fi/mikbar/kieli`.

[3] M. Barash, A. Okhotin, “Defining contexts in context-free grammars”, *Language and Automata Theory and Applications* (LATA 2012, A Coruña, Spain, 5–9 March 2012), LNCS 7183, 106–118, DOI: `10.1007/978-3-642-28332-1_10`.

[4] M. Barash, A. Okhotin, “An extension of context-free grammars with one-sided context specifications”, *Information and Computation*, in press, DOI: `10.1016/j.ic.2014.03.003`.

[5] M. Barash, A. Okhotin, “Linear grammars with one-sided contexts and their automaton representation”, *LATIN 2014: Theoretical Informatics* (Montevideo, Uruguay, 31 March–4 April 2014), LNCS 8392, 190–201, DOI: `10.1007/978-3-642-54423-1_17`.

[6] N. Chomsky, “On certain formal properties of grammars”, *Information and Control*, 2:2 (1959), 137–167, DOI: `10.1016/S0019-9958(59)90362-6`.

[7] W. F. Dowling, J. H. Gallier, “Linear-time algorithms for testing the satisfiability of propositional Horn formulae”, *Journal of Logic Programming*, 1:3 (1984), 267–284, DOI: `10.1016/0743-1066(84)90014-1`.

[8] Z. Ésik, W. Kuich, “Boolean fuzzy sets”, *International Journal of Foundations of Computer Science*, 18:6 (2007), 1197–1207, DOI: `10.1142/S0129054107005248`.

[9] S. Ginsburg, H. G. Rice, “Two families of languages related to ALGOL”, *Journal of the ACM*, 9 (1962), 350–371, DOI: `10.1145/321127.321132`.

[10] A. Jeż, “Conjunctive grammars can generate non-regular unary languages”, *International Journal of Foundations of Computer Science*, 19:3 (2008), 597–615, DOI: `10.1142/S012905410800584X`.

[11] R. Kowalski, *Logic for Problem Solving*, North-Holland, Amsterdam, 1979.

[12] V. Kountouriotis, Ch. Nomikos, P. Rondogiannis, “Well-founded semantics for Boolean grammars”, *Information and Computation*, 207:9 (2009), 945–967, DOI: `10.1016/j.ic.2009.05.002`.

[13] A. Okhotin, “Conjunctive grammars”, *Journal of Automata, Languages and Combinatorics*, 6:4 (2001), 519–535.

[14] A. Okhotin, “Conjunctive grammars and systems of language equations”, *Programming and Computer Software*, 28:5 (2002), 243–249, DOI: `10.1023/A:1020213411126`.

[15] A. Okhotin, “Boolean grammars”, *Information and Computation*, 194:1 (2004), 19–48, DOI: `10.1016/j.ic.2004.03.006`.

[16] A. Okhotin, “Conjunctive and Boolean grammars: the true general case of the context-free grammars”, *Computer Science Review*, 9 (2013), 27–59, DOI: `10.1016/j.cosrev.2013.06.001`.

[17] A. Okhotin, “Improved normal form for grammars with one-sided contexts”, *Descriptional Complexity of Formal Systems* (DCFS 2013, London, Ontario, Canada, 22-25 July 2013), LNCS 8031, 205–216, DOI: `10.1007/978-3-642-39310-5_20`.

[18] A. Okhotin, “Parsing by matrix multiplication generalized to Boolean grammars”, *Theoretical Computer Science*, 516 (2014), 101–120, DOI: `10.1016/j.tcs.2013.09.011`.

[19] F. C. N. Pereira, D. H. D. Warren, “Parsing as deduction”, *21st Annual Meeting of the Association for Computational Linguistics* (ACL 1983, Cambridge, Massachusetts, USA, 15–17 June 1983), 137–144.

[20] M. Rabkin, “Recognizing two-sided contexts in cubic time”, *Computer Science—Theory and Applications* (CSR 2014, Moscow, Russia, 6–12 June 2014), LNCS 8476, to appear.

[21] W. C. Rounds, “LFP: A logic for linguistic descriptions and an analysis of its complexity”, *Computational Linguistics*, 14:4 (1988), 1–9.

[22] I. H. Sudborough, “A note on tape-bounded complexity classes and linear context-free languages”, *Journal of the ACM*, 22:4 (1975), 499–500, DOI: `10.1145/321906.321913`.

[23] L. G. Valiant, “General context-free recognition in less than cubic time”, *Journal of Computer and System Sciences*, 10:2 (1975), 308–314, DOI: `10.1016/S0022-0000(75)80046-8`.

[24] R. Zier-Vogel, M. Domaratzki, “RNA pseudoknot prediction through stochastic conjunctive grammars”, *Computability in Europe 2013. Informal Proceedings*, 80–89.

Analyzing Catastrophic Backtracking Behavior in Practical Regular Expression Matching

Martin Berglund
Department of Computing Science,
Umeå University,
Umeå, Sweden
mbe@cs.umu.se

Frank Drewes
Department of Computing Science,
Umeå University,
Umeå, Sweden
drewes@cs.umu.se

Brink van der Merwe
Department of Mathematical Sciences,
Computer Science Division,
University of Stellenbosch,
Stellenbosch, South Africa
abvdm@cs.sun.ac.za

We develop a formal perspective on how regular expression matching works in Java[1], a popular representative of the category of regex-directed matching engines. In particular, we define an automata model which captures all the aspects needed to study such matching engines in a formal way. Based on this, we propose two types of static analysis, which take a regular expression and tell whether there exists a family of strings which makes Java-style matching run in exponential time.

1 Introduction

Regular expressions constitute a concise, powerful, and useful pattern matching language for strings. They are commonly used to specify token lexemes for scanner generation during compiler construction, to validate input for web-based applications, to recognize meaningful patterns in natural language processing and data mining, for example, locating e-mail addresses, and to guard against computer system intrusion. Libraries for their use are found in most widely-used programming languages.

There are two fundamentally different types of regex matching engines: DFA (Deterministic Finite Automaton) and NFA (Non-deterministic Finite Automaton) matching engines. DFA matchers are used in (most versions of) awk, egrep, and in MySQL, and are based on the NFA to DFA subset conversion algorithm. This paper deals with NFA engines, which are found in GNU Emacs, Java, many command line tools, .NET, the PCRE (Perl compatible regular expressions) library, Perl, PHP, Python, Ruby and Vim. NFA matchers make use of an input-directed depth-first search on an NFA, and thus the matching performed by NFA engines is referred to as backtracking matching. NFA engines have made it possible to extend regular expressions with captures, possessive quantifiers, and backreferences.

Theory has however not kept pace with practice when it comes to understanding NFA engines. We now have NFA matchers that are more expressive and succinct than the originally developed DFA matchers, but are also in some cases significantly slower. Although it is known that in the worst case, the matching time of NFA matchers is exponential in the length of input strings [7], their performance characteristics and operational matching semantics are poorly understood in general. Exponential matching time, also referred to as catastrophic backtracking (by NFA matchers), can of course be avoided by using the DFA matchers, but then a less expressive pattern matching language has to be used. Catastrophic backtracking has potentially severe security implications, as denial-of-service attacks are possible in any application which matches a regular expression to data not carefully controlled by the application.

This work was motivated by the algorithm presented by Kirrage et. al. in [7], which for regular expressions with catastrophic backtracking comes up with a family of strings exhibiting this exponential

[1]Java is a registered trademark of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

matching time behavior. However, they only consider the case where the exponential matching behavior can be exhibited by strings that are rejected. We investigate the complexity of deciding exponential backtracking matching on strings that are rejected (which we refer to as deciding exponential failure backtracking) further, and in addition we consider the general case of exponential backtracking. For this we introduce prioritized NFA (pNFA), which make non-deterministic choices in an ordered manner, thus prioritizing some over others in a way very reminiscent of parsing expression grammars (PEGs). The latter introduce ordered choice to the world of context-free grammars [5]. An interesting algorithm bridging the two areas is given in [8] by translating extended regular expressions to PEGs.

By linking failure backtracking with ambiguity in NFA, we show that catastrophic failure backtracking can be decided in polynomial time, and in the case of polynomial failure backtracking, the degree of the polynomial can be determined in polynomial time. General backtracking is shown decidable in EXPTIME by associating a tree transducer with the expression and applying a result from [4].

2 Preliminaries

For a set A, we denote by $\mathscr{P}(A)$ the power set of A. The constant function $f\colon A \to B$ with $f(a) = b \in B$ for all $a \in A$ is denoted by b^A. Also, given any function $f\colon A \to B$ and elements $a \in A$, $b \in B$, we let $f_{a\mapsto b}$ denote the function f' such that $f'(a) = b$ and $f'(x) = f(x)$ for all $x \in A \setminus \{a\}$. The set of all strings (or sequences) over A is denoted by A^*. In particular, it contains the empty string ε. To avoid confusion, it is assumed that $\varepsilon \notin A$. The length of a string w is denoted by $|w|$, and the number of occurrences of $a \in A$ in w is denoted by $|w|_a$. The union of disjoint sets A and B is denoted by $A \uplus B$.

As usual, a regular expression over an alphabet Σ (where $\varepsilon, \emptyset \notin \Sigma$) is either an element of $\Sigma \cup \{\varepsilon, \emptyset\}$ or an expression of one of the forms $(E \,|\, E')$, $(E \cdot E')$, or (E^*), where E and E' are regular expressions. Parentheses can be dropped using the rule that * (Kleene closure) takes precedence over $\cdot$ (concatenation), which takes precedence over $|$ (union). Moreover, outermost parentheses can be dropped, and $E \cdot E'$ can be written as EE'. The language $\mathscr{L}(E)$ denoted by a regular expression is obtained by evaluating E as usual, where $\emptyset$ stands for the empty language and $a \in \Sigma \cup \{\varepsilon\}$ for $\{a\}$.

A *tree* with labels in a set Σ is a function $t\colon V \to \Sigma$, where $V \subseteq \mathbb{N}_+^*$ is a non-empty, finite set of vertices (or nodes) which are such that (i) V is prefix-closed, i.e., for all $v \in \mathbb{N}_+^*$ and $i \in \mathbb{N}_+$, $vi \in V$ implies $v \in V$; and; (ii) V is closed to the left, i.e., for all $v \in \mathbb{N}_+^*$ and $i \in \mathbb{N}_+$, $v(i+1) \in V$ implies $vi \in V$.

The vertex ε is the root of the tree and vertex vi is the ith child of v. We let $|t| = |V|$ denote the size of t. t/v denotes the tree t' with vertex set $V' = \{w \in \mathbb{N}_+^* \mid vw \in V\}$, where $t'(w) = t(vw)$ for all $w \in V'$. If V is not explicitly named, we may denote it by $V(t)$. The *rank* of a tree t is the maximum number of children of vertices of t. Given trees $t_1, \ldots, t_n$ and a symbol α, we let $\alpha[t_1, \ldots, t_n]$ denote the tree t with $t(\varepsilon) = \alpha$ and $t/i = t_i$ for all $i \in \{1, \ldots, n\}$. The tree $\alpha[]$ may be abbreviated by α.

Given an alphabet Σ, the set of all trees of the form $t\colon V \to \Sigma$ is denoted by T_Σ. Moreover, if Q is an alphabet disjoint with Σ, we denote by $T_\Sigma(Q)$ the set of all trees $t\colon V \to \Sigma \cup Q$ such that only leaves may be labeled with symbols in Q, i.e., $t(v) \in Q$ implies that $v \cdot 1 \notin V$.

A *non-deterministic finite automaton* (NFA) is a tuple $A = (Q, \Sigma, q_0, \delta, F)$ where Q is a finite set of states, Σ is an alphabet with $\varepsilon \notin \Sigma$, $q_0 \in Q$, $F \subseteq Q$ and $\delta\colon Q \times (\{\varepsilon\} \cup \Sigma) \to \mathscr{P}(Q)$ is the transition function. The fact that $p \in \delta(q, \alpha)$ may also be denoted by $q \xrightarrow{\alpha} p$.

A *run* on a string $w \in \Sigma^*$ is a sequence $p_1 \cdots p_{m+1} \in Q^*$ such that there exist $\alpha_1, \ldots, \alpha_m \in \Sigma \cup \{\varepsilon\}$ with $\alpha_1 \cdots \alpha_m = w$ and $p_i \xrightarrow{\alpha_i} p_{i+1}$ for all $i \in \{1, \ldots, m\}$. Such a run is *accepting* if $p_1 = q_0$ and $p_m \in F$. The string w is accepted by A if and only if there exist an accepting run on w. The set of strings in Σ^* that are accepted by A is denoted by $\mathscr{L}(A)$.

A *string-to-tree transducer* is a tuple $stt = (Q, \Sigma, \Gamma, q_0, \delta)$, where Σ and Γ are the input and output alphabets respectively, Q is the finite set of states, $q_0 \in Q$ is the initial state, and $\delta\colon Q \times \Sigma \to T_\Gamma(Q)$ is the transition function. When $\delta(q, \alpha) = t$ we also write $q \xrightarrow{\alpha} t$.

For $\alpha_1, \ldots, \alpha_n \in \Sigma$, $stt(\alpha_1 \cdots \alpha_n)$ is the set of all trees $t \in T_\Gamma$ such that there exists a sequence of trees $t_0, \ldots, t_n$ which fulfill the requirement that $t_0 = q_0$ and $t_n = t$; and for every $i \in \{1, \ldots, n\}$, t_i is obtained from t_{i-1} by replacing every leaf v for which $t_{i-1}(v) \in Q$ with a tree in $\delta(t_{i-1}(v), \alpha_i)$, i.e., it holds that $t_i/v \in \delta(t_{i-1}(v), \alpha_i)$.

3 Regular Expression Matching in Java

Here we will take a look at the algorithm used for matching regular expressions in Java, using the default `java.util.regex` package, and describe in pseudocode how matching is accomplished in this package. The Java implementation is a good representative of the class of NFA search matchers. It is both fairly typical and very consistent across different versions (Java 1.6.0u27 is used to generate figures here). Many other implementations behave similarly, e.g. the popular Perl Compatible Regular Expressions library (PCRE). We try to capture the essence of the Java matching procedure as accurately as possible while omitting details, add-ons, and tricks that are irrelevant for the purpose of this paper.

Let us first describe the Java matcher in some detail. Readers who are not interested in this description may skip ahead to the second last paragraph before Algorithm 1. The core of the matcher is implemented in `java.lang.regex.Pattern`. Given a regular expression, it constructs an object graph of subclasses of the class `java.lang.regex.Pattern$Node` (we briefly call it `Node`, assuming all classes to be inner classes of `java.lang.regex.Pattern` unless otherwise stated). `Node` objects correspond to states, encapsulating their transitions in addition, and have one relevant method, `boolean Node.match(Matcher m, int i, CharSequence s)`, which we will closely mimic later. The implicit `this` pointer corresponds to the state, `s` is the entire string, `i` is the index of the next symbol to be read. The argument `m` contains a variety of book-keeping, notably variables corresponding to C in Algorithm 1 below, as well as after-the-fact information regarding the accepting run found. (In contrast, `match` returns `true` if and only if the node can, potentially recursively, match the remainder of the string). Every `Node` contains at least a pointer `next` which serves as the “default” next transition out of the node. Let us look at the object graph on the left in Figure 1. There are quite a few nodes even for a small expression like ab^*, but most are needed for fairly minor book-keeping, and for features we are not concerned with here. For example **LastNode** checks that all symbols are read by the matching, but can be made to do other things using additional features in `java.util.regex` which we do not deal with.

The matching starts with a call to `match` on **Begin** with the full string (i.e., `i` set to one and the string in `s`). See Figure 2 for pseudo-code for the behavior of **Begin**, **Single** and **Curly**. **Begin** (and **LastNode**) are trivial, they just check that we are in the expected position of the string, and in the case of **Begin** calls its `next`. **Single** reads a single symbol (equal to its internal variable `c`) and continues to `next`. **Accept** is even more trivial and always returns true. **Curly** handles the Kleene closure, and, since it has to resolve non-determinism (i.e. how many repetitions to perform), it is a bit more complex. The values `type`, `cmin`, and `cmax` are irrelevant for our concerns, they implement the counted repetition extension. Note that that line 2 in the right-most code snippet in Figure 2 works by updating values in the “`m`” in-out argument, but we leave that unspecific here. **Curly** starts by trying to match the atom node `atom` to a prefix of the string. If it succeeds it calls itself recursively, calling match on `this`, to process the remainder. When `atom` fails to match any further, **Curly** instead continues to `next` (backtracking as needed). In reality **Curly** uses imperative loops for efficiency, but it only serves to achieve a constant

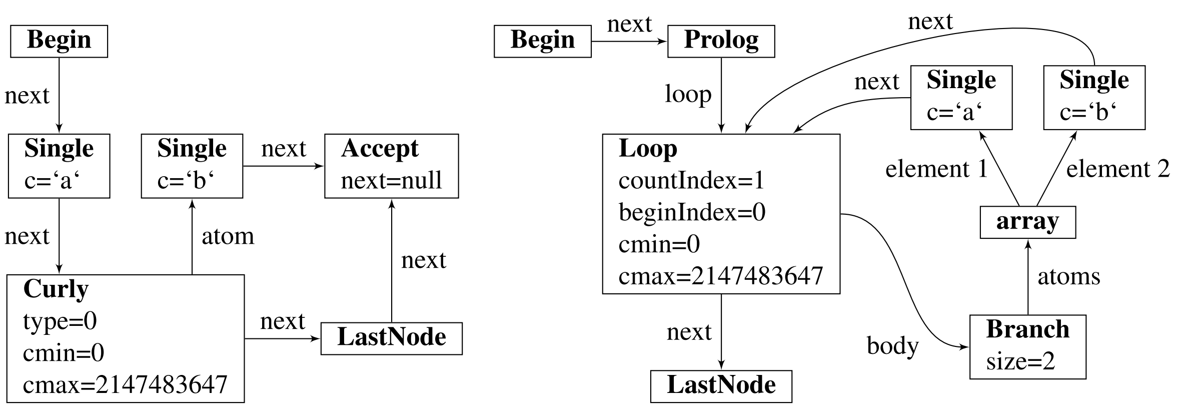


Figure 1: The left diagram shows essentially the complete internal object graph (i.e. internal data-structure) of subclasses of `Node` Java constructs for ab^*. On the right we show a simplified version of the corresponding object graph for $(a|b)^*$. In the latter all nodes without matching effect (in our limited expressions) are removed (e.g. the node **Accept** seen in the more complete example on the left).

```
if $i = 0$ then
    return next.match($i, w$)
else
    return false
end if
```

```
if $\alpha_i = $ c then
    return next.match($i+1, w$)
else
    return false
end if
```

```
if atom.match($i, w$) then
    $b :=$ #symbols read above
    if this.match($i+b, w$) then
        return true
    end if
end if
return next.match($i, w$)
```

Figure 2: The code for a call of the form `match`$(i, w = \alpha_1 \cdots \alpha_n)$ on a **Begin** (left), **Single** (middle) and **Curly** (right) node. **Single** has a member variable `c` identifying the symbol it should read. **Curly** tries to recursively repeat `atom`, calling `next` when that fails.

speedup and is as such irrelevant for us. **Curly** is not used for all Kleene closures, if $b = 0$ it would loop forever, so the construction procedure for the object graph only uses **Curly** when it (with a fairly limited decision procedure) can tell that the contents looped is of constant non-zero length.

Next we look at the more general example on the right side of Figure 1. Here there are some additional nodes to consider. **Branch** implements the union, and **Prolog** and **Loop** implement the Kleene closure (with **Prolog** calling `matchInit` on **Loop** to initialize the loop). Let us look at each function in Figure 3. In **Branch**, `match` starts by letting the first subexpression match, continuing with the second and so on if the first attempts fail. The symbiotic relationship between **Prolog** and **Loop** is trickier. Where all other nodes calls into **Loop** with `match`(i, w) as usual **Prolog** calls in with `matchInit`(i, w) (on the left in Figure 3). This serves only one purpose: it eliminates ε-cycles. That is, it prevents **Loop** from recursively matching `body` to the empty string, making no progress. In `matchInit` the current value of i is stored, and in `match` (in the middle in Figure 3) an attempt to match `body` will only be made if at least one symbol has been read since the last attempt.

As an additional example, consider the regular expression $(a|a)^*$, which has an object graph almost like on the right of Figure 1, except the second **Single** *also* has `c` set to a. Matching this against $aa \cdots ab$ will take exponential time in the number of as, as all ways to match each a to each **Single** in $a|a$ will be tried as the matching backtracks trying to match the final b. In an experiment on one of the authors'

```
1: this.sp := $i$
2: if body.match($i,w$) then
3:     return true
4: else
5:     return next.match($i,w$)
6: end if
```

```
1: if $i >$ this.sp then
2:     if body.match($i,w$) then
3:         return true
4:     end if
5: end if
6: return next.match($i,w$)
```

```
1: for $e \in$ array do
2:     if $e$.match($i,w$) then
3:         return true
4:     end if
5: end for
6: return $false$
```

Figure 3: The code for a call of the form `match`$(i, w = \alpha_1 \cdots \alpha_n)$ on a **Loop** (middle), **Branch** (right) and, as a special case, the call `matchInit`(i,w) on **Loop** on the left. `matchInit` is called by **Prolog** in lieu of `match`. Notice that the loop in **Branch** is *in array order*.

desktop PCs an attempt to match $(a|a)^*$ to $a^{35}b$ using Java took roughly an hour of CPU time.

The object graph on the right in Figure 1 is, as is noted in the caption, a bit of creative editing of reality. A number of nodes not affecting the search behavior or matching are removed: the **Accept** node, which is just a `next` placeholder with no effect, **GroupHead** and **GroupTail**, which tracks what part of the match corresponds to a parenthesized subexpression, and finally **BranchConn**, which is placed in relation to **Branch** in the right of Figure 1 and records some information for the optimizer.

In general all nodes have numerous additional features not discussed, and there are many additional nodes serving similar purposes. For example **Single** may be replaced with **Slice** to match multiple symbols at once or **BnM** to matche multiple symbols using Boyer-Moore matching [2]. However, the optimizations are too minor to matter for our concerns (e.g. using **Slice** and **BnM** instead of **Single** yields at most a linear speed-up), and the additional features are outside our scope.

Let us take the above together and assemble the snippets of code into a function which takes a regular expression and a string as input and decides if the expression matches the string. A regular expression is represented by its parse tree, $T\colon \mathbb{N}_+^* \to \{|, {}^*, {}^{*?}, \cdot, \varepsilon\} \cup \Sigma$, defined in the obvious way with each $\cdot$ and $|$ having two children, * and $^{*?}$ one, and $\alpha \in \Sigma \cup \{\varepsilon\}$ zero. The operator $^{*?}$ is the lazy Kleene closure, which is the same as *, except that it attempts to make as *few* repetitions as possible.

We now define a function $next : \mathbb{N}_+^* \to \mathbb{N}_+^* \uplus \{nil\}$ on the nodes of T, where nil is a special value. Roughly speaking, $next(v)$ is the node at which parsing continues when the subexpression rooted av v has successfully matched (compare to the `cont` pointers in Kirrage at al. [7]). Let $next(\varepsilon) = nil$, and

1. if $T(v) = |$ then $next(v \cdot 1) = next(v \cdot 2) = next(v)$;
2. if $T(v) = \cdot$ then $next(v \cdot 1) = v \cdot 2$ and $next(v \cdot 2) = next(v)$; and
3. if $T(v) = {}^*$ or $T(v) = {}^{*?}$ then $next(v \cdot 1) = v$.

Then, collapsing the object graph and ignoring precise node choices in Java we get Algorithm 1.

Algorithm 1. *Simplified pseudocode of the Java matching algorithm. The implicit regular expression parse tree is T. The call-by-value input parameters are the node of T currently processed, the remainder of the input string, and a set of nodes that we should not revisit before consuming the next input symbol. This prevents ε-cycles as discussed above. The initial call made is* $\textsc{Match}(\varepsilon, w, \emptyset)$.

```
1: function MATCH($v, w = a_1 \cdots a_n, C$)
2:     if $v = nil$ then
3:         return $n = 0$
4:     else if $T(v) = \varepsilon$ then
5:         return MATCH($next(v), w, C$)
6:     else if $T(v) \in \Sigma$ then
7:         if $n \geq 1 \land T(v) = a_1$ then
8:             return MATCH($next(v), a_2 \cdots a_n, \emptyset$)
9:         end if
10:        return $false$
11:    else if $T(v) = |$ then
12:        if MATCH($v \cdot 1, w, C$) then
```

```
13:         return true
14:     end if
15:     return MATCH($v \cdot 2, w, C$)
16: else if $T(v) = \cdot$ then
17:     return MATCH($v \cdot 1, w, C$)
18: else if $T(v) = {}^*$ then
19:     if $v \cdot 1 \notin C$ then
20:         if MATCH($v \cdot 1, w, C \cup \{v \cdot 1\}$) then
21:             return true
22:         end if
23:     end if
24:     return MATCH($next(v), w, C$)
25: else if $T(v) = {}^{*?}$ then
26:     if MATCH($next(v), w, C$) then
27:         return true
28:     else if $v \cdot 1 \notin C$ then
29:         return MATCH($v \cdot 1, w, C \cup \{v \cdot 1\}$)
30:     else
31:         return false
32:     end if
33: end if
34: end function
```

Notice how the code for the two Kleene closure variants *only* differs in what they try first: * tries to repeat its body first, whereas $^{*?}$ tries to not repeat the body. Note also how C is used to prevent ε-cycles in lines 19–20 and 28–29. If the node we *would* go to is already in C this means that no symbol has been read since last time we tried this, meaning repeating it would be a loop without progress.

4 Prioritized Non-Deterministic Finite Automata

We now define a modified type of NFA that provides us with an abstract view of the matching procedure discussed in the previous section. The modifications have no impact on the language accepted, but make the automaton "run deterministic". Every string in the language accepted has a unique accepting run, a property brought about by ordering the non-deterministic choices into a first, second, etc alternative, and letting the unique accepting run be given by trying, at any given state, alternative $i+1$ only when alternative i has failed. In our definition, only ε-transitions can be nondeterministic.

Definition 2. *A* prioritized non-deterministic finite automaton *(*pNFA*) is a tuple $A = (Q_1, Q_2, \Sigma, q_0, \delta_1, \delta_2, F)$, where Q_1 and Q_2 are disjoint finite sets of states; Σ is a finite alphabet; $q_0 \in Q_1 \cup Q_2$ is the initial state; $\delta_1 \colon Q_1 \times \Sigma \to (Q_1 \cup Q_2)$ is the deterministic transition function; $\delta_2 \colon Q_2 \to (Q_1 \cup Q_2)^*$ is the non-deterministic prioritized transition function; and $F \subseteq Q_1 \cup Q_2$ are the final states.*

The NFA corresponding to the pNFA A is given by $\overline{A} = (Q_1 \cup Q_2, \Sigma, q_0, \bar{\delta}, F)$, where

$$\bar{\delta}(q,\alpha) = \begin{cases} \{\delta_1(q,\alpha)\} & \text{if } q \in Q_1 \text{ and } \alpha \in \Sigma, \\ \{q_1, \ldots, q_n\} & \text{if } q \in Q_2,\ \alpha = \varepsilon, \text{ and } \delta_2(q) = q_1 \cdots q_n. \end{cases}$$

The language accepted by *A, denoted by $\mathscr{L}(A)$, is $\mathscr{L}(\overline{A})$.*

Next, we define the so-called backtracking run of a pNFA on an input string w. This run takes the form of a tree which, intuitively, represents the attempts a matching algorithm such as Algorithm 1 would make until accepting the input string (or eventually rejecting it). The definition makes use of a parameter C whose purpose is to remember, for every state, the highest nondeterministic alternative that has been tried since the last symbol was consumed. This corresponds to the parameter C in Algorithm 1 and avoids infinite runs caused by ε-cycles.

Definition 3. *Let $A = (Q_1, Q_2, \Sigma, q_0, \delta_1, \delta_2, F)$ be a pNFA, $q \in Q_1 \cup Q_2$, $w = \alpha_1 \cdots \alpha_n \in \Sigma^*$, and $C \colon Q_2 \to \mathbb{N}$. Then the (q,w,C)-*backtracking run *of A is a tree over $Q_1 \cup Q_2 \uplus \{Acc, Rej\}$. It* succeeds *if and only if Acc occurs in it. We denote the (q,w,C)-backtracking run by $btr_A(q,w,C)$ and inductively define it as follows. If $q \in F$ and $w = \varepsilon$ then $btr_A(q,w,C) = q[Acc]$. Otherwise, we distinguish between two cases:*[2]

[2]For the first case, recall that 0^{Q_2} denotes the function $C \colon Q_2 \to \mathbb{N}$ such that $C(q) = 0$ for all $q \in Q_2$.

1. *If $q \in Q_1$, then*

$$btr_A(q,w,C) = \begin{cases} q[btr_A(\delta_1(q,\alpha_1),\alpha_2\cdots\alpha_n,0^{Q_2})] & \text{if } n>0 \text{ and } \delta_1(q,\alpha_1) \text{ is defined,} \\ q[Rej] & \text{otherwise.} \end{cases}$$

2. *If $q \in Q_2$ with $\delta_2(q) = q_1\cdots q_k$, let $i_0 = C(q)+1$ and $r_i = btr_A(q_i,w,C_{q\mapsto i})$ for $i_0 \le i \le k$. Then*

$$btr_A(q,w,C) = \begin{cases} q[Rej] & \text{if } i_0 > k, \\ q[r_{i_0},\dots,r_k] & \text{if } i_0 \le k \text{ but no } r_i \ (i_0 \le i \le k) \text{ succeeds,} \\ q[r_{i_0},\dots,r_i] & \text{if } i \in \{i_0,\dots,k\} \text{ is the least index such that } r_i \text{ succeeds.} \end{cases}$$

The backtracking run of *A* on *w is $btr_A(w) = btr_A(q_0,w,0^{Q_2})$. If $btr_A(w)$ succeeds, then the* accepting run of *A* on *w is the sequence of states on the right-most path in $btr_A(w)$.*

Notice that the third parameter C in $btr_A(q,w,C)$ fulfills a similar purpose as the set C in Algorithm 1. It is used to track transitions that must not be revisited to avoid cycles.

Clearly, for a pNFA A and a string w, $w \in \mathscr{L}(A)$ if and only if $btr_A(w)$ succeeds, if and only if the accepting run of A on w is an accepting run of the NFA $\overline{A}$. Backtracking runs capture the behavior of the following algorithm which generalizes Algorithm 1 to arbitrary pNFAs to deterministically find the accepting run of A on w if it exists.

Algorithm 4. *Let $A = (Q_1,Q_2,\Sigma,q_0,\delta_1,\delta_2,F)$ be a pNFA. The call* MATCH$(q_0,w,0^{Q_2})$ *of the following procedure yields the accepting run of A on w if it exists, and $\perp \notin Q_1 \cup Q_2$ otherwise. The third parameter is similar to the C in Definition 3. For every state $q \in Q_2$ with out-degree d we have $C(q) \in \{0,\dots,d\}$.*

```
 1: function MATCH($q,w = a_1\cdots a_n,C$)
 2:   if $q \in Q_1$ then
 3:     if $n = 0$ then
 4:       if $q \in F$ then
 5:         return $q$
 6:       else
 7:         return $\perp$
 8:       end if
 9:     else
10:       return $q \cdot$ MATCH$(\delta_1(q,a_1),a_2\cdots a_n,0^{Q_2})$
11:     end if
12:   else
13:     if $n = 0 \wedge q \in F$ then
14:       return $q$
15:     else
16:       $q_1\cdots q_k := \delta_2(q)$
17:       for $i = C(q)+1,\dots,k$ do
18:         $r :=$ MATCH$(q_i,w,C_{q\mapsto i})$
19:         if $r \neq \perp$ then
20:           return $q \cdot r$
21:         end if
22:       end for
23:       return $\perp$
24:     end if
25:   end if
26: end function
```

Notice especially line 10 where a symbol is read and C is reset to 0^{Q_2} in the recursive call. The case for Q_2 starts at line 13, the loop at 17 tries all not yet tried *transitions for that state. If no transition succeeds we fail on line 23.*

We note here that the running time of Algorithm 4 is exponential in general, just like Algorithm 1. This can be remedied by means of memoization, but potentially with a significant memory overhead, due to the fact that memoization needs to keep track of each possible assignment to all $C(q)$ with $q \in Q_2$.[3]

Depending on how one turns a given regular expression into a pNFA, Algorithm 4 will run more or less efficiently. For example, if the pNFA is built in a way that reflects Algorithm 1, analyzing the efficiency of Algorithm 4 or, equivalently, the size of backtracking runs, yields a (somewhat idealized) statement about the efficiency of the Java matcher.

[3]Apparently, starting from version 5.6, Perl uses memoization in its regular expression engine in order to speed up matching.

4.1 Two Constructions for Turning Regular Expressions into pNFA

In this section we give two examples of constructions that can be used to turn a regular expression E into a pNFA A such that $\mathscr{L}(A) = \mathscr{L}(E)$. The first is a prioritized version of the classical Thompson construction [9], whereas the second follows the Java approach.

Recall that the classical Thompson construction converts the parse tree T of a regular expression E to an NFA, which we denote by $Th(E)$, by doing a postorder traversal on T. An NFA is constructed for each subtree T' of T, equivalent to the regular expression represented by T'. We do not repeat this well-known construction here, assuming that the reader is familiar with it. Instead, we define a prioritized version, which constructs a pNFA denoted by $Th^p(E)$ such that $\overline{Th^p(E)} = Th(E)$.

Just as the construction for $Th(E)$, we define $Th^p(E)$ recursively on the parse tree for E. For each subexpression F of E, $Th^p(F)$ has a single initial state with no ingoing transitions, and a single final state with no outgoing transitions. The constructions of $Th^p(\emptyset)$, $Th^p(\varepsilon)$, $Th^p(a)$, and $Th^p(F_1 \cdot F_2)$, given that $Th^p(F_1)$ and $Th^p(F_2)$ are already constructed, are defined as for $Th(E)$, splitting the state set into Q_1 and Q_2 in the obvious way. It is only when we construct $Th^p(F_1|F_2)$ from $Th^p(F_1)$ and $Th^p(F_2)$, and $Th^p(F_1^*)$ from $Th(F_1)$, where the priorities of introduced ε-transitions require attention. We also consider the lazy Kleene closure $F_1^{*?}$, to illustrate the difference in priorities of transitions between constructions for the greedy and lazy Kleene closure. In each of the constructions below, we assume that $Th^p(F_i)$ ($i \in \{1,2\}$) has the initial state q_i and the final state f_i. Furthermore, δ_2 denotes the transition function for ε-transitions in the newly constructed pNFA $Th^p(E)$. All non-final states in $Th^p(E)$ that are in $Th^p(F_i)$ inherit their outgoing transitions from $Th^p(F_i)$.

- If $E = F_1|F_2$ then $Th^p(E)$ is built like $Th(E)$, thus introducing new initial and final states q_0 and f_0, respectively, and defining $\delta_2(q_0) = q_1q_2$ and $\delta_2(f_1) = \delta_2(f_2) = f_0$.
- If $E = F_1^*$ then we add new initial and final states q_0 and f_0 to Q_2 and define $\delta_2(q_0) = q_1f_0$ and $\delta_2(f_1) = q_1f_0$. The case $E = F_1^{*?}$ is the same, except that $\delta_2(q_0) = f_0q_1$ and $\delta_2(f_1) = f_0q_1$.

Thus, the pNFA $Th^p(F^*)$ tries F as often as possible whereas $Th^p(F^{*?})$ does the opposite.

The second pNFA construction is the one implicit in the Java approach and Algorithm 1. We denote this pNFA by $J^p(E)$. The base cases $J^p(\emptyset)$, $J^p(\varepsilon)$, $J^p(a)$ are identical to $Th^p(\emptyset)$, $Th^p(\varepsilon)$, $Th^p(a)$, respectively. Now, let us consider the remaining operators. Again, we assume that $J^p(F_i)$ ($i \in \{1,2\}$) has the initial state q_i and the final state f_i. Furthermore, δ_2 denotes the transition function for ε-transitions in the newly constructed pNFA $J^p(E)$.

- Assume that $E = F_1 \cdot F_2$. Then $J^p(E)$ is built from $J^p(F_1)$ and $J^p(F_2)$ by identifying f_1 with q_2, adding a new initial state $q_0 \in Q_2$ with $\delta_2(q_0) = q_1$, and making f_2 the final state. Thus, $J^p(E)$ is built like $Th^p(E)$, except that a new initial state is added and connected to the initial state of $J^p(F_1)$ by means of an ε-transition.
- If $E = F_1|F_2$ then $J^p(E)$ is constructed by introducing a new initial state q_0, defining $\delta_2(q_0) = q_1q_2$, and identifying f_1 and f_2, the result of which becomes the new final state.
- Now assume that $E = F_1^*$. Then we add a new final state f_0 to $J^p(F_1)$, make $q_0 = f_1$ the initial state of $J^P(E)$, and set $\delta_2(q_0) = q_1f_0$. The case $E = F_1^{*?}$ is exactly the same, except that $\delta_2(q_0) = f_0q_1$.

Observation 5. *Let E be a regular expression and A a pNFA. Then the running time of Algorithm 4 on w (with respect to E) is $\Theta(|btr_A(w)|)$.*

The two variants of implementing regular expressions by pNFA are closely related. In fact, Kirrage et al. [7] seem to regard them as being essentially identical and write that their reasons for choosing

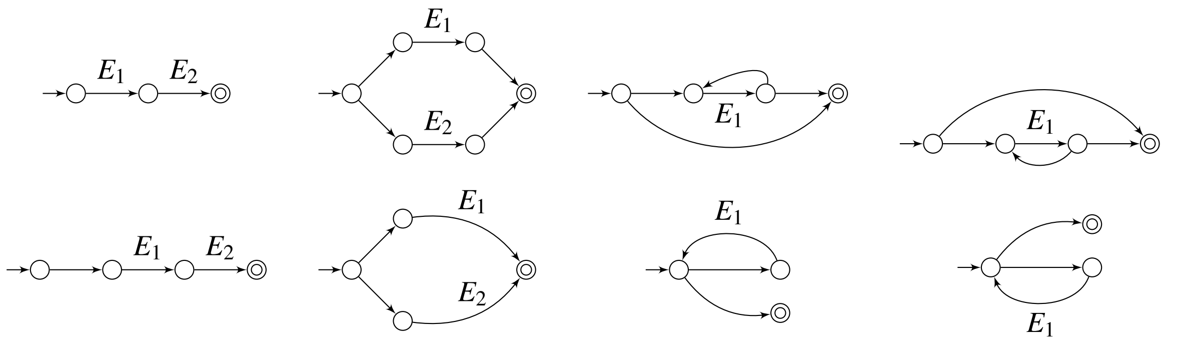
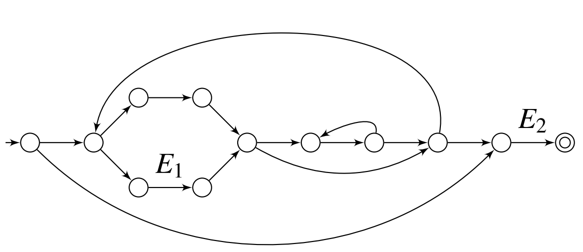


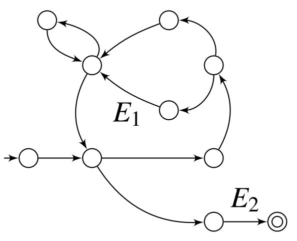
Figure 4: Abstract pNFA corresponding to $E_1 \cdot E_2$, $E_1 \,|\, E_2$, E_1^* and $E_1^{*?}$, from which $Th^p(E)$ (top row) and $J^p(E)$ (bottom row) are constructed. The transitions are prioritized in clockwise order, starting at noon.

$J^p(E)$ are "purely of presentational nature". However, using our notion of pNFA we can show that this is not always the case. For this, note first that the construction of both $Th^p(E)$ and $J^p(E)$ can be viewed in a top-down fashion, where each operation is represented by an abstract pNFA in which zero, one, or two transitions are labeled with regular expressions. Replacing such a transition with the corresponding pNFA yields the constructed pNFA for the whole expression. Figure 4 shows the building blocks for the operations $\cdot$, $|$, *, and $^{*?}$ in both cases. Priorities follow the convention that ε-transitions leaving a state are drawn in clockwise order, starting at noon. Unlabeled edges denote ε-transitions.

Now consider an expression E of the form $((\varepsilon|E_1) \cdot \varepsilon^*)^* \cdot E_2$. When building $Th^p(E)$ and $J^p(E)$, these correspond to the following abstract pNFA:



and



In $Th^p(E)$, when processing an input string w, the run will first choose the prioritized choice of the union operator (which is ε), iterate the inner loop once, and then return to the initial state of the sub-pNFA corresponding to $\varepsilon|E_1$. Now, the first alternative is blocked, meaning that Algorithm 4 tries to match E_1. Assuming that no failure occurs, it will then proceed by following ε transitions leading to E_2.

Now look at $J^p(E)$. Here, the run first bypasses E_1, similarly to $Th^p(E)$, but this leads to the state following the start state. As the first alternative of transitions leaving this state has already been used, the run drops out of the loop and proceeds with E_2. E_1 will only be tried after backtracking in case E_2 fails.

We thus get several cases by appropriately instantiating E_1 and E_2. Assume first that we choose E_1 in such a way that $Th^p(E_1)$ suffers from exponential backtracking on a set W of input strings over Σ, and $E_2 = \Sigma^*$. Then $Th^p(E)$ causes exponential backtracking on strings in W whereas $J^p(E)$ does not backtrack at all. A concrete example is obtained by taking $\Sigma = \{a,b\}$, $E_1 = (a^*)^*$, and $W = \{a^n b \mid n \in \mathbb{N}\}$.

Conversely, we may choose $E_2 = \varepsilon \,|\, E_2'$ so that $J^p(E_2')$ fails exponentially on W, but $E_1 = \Sigma^*$. Then $Th^p(E)$ will match strings in W in linear time whereas $J^p(E)$ will take exponential time.

One can easily combine two examples of the types above into one, to obtain an expression such that $Th^p(E)$ shows exponential behavior on a set W of strings on which $J^p(E)$ runs in linear time whereas $J^p(E)$ shows exponential behavior on another set W' of strings on which $Th^p(E)$ runs in linear time.

5 Static Analysis of Exponential Backtracking

We now consider the problem of deciding whether a given pNFA causes backtracking matching similar to Algorithm 1 to run exponentially. More precisely, we ask whether a pNFA has exponentially large backtracking runs. In the case where the considered pNFA is $J^p(E)$, this yields a statement about the running time of Algorithm 1. However, we are interested in the problem in general, because other regular expression engines may correspond to other pNFA. There are two variants of the decision problem, with very different complexities. Let us start by defining the first.

Definition 6. *Given a pNFA $A = (Q_1, Q_2, \Sigma, q_0, \delta_1, \delta_2, F)$, let $f(n) = \max\{|btr_A(w)| \mid w \in \Sigma^*, |w| \leq n\}$ for all $n \in \mathbb{N}$. We say that A has* exponential backtracking *if $f \in 2^{\Omega(n)}$ (or equivalently, if $f(n) \in 2^{\Theta(n)}$) and* polynomial backtracking of degree k *for $k \in \mathbb{N}$ if $f \in \Theta(n^{k+1})$.*

If the pNFA $A^f = (Q_1, Q_2, \Sigma, q_0, \delta_1, \delta_2, \emptyset)$, has exponential backtracking (or polynomial backtracking), then we say that A has exponential failure backtracking *(polynomial failure backtracking, resp.).*

Failure backtracking provides an upper bound for the general case. In cases where the worst-case matching complexity can be exhibited by a family of strings not in $\mathscr{L}(A)$, this analysis is precise. This happens for example if for some $\$ \in \Sigma$, we have $w\$ \notin \mathscr{L}(A)$ for all $w \in \Sigma^*$, or more generally, if for each $w \in \Sigma^*$, there is $w' \in \Sigma^*$ such that $ww' \notin \mathscr{L}(A)$. Failure backtracking analysis is of great interest in that it is more efficiently decidable (being in PTIME) than the general case. It is closely related to the case considered in e.g. [7], where the matching complexity of the strings not in $\mathscr{L}(A)$ is studied.

5.1 An Upper Bound on the Complexity of General Backtracking Analysis

Let us first establish an upper bound on the complexity of general backtracking analysis. We will give an algorithm which solves this problem in EXPTIME. Afterwards, we will also note some minor hardness results. The EXPTIME decision procedure relies heavily on a result from [4].

Lemma 7. *Given a string-to-tree transducer $stt = (Q, \Sigma, \Gamma, q_0, \delta)$, it is decidable in deterministic exponential time whether the function $f(n) = \max\{|t| \mid t \in stt(s),\ s \in \Sigma^*,\ |s| \leq n\}$ grows exponentially, i.e. whether $f \in 2^{\Omega(n)}$.*

In short, we will hereafter construct a string-to-tree transducer from a pNFA A which reads an input string (suitably decorated) and outputs the corresponding backtracking run of A (see Definition 3). In this way, we model the running of Algorithm 4 on that string. Then Lemma 7 can be applied to this transducer to decide exponential backtracking. To simplify the construction we first make a small adjustment to the input pNFA in the form of a "flattening", which ensures that δ_2 maps Q_2 to Q_1^*. That is, we remove the opportunity for repeated ε-transitions.

Definition 8. *Let $A = (Q_1, Q_2, \Sigma, q_0, \delta_1, \delta_2, F)$ be a pNFA. Define $d\colon (Q_1 \cup Q_2) \times (Q_2 \to \mathbb{N}) \to Q_1^*$, and $\bar{r}\colon Q_1^* \to Q_1^*$ as follows:*

$$d(q, C) = \begin{cases} q & \text{if } q \in Q_1, \\ d(q_{i+1}, C_{q \mapsto i+1}) \cdots d(q_n, C_{q \mapsto i+1}) & \text{if } q \in Q_2,\ \delta_2(q) = (q_1 \cdots q_n) \text{ and } C(q) = i. \end{cases}$$

$$\bar{r}(s) = \begin{cases} \bar{r}(uv) & \text{if } s = uqv \text{ for some } u, v \in Q_1^* \text{ and } q \in Q_1 \text{ with } |u|_q \geq 2 \\ s & \text{otherwise.} \end{cases}$$

That is, $\bar{r}$ removes all repetitions of each state q beyond the first two occurrences.

Now, the δ_2-flattening *of A is the pNFA $A' = (Q_1, Q_2, \Sigma, q_0, \delta_1, \delta_2', F')$ with $\delta_2'(q) = \bar{r}(d(q, 0^{Q_2}))$ for all $q \in Q_2$, and $F' = \{q \in Q_1 \cup Q_2 \mid d(q, 0^{Q_2}) \cap F \neq \emptyset\}$.*

First let us note that the size of A' in Definition 8 is polynomial in the size of A, as no new states are added and no right-hand side is greater than polynomial in length ($2|Q_1|$ is the maximum length after applying $\bar{r}$). Furthermore, the construction itself can be performed in polynomial time in a straightforward way by computing d incrementally in a left-to-right fashion, and aborting each recursion visiting a state that has already been seen twice to the left.

Before proving some properties of the above construction we make a supporting observation.

Lemma 9. *Let σ be a function on trees such that, for $t = f[t_1,\dots,t_k]$*

$$\sigma(t) = \begin{cases} t & \text{if } k = 0 \\ f[\sigma(t_1)] & \text{if } k = 1 \\ f[\sigma(t_i),\sigma(t_j)] & \text{otherwise, where } t_i,t_j\ (i \neq j) \text{ are largest among } t_1,\dots,t_k. \end{cases}$$

Let $T_0,T_1,T_2,\dots$ be sets of trees of rank at most k. Then the function $f(n) = \max\{|t| \mid t \in T_n\}$ grows exponentially if and only if $f'(n) = \max\{|\sigma(t)| \mid t \in T_n\}$ grows exponentially.

We leave out the (rather easy) proof of the lemma due to space limitations.

Lemma 10. *Let $A = (Q_1,Q_2,\Sigma,q_0,\delta_1,\delta_2,F)$ be a pNFA and A' its δ_2-flattening. Then A' can be constructed in polynomial time, $\mathscr{L}(A') = \mathscr{L}(A)$, and the function $f(n) = \max\{|btr_A(w)| \mid w \in \Sigma^*,\ |w| \leq n\}$ grows exponentially if and only if $f'(n) = \max\{|btr_{A'}(w)| \mid w \in \Sigma^*,\ |w| \leq n\}$ grows exponentially.*

Proof sketch. Let $A' = (Q_1,Q_2,\Sigma,q_0,\delta_1,\delta_2',F')$. As noted, A' can be constructed in polynomial time.

The language equivalence of A and A' can be established by induction on the accepting runs of A and A'. δ_2' is a closure on δ_2, such that any accepting run for A of the form $p_1 \cdots p_n$ can be turned into one for A' by replacing each maximal subsequence $p_k \cdots p_{k+i} \in Q_2^*$ with just p_k. The function d in the construction of δ_2 will ensure that p_k is accepting if this was at the end of the run, and that p_k can go directly to the following Q_1 state. The converse is equally straightforward, as a suitable sequence from Q_2 can be inserted into an accepting run for A' to create a correct accepting run for A.

Finally, we argue that A' exhibits exponential backtracking behavior if and only if A does. By the construction of A', we have $btr_{A'}(w) \leq btr_A(w)$. Hence, f grows exponentially if f' does. It remains to consider the other direction. Thus, assume that $f(n)$ grows exponentially. We have to show that $f'(n)$ grows exponentially as well. Let A'' be the pNFA generated by δ_2-flattening A without applying $\bar{r}$. Let $t = btr_A(w)$ and $t'' = btr_{A''}(w)$ for some input string w. Then t'' is obtained from t by repeatedly replacing subtrees of the form $q[s_1,\dots,s_k,q'[t_1,\dots,t_l],s_{k+1},\dots,s_m]$, where $q,q' \in Q_2$, by $q[s_1,\dots,s_k,t_1,\dots,t_l,s_{k+1},\dots,s_m]$. Since Definition 3 prevents repeated ε-cycles, this process removes only a constant fraction of the nodes in t.[4] Hence, $f''(n) = \max\{|btr_{A''}(w)| \mid w \in \Sigma^*,\ |w| \leq n\}$ grows exponentially. Now, compare t'' with $t' = btr_{A'}(w)$. If a node of t'' has m children with the same state $q \in Q_2$ in their roots, by the definition of backtracking runs the m subtrees rooted at those nodes will be identical. This is the case since the run for each subtree starts in the same state and string position, and the application of d in the partial flattening ensures that C is made irrelevant by an immediately following δ_1 transition resetting it to 0^{Q_2}. The application of $\bar{r}$ to A'' means that, in effect, the first two copies of these m subtrees are kept in t'. In particular, the two largest subtrees of the node are kept in t'. According to Lemma 9, this means that g' grows exponentially. □

It should be noticed that, for the proof above to be valid, it is important that $\bar{r}$ preserves the order of occurrences of states from the left, as a subtree being accepting means that no further subtrees are constructed to the right of it (ensuring no extraneous subtrees get included).

[4]The constant may be exponential in the size of A, but for the question at hand this does not matter since the backtracking behavior in the length of the string is what is considered.

We are now prepared to define the construction which for any δ_2-flattened pNFA A produces a string-to-tree transducer *stt* such that $btr_A(w) = t$ if and only if $t \in stt(w')$. Here, w' is a version of w decorated with extra symbols $\flat$ and \$. The former will serve as padding to be read when δ_2 transitions are taken, and \$ marks the beginning and the end of the string.

Definition 11. *Given a δ_2-flattened pNFA $A = (Q_1, Q_2, \Sigma, q_0, \delta_1, \delta_2, F)$ we construct the string-to-tree transducer $stt = (Q, \Sigma', \Gamma, q'_0, \delta)$ in the following way. $Q = \{q'_0\} \cup \{a_q, f_q \mid q \in Q_1 \cup Q_2\}$, $\Sigma' = \Sigma \uplus \{\flat, \$\}$, and $\Gamma = Q_1 \cup Q_2 \uplus \{Acc, Rej\}$. Furthermore, δ consists of the following transitions:*

1. *Let $q'_0 \xrightarrow{\$} a_{q_0}$ and $q'_0 \xrightarrow{\$} f_{q_0}$. For all $q \in Q$ let $q \xrightarrow{\flat} q$.*
2. *For all $q \in Q_1$ and $\alpha \in \Sigma$:*
   (a) *If $\delta_1(q, \alpha) = q'$ let $a_q \xrightarrow{\alpha} q[a_{q'}]$ and $f_q \xrightarrow{\alpha} q[f_{q'}]$.*
   (b) *If $\delta_1(q, \alpha)$ is undefined let $f_q \xrightarrow{\alpha} q[Rej]$.*
3. *For all $q \in Q_2$, if $q_1 \cdots q_n = \delta_2(q)$, then for all $i \in \{0, \ldots, n-1\}$ let $a_q \xrightarrow{\flat} q[f_{q_1}, \ldots, f_{q_i}, a_{q_{i+1}}]$, and let $f_q \xrightarrow{\flat} q[f_{q_1}, \ldots, f_{q_n}]$.*
4. *Finally if $q \in F$ let $a_q \xrightarrow{\$} q[Acc]$, whereas when $q \notin F$:*
   (a) *if $q \in Q_1$ let $f_q \xrightarrow{\$} q[Rej]$, and,*
   (b) *if $q \in Q_2$ and $q_1 \cdots q_n = \delta_2(q)$, then $f_q \xrightarrow{\$} q[q_1[Rej], \ldots, q_n[Rej]]$.*

Definition 12. *The string $w_1\alpha_1 w_2\alpha_2 \cdots w_n\alpha_n w_{n+1}$ is a* decoration *of $\alpha_1 \cdots \alpha_n \in \Sigma^*$ if $w_i \in \{\$, \flat\}^*$ for each i. $\$\flat\alpha_1\flat\alpha_2 \cdots \flat\alpha_n\$$ is the* correct decoration *of $\alpha_1 \cdots \alpha_n$, denoted $dec(\alpha_1 \cdots \alpha_n)$.*

Lemma 13. *For a δ_2-flattened pNFA A, the string-to-tree transducer stt as constructed by Definition 11, and an input string $w = \alpha_1 \cdots \alpha_n$, it holds that $stt(dec(w)) = \{btr_A(w)\}$. For* all *$u$ which are decorations of w either $stt(u) = \emptyset$ or $stt(u) = \{btr_A(w)\}$.*

Proof sketch. First, notice how A being δ_2-flattened impacts btr_A. The flattening ensures that there is no way to take two ε-transitions in a row in A, meaning that every time case 2 of Definition 3 applies, we have $C(q) = 0$ since the previous step is either the initial call or a call from case 1 where C gets reset. As such we will have $C = 0^{Q_2}$ in every recursive call below. Let stt_q denote the string-to-tree transducer *stt* with the initial state q (instead of q_0).

Let $v = \$\flat\alpha_1\flat\alpha_2\flat \cdots \flat\alpha_n\$$. Establishing that $stt(dec(w)) = \{btr_A(w)\}$ merely requires a straightforward case analysis the details of which we leave out due to space limitations. Starting with the case where the backtracking run on w *fails*, the analysis establishes that for rejecting backtracking runs $t = btr_A(q, w, 0^{Q_2})$, we have $t \in stt_{f_q}(v)$, for all q, where v equals $dec(w)$ with the initial \$ removed (we will deal with this at the end) and, vice versa, $t \in stt_{f_q}(v)$ is true for exactly one t, so $t = btr_A(q, w, 0^{Q_2})$.

The proof for the accepting runs follows very similar lines, but with the extra wrinkle of how Q_2 rules are handled when some path accepts. The invariant that $t \in stt_{a_q}(v)$ is true for at most one t is maintained however, as is, of course, the parallel to btr_A. Again, the proof shows that $stt_{a_q}(v)$ outputs precisely one tree if v is $dec(w)$ with the initial \$ removed. That initial \$ is now used by the initial rules in *stt*: $q'_0 \xrightarrow{\$} a_{q_0}$ and $q'_0 \xrightarrow{\$} f_{q_0}$. This means that *stt* produces exactly one tree for every $dec(w)$, and in both the accepting and rejecting case it matches the tree from btr_A.

Finally, we need to deal with *incorrect* decorations. Let v be a decoration of w which is not $dec(w)$. If v has no leading \$, or no trailing \$, or has a \$ in any other position, $stt(v) = \emptyset$, since *stt* has no other possible rules for \$. If v contains *extraneous* $\flat$ we still have $stt(v) = \{btr_A(w)\}$, since they will just be consumed by $q \xrightarrow{\flat} q$ rules. If some $\flat$ is "missing" compared to $dec(w)$ this either causes $stt(v) = \emptyset$, if a Q_2 rule needed it, or $stt(v) = \{btr_A(w)\}$, if it is just removed by a $q \xrightarrow{\flat} q$ rule anyway. □

Theorem 14. *It is decidable in exponential time whether a given pNFA A has exponential backtracking.*

Proof. From A, construct the δ_2-flattened pNFA A' according to Definition 8. According to Lemma 10, A' can be constructed in polynomial time, and it has exponential backtracking if and only if A has. Construct the transducer *stt* for A' according to Definition 11. By Lemma 13 *stt* outputs exponentially large trees if and only if A' has exponential backtracking. The construction of *stt* can clearly be implemented to run in polynomial time. Hence, Lemma 7 yields the result. □

5.2 Hardness of General Backtracking Analysis

It seems likely that general backtracking analysis is computationally difficult. We cannot prove this yet, but here we demonstrate that either it is hard to decide if $J^p(E)$ has exponential backtracking *or* the class of regular expressions E such that $J^p(E)$ does *not* have exponential backtracking has an easy universality decision problem. In the following, we say that E has exponential backtracking if $J^p(E)$ does.

Let us briefly recall the universality problem.

Definition 15. *A regular expression E is* Σ-universal *if $\Sigma^* \subseteq \mathscr{L}(E)$. The input of* RE Universality *is an alphabet Σ and a regular expression E over Σ. The question asked is whether $\mathscr{L}(E)$ is Σ-universal.*

This problem is well-known to be PSPACE-complete. See e.g. [6]. We will now give a simple polynomial reduction which takes a regular expression E and constructs a new regular expression E' such that E' has exponential backtracking if E has exponential backtracking *or* E is not universal.

Lemma 16. *Let E be a regular expression over Σ, $\alpha \in \Sigma$, and $\Gamma = \Sigma \cup \{\$\}$ for some $\$ \notin \Sigma$. If E does not have exponential backtracking then $E' = ((E \,|\, E\$\Gamma^*)\,|(\Sigma^*\$(\alpha^*)^*\$)$ has exponential backtracking if and only if E is not Σ-universal.*

Proof. If E does not have exponential backtracking then neither does $E\$\Gamma^*$, since Γ^* never fails. Now, let $A = J^p(E')$. For every input string, the backtracking run of A will attempt to match $\Sigma^*\$(\alpha^*)^*\$$ to the string only if neither E nor $E\$\Gamma^*$ matches it. If E is universal, i.e. equal to Σ^*, then $\mathscr{L}(E|(E\$\Gamma^*)) = \mathscr{L}(\Sigma^*|(\Sigma^*\$\Gamma^*)) = \Gamma^*$ (since a string in Γ^* is either in Σ^* or has a prefix in Σ^* followed by a suffix in Γ^* that begins with a \$). Hence, in this case E' has exponential backtracking if and only if E does.

If we instead assume that E is *not* universal, then there exists some $w \in \Sigma^*$ such that $w \notin \mathscr{L}(E)$. Consider the string $w\$\alpha^n$ for any $n \in \mathbb{N}$. Neither E nor $E\$\Gamma^*$ matches it, which means that backtracking will proceed into $\Sigma^*\$(\alpha^*)^*\$$, where 2^n backtracking attempts will be made to match the suffix $\alpha^n\$$ to the subexpression $(\alpha^*)^*\$$ (as the final \$ keeps failing to match). □

The previous lemma yields the following corollary.

Corollary 17. *Let $\mathscr{E}$ be the set of all regular expressions that do* not *have exponential backtracking. Then either RE Universality is not PSPACE-hard for inputs in $\mathscr{E}$,* or *deciding whether regular expressions have exponential backtracking is PSPACE-hard.*

5.3 The Complexity of Failure Backtracking Analysis

Now we look at the problem to decide whether a given pNFA has exponential failure backtracking (see Definition 6). For reasons of technical simplicity, assume that parallel ε-transitions are absent from pNFA in this section. To simplify the exposition in this section, and to obtain a useful notion of ambiguity for NFA with ε-cycles, we restrict our notion of accepting runs of an NFA, as originally defined in Section 2. Consider a run $p_1 \cdots p_{m+1}$ on an input string $w = \beta_1 \cdots \beta_m \in \Sigma^*$. This run is called *short* if there are no

i, j, $1 \leq i < j \leq m$, such that $\beta_i = \ldots = \beta_j = \varepsilon$, $p_i = p_j$, and $p_{i+1} = p_{j+1}$. Thus, a short run must not contain any ε-cycle in which an ε-transition appears twice.

First we recall definitions from [1] on ambiguity for NFA, but for NFA with ε-cycles. These definitions differ from those in [1], due to the fact that we allow ε-cycles by using short accepting runs. We define the *degree of ambiguity* of a string w in N, denoted by $da(N,w)$, to be the number of short accepting runs in N labeled by w. N is *polynomially ambiguous* if there exists a polynomial h such that $da(N,w) \leq h(|w|)$ for all $w \in \Sigma^*$. The minimal degree of such a polynomial is the *degree of polynomial ambiguity* of N. We call N *exponentially ambiguous* if $g(n) = \max_{|w| \leq n} da(N,w) \in 2^{\Omega(n)}$ (or equivalently, if $g(n) \in 2^{\Theta(n)}$). It follows from Proposition 1 of [1] that N is either polynomially or exponentially ambiguous, i.e., there is nothing in between. To be precise, this concerns only NFA without ε-cycles, but as the proof of the following theorem shows, it extends to our more general case.

Theorem 18. *For an NFA N it is decidable in time $O(|N|_E^3)$ whether N is polynomially ambiguous, where $|N|_E$ denotes the number of transitions of N. If N is polynomially ambiguous, the degree of polynomial ambiguity can be computed in time $O(|N|_E^3)$.*

Proof. If N is ε-cycle free, the result follows from Theorems 5 and 6 in [1]. Now let $N = (Q, \Sigma, q_0, \delta, F)$ be an NFA, potentially with ε-cycles, and define the equivalence relation $\sim$ on Q, where $p \sim q$ if and only if they are in the same strongly connected component determined by using only ε-transitions in N. Let $N' := N/\sim$ be the quotient of N by $\sim$, having as states the equivalence classes of $\sim$.

The correctness of the remainder of the argument requires N not to have equivalence classes with two elements, say p, q, where both p and q do not have ε self-loops. We briefly argue how equivalences classes of this form can be removed without changing the ambiguity properties of N. It is tedious, but straightforward, to verify that this can for example be achieved by replacing p and q (and $p \xrightarrow{\varepsilon} q$, $q \xrightarrow{\varepsilon} p$) with 6 states and the appropriately defined ε-transitions to model the behavior of short runs in N that go through one or both consecutively of p and q. Three of the 6 states are used to model incoming transitions to p in (short) runs that after reaching p do not follow $p \xrightarrow{\varepsilon} q$, or follow only $p \xrightarrow{\varepsilon} q$, or follow consecutively $p \xrightarrow{\varepsilon} q$ and $q \xrightarrow{\varepsilon} p$, and the other 3 states are used for q in a similar way.

N' could potentially have (parallel) ε self-loops. Let N'' be N' with ε self-loops removed. Each state in N'' will belong to exactly one of the following categories of equivalence classes: (a) a single state of N without an ε self-loop in N; (b) a single state of N with an ε self-loop in N; (c) at least two states such that, in N, there are at least two distinct ε-runs (staying within the equivalence class) between any two states in the equivalence class (thanks to the modification of N described in the preceding paragraph).

Let Z be the set of states in N'' having the properties specified in (b) or (c). In N'' there are two possibilities. Either (i) there is a (short run which is a) cycle in N'' having at least one state in Z, or (ii) each short run in N'' goes through at most k states in Z (k is bounded by the number of states in N''). In case (i), N is exponentially ambiguous, since we have at least two ε-runs in N between any two states in an equivalence class in Z. In case (ii), the number of accepting runs in N'' (by definition without ε-cycles) and number of short accepting runs in N, differ by a constant factor, and we can apply the ε-cycle free result from [1] to N''. □

Theorem 19. *A pNFA A has either polynomial or exponential failure backtracking. It can be decided in time $O(|A|_E^3)$ whether A has polynomial failure backtracking, and if so, the degree of backtracking can be computed in time $O(|A|_E^3)$.*

Proof. Recall that A^f denotes the pNFA obtained from A where we change all states of A so that they are not accepting, and $\overline{A}^f$ denotes the NFA obtained by ignoring priorities on transitions of A^f. For an NFA

N, $a(N)$ is obtained from N by adding a new accepting sink state z (having transitions to itself on all input letters), all other states in N are made non-accepting, and we add ε-transitions from all states in N to z. Since $da(a(\overline{A}^f), w) = |btr_{A^f}(w)|$, and thus $\max\{da(a(\overline{A}^f), w) \mid w \in \Sigma^*, |w| \leq n\} = \max\{|btr_{A^f}(w)| \mid w \in \Sigma^*, |w| \leq n\}$, the failure backtracking complexity of A is equal to the ambiguity of $a(\overline{A}^f)$. To complete the proof, apply Theorem 18 to $a(\overline{A}^f)$. □

6 Conclusion/Future Work

Our prioritized NFA model is the only automata model, that we are aware of, which formalizes backtracking regular expression matching. This model is well suited to be extended to describe notions such as possessive quantifiers, captures and backreferences found in practical regular expressions. Backreferences have been formalized in [3], but without eliminating ambiguities due to multiple matches. Trying to improve our current complexity result for deciding backtracking complexity (as in Definition 6), and secondly, to formalize what is meant by equivalence of a regular expression with a pNFA, will provide the impetus for future investigations.

Acknowledgment We thank the referees for extensive lists of valuable comments.

References

[1] Cyril Allauzen, Mehryar Mohri & Ashish Rastogi (2008): *General Algorithms for Testing the Ambiguity of Finite Automata.* In Masami Ito & Masafumi Toyama, editors: *Developments in Language Theory*, *Lecture Notes in Computer Science* 5257, Springer, pp. 108–120, doi:10.1007/978-3-540-85780-8_8.

[2] Robert S. Boyer & J. Strother Moore (1977): *A fast string searching algorithm.* *Communications of the ACM* 20(10), pp. 762–772, doi:10.1145/359842.359859.

[3] Cezar Câmpeanu, Kai Salomaa & Sheng Yu (2003): *A Formal Study of Practical Regular Expressions.* *International Journal of Foundations of Computer Science* 14(6), pp. 1007–1018, doi:10.1142/S012905410300214X.

[4] Frank Drewes (2001): *The Complexity of the Exponential Output Size Problem for Top-Down and Bottom-Up Tree Transducers,.* *Information and Computation* 169(2), pp. 264 – 283, doi:10.1006/inco.2001.3039.

[5] Bryan Ford (2004): *Parsing Expression Grammars: A Recognition-Based Syntactic Foundation.* In Neil D. Jones & Xavier Leroy, editors: *Symposium on Principles of Programming Languages (POPL'04)*, ACM Press, pp. 111–122, doi:10.1145/964001.964011.

[6] Michael R. Garey & David S. Johnson (1979): *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA.

[7] James Kirrage, Asiri Rathnayake & Hayo Thielecke (2013): *Static Analysis for Regular Expression Denial-of-Service Attacks.* In: *Network and System Security*, Springer, pp. 135–148, doi:10.1007/978-3-642-38631-2_11.

[8] Sérgio Medeiros, Fabio Mascarenhas & Roberto Ierusalimschy (2012): *From Regexes to Parsing Expression Grammars.* *CoRR* abs/1210.4992, doi:10.1016/j.scico.2012.11.006.

[9] Ken Thompson (1968): *Regular Expression Search Algorithm.* *Communications of the ACM* 11(6), pp. 419–422, doi:10.1145/363347.363387.

Measuring Communication in Parallel Communicating Finite Automata

Henning Bordihn
Institut für Informatik, Universität Potsdam,
August-Bebel-Str. 89, 14482 Potsdam, Germany
henning@cs.uni-potsdam.de

Martin Kutrib and Andreas Malcher
Institut für Informatik, Universität Giessen,
Arndtstr. 2, 35392 Giessen, Germany
{kutrib,malcher}@informatik.uni-giessen.de

Systems of deterministic finite automata communicating by sending their states upon request are investigated, when the amount of communication is restricted. The computational power and decidability properties are studied for the case of returning centralized systems, when the number of necessary communications during the computations of the system is bounded by a function depending on the length of the input. It is proved that an infinite hierarchy of language families exists, depending on the number of messages sent during their most economical recognitions. Moreover, several properties are shown to be not semi-decidable for the systems under consideration.

1 Introduction

Communication is one of the most fundamental concepts in computer science: objects of object-oriented programs, roles or pools in business processes, concurrent processes in computer networks or in information or operating systems are examples of communicating agents.

Parallel communicating finite automata systems (PCFA) have been introduced in [12] as a simple automaton model of parallel processes and cooperating systems, see also [1, 2, 4]. A PCFA consists of several finite automata, the components of the system, that process a joint input string independently of each other. However, their transitions are synchronized according to a global clock. The cooperation of the components is enabled by communication steps in which components can request the state reached by another component. The system can work in returning or non-returning mode. In the former case each automaton which sends its current state is set back to its initial state after this communication step. In the latter case the state of the sending automaton is not changed. Recently, these communication protocols have been refined in [15] and further investigated for the case of parallel communicating systems of pushdown automata [14]. There, the communication process is performed in an asynchronous manner, reflecting the technical features of many real communication processes. In the sequel of this paper and as a first step towards an investigation of the influence of restricted communication to parallel communicating systems of automata, we stick with the simpler model having synchronized communication steps.

In a PCFA, one also distinguishes between centralized systems where only one designated automaton, called master, can request information from other automata, and non-centralized systems where every automaton is allowed to request information from others. Taking the distinction between returning and non-returning systems into account, we are led to four different working modes. Moreover, one distinguishes between deterministic and nondeterministic PCFA. The system is deterministic, if all its components are deterministic finite automata.

It is known from [2, 4, 12] that deterministic (nondeterministic) non-centralized PCFA are equally powerful as deterministic (nondeterministic) one-way multi-head finite automata [6], both in returning and non-returning working modes. Moreover, it is proved in [2] that nondeterminism is strictly more powerful than determinism for all the four working modes, and that deterministic centralized returning systems are not weaker than deterministic centralized non-returning ones.

All variants of PCFA accept non-regular languages due to the feature that communication between the components of the system is allowed. Thus it is of interest to measure the amount of communication needed for accepting those languages. Mitrana proposed in [13] a dynamical measure of descriptional complexity as follows: The degree of communication of a PCFA for a given word is the minimal number of communications necessary to recognize the word. Then, the degree of communication of a PCFA is the supremum of the degrees of communication taken over all words recognized by the system, while the degree of communication of a language (with respect to a PCFA of type X) is the infimum of the degrees of communication taken over all PCFA of type X that accept the language. Mitrana proved that this measure cannot be algorithmically computed for languages accepted by nondeterministic centralized or non-centralized non-returning PCFA. The computability status of the degree of communication for the other types of PCFA languages as well as for all types of PCFA is stated as open question in [13].

In this paper, we study PCFA where the degree of communication is bounded by a function in the length of the input word. We restrict ourselves to one of the simplest types of PCFA, namely to deterministic centralized returning systems of finite automata. In the next section, the basic definitions and two examples of languages accepted by communication bounded PCFA are presented. In Section 3, we show that bounding the degree of communication by logarithmic, square root or linear functions leads to three different families of languages. For the strictness results, we use similar witness languages and a proof technique based on Kolmogorov complexity as in [9], where the second and the third author investigated the computational power of two-party Watson-Crick systems, that is, synchronous systems consisting of two finite automata running in opposite directions on a shared read-only input and communicating by broadcasting messages.

In Section 4, non-semi-decidability results are proved for deterministic returning centralized PCFA and their languages, thus partially answering questions listed as open in [13]. Similarly to [1] the proofs rely on properties of one-way cellular automata and their valid computations. Finally, Section 5 refines the three-level hierarchy from Section 3 to an infinite hierarchy.

2 Preliminaries and Definitions

We write Σ^* for the set of all words over the finite alphabet Σ, and $\mathbb{N}$ for the set $\{0,1,2,\dots\}$ of non-negative integers. The *empty word* is denoted by λ. For the *length* of w we write $|w|$. We use $\subseteq$ for *inclusions* and $\subset$ for *strict inclusions*.

Next we turn to the definition of parallel communicating finite automata systems. The nondeterministic model has been introduced in [12]. Following [1], the formal definition is as follows.

A *deterministic parallel communicating finite automata system of degree k* (DPCFA(k)) is a construct $A = \langle \Sigma, A_1, A_2, \dots, A_k, Q, \triangleleft \rangle$, where

1. Σ is the set of *input symbols*,
2. each $A_i = \langle S_i, \Sigma, \delta_i, s_{0,i}, F_i \rangle$, $1 \leq i \leq k$, is a *deterministic finite automaton* with finite state set S_i, *partial* transition function $\delta_i : S_i \times (\Sigma \cup \{\lambda, \triangleleft\}) \to S_i$ (requiring that $\delta_i(s,a)$ is undefined for all

$a \in \Sigma \cup \{\lhd\}$, if $\delta_i(s,\lambda)$ is defined), initial state $s_{0,i} \in S_i$, and set of accepting states $F_i \subseteq S_i$,

3. $Q = \{q_1, q_2, \ldots, q_k\} \subseteq \bigcup_{1 \leq i \leq k} S_i$ is the set of *query states*, and
4. $\lhd \notin \Sigma$ is the *end-of-input symbol*.

The single automata are called *components* of the system A. A *configuration* $(s_1, x_1, s_2, x_2, \ldots, s_k, x_k)$ of A represents the current states s_i as well as the still unread parts x_i of the tape inscription of all components $1 \leq i \leq k$. System A starts with all of its components scanning the first square of the tape in their initial states. For input word $w \in \Sigma^*$, the initial configuration is $(s_{0,1}, w\lhd, s_{0,2}, w\lhd, \ldots, s_{0,k}, w\lhd)$.

Basically, a computation of A is a sequence of configurations beginning with an initial configuration and ending with a halting configuration, when no successor configuration exists. Each step can consist of two phases. In a first phase, all components are in non-query states and perform an ordinary (non-communicating) step independently. The second phase is the communication phase during which components in query states receive the requested states as long as the sender is not in a query state itself. That is, if a component A_i is in query state q_j, then A_i is set to the current state of component A_j. This process is repeated until all requests are resolved, if possible. If the requests are cyclic, no successor configuration exists. For the first phase, we define the successor configuration relation $\vdash$ by $(s_1, a_1y_1, s_2, a_2y_2, \ldots, s_k, a_ky_k) \vdash (p_1, z_1, p_2, z_2, \ldots, p_k, z_k)$, if $Q \cap \{s_1, s_2, \ldots, s_k\} = \emptyset$, $a_i \in \Sigma \cup \{\lambda, \lhd\}$, $p_i \in \delta_i(s_i, a_i)$, and $z_i = \lhd$ for $a_i = \lhd$ and $z_i = y_i$ otherwise, $1 \leq i \leq k$. For non-returning communication in the second phase, we set $(s_1, x_1, s_2, x_2, \ldots, s_k, x_k) \vdash (p_1, x_1, p_2, x_2, \ldots, p_k, x_k)$, if, for all $1 \leq i \leq k$ such that $s_i = q_j$ and $s_j \notin Q$, we have $p_i = s_j$, and $p_r = s_r$ for all the other r, $1 \leq r \leq k$. Alternatively, for returning communication in the second phase, we set $(s_1, x_1, s_2, x_2, \ldots, s_k, x_k) \vdash (p_1, x_1, p_2, x_2, \ldots, p_k, x_k)$, if, for all $1 \leq i \leq k$ such that $s_i = q_j$ and $s_j \notin Q$, we have $p_i = s_j$, $p_j = s_{0,j}$, and $p_r = s_r$ for all the other r, $1 \leq r \leq k$.

A computation *halts* when the successor configuration is not defined for the current situation. In particular, this may happen when cyclic communication requests appear, or when the transition function of one component is not defined. The language $L(A)$ accepted by a DPCFA(k) A is precisely the set of words w such that there is some computation beginning with $w\lhd$ on the input tape and halting with at least one component having an undefined transition function and being in an accepting state. Let $\vdash^*$ denote the reflexive and transitive closure of the successor configuration relation $\vdash$ and define $L(A)$ as

$$\{\, w \in \Sigma^* \mid (s_{0,1}, w\lhd, s_{0,2}, w\lhd, \ldots, s_{0,k}, w\lhd) \vdash^* (p_1, a_1y_1, p_2, a_2y_2, \ldots, p_k, a_ky_k),$$
$$\text{such that } p_i \in F_i \text{ and } \delta_i(p_i, a_i) \text{ as well as } \delta_i(p_i, \lambda) \text{ are undefined for some } 1 \leq i \leq k \,\}.$$

Whenever the degree is missing in the notation DPCFA(k), we mean systems of arbitrary degree. The absence or presence of an R in the type of the system denotes whether it works in *non-returning* communication, that is, the sender remains in its current state, or *returning* communication, that is, the sender is reset to its initial state. If there is just one component, say A_1, that is allowed to query for states, that is, $S_i \cap Q = \emptyset$, for $2 \leq i \leq k$, then the system is said to be *centralized*. In this case, we refer to A_1 as the *master component* and add a C to the notation of the type of the system. The *family of languages accepted* by devices of type X with arbitrary degree (with degree k) is denoted by $\mathscr{L}(X)$ ($\mathscr{L}(X(k))$).

In the following, we study the impact of communication in PCFA. The communication is measured by the *total number of queries sent during a computation*. That is, we count the number of time steps at which a component enters a query state and consider the sum of these numbers for all components. Let $f : \mathbb{N} \to \mathbb{N}$ be a mapping. If all $w \in L(A)$ are accepted with computations where the total number of queries sent is bounded by $f(|w|)$, then A is said to be *communication bounded by f*.

We denote the class of devices of type X (with degree k) that are communication bounded by some function f by f-X (f-X(k)).

In order to clarify the notation we give two examples. Whenever we refer to a time t of a computation of a DPCFA, then the configuration reached after exactly t computation steps is considered.

Example 1 The language $L_{expo} = \{\,\$a^{2^0}ba^{2^1}b\cdots ba^{2^m}\& \mid m \geq 1\,\}$ belongs to $\mathscr{L}(f\text{-DRCPCFA}(2))$ with $f \in O(\log(n))$. Roughly, the idea of the construction is that the lengths of adjacent a-blocks (separated by a b) are compared. To this end, the master reads the left block with half speed, that is, moving one symbol to the right in every other time step, while the non-master component reads the right block with full speed, that is, moving one symbol to the right in every time step. If the master reaches a b, it queries the non-master whether it has also reached a b. If this is true, the comparison of the next two a-blocks is started. The input is accepted if the master obtains the symbol & from the non-master component and the remaining input is in $a^+\&\lhd$.

Formally, we define $A = \langle\{a,b,\$,\&\},A_1,A_2,\{q_2\},\lhd\rangle$ to be a DRCPCFA(2) with master component $A_1 = \langle\{s_{0,1},s_{1,1},s_{2,1},s_{3,1},s_{4,1},s_{5,1},s_b,s_\&,q_2,accept\},\{a,b,\$,\&\},\delta_1,s_{0,1},\{accept\}\rangle$, second component $A_2 = \langle\{s_{0,2},s_{1,2},s_{2,2},s_{3,2},s_b,s_\&,s_\lhd\},\{a,b,\$,\&\},\delta_2,s_{0,2},\emptyset\rangle$, and transition functions δ_1 and δ_2 as follows.

The non-master component A_2:

1. $\delta_2(s_{0,2},\$) = s_{1,2}$	4. $\delta_2(s_{3,2},a) = s_{3,2}$	7. $\delta_2(s_{0,2},a) = s_{3,2}$
2. $\delta_2(s_{1,2},a) = s_{2,2}$	5. $\delta_2(s_{3,2},b) = s_b$	8. $\delta_2(s_{0,2},\lhd) = s_\lhd$
3. $\delta_2(s_{2,2},b) = s_{3,2}$	6. $\delta_2(s_{3,2},\&) = s_\&$	9. $\delta_2(s_\lhd,\lambda) = s_\lhd$

The component reads the input prefix $\$ab$ in the first three time steps (rules 1,2,3). Subsequently, it reads an a-block in state $s_{3,2}$ (rule 4). Whenever it moves on a symbol b it changes into state s_b (rule 5). So, it enters state s_b at time step 3 plus the length of the second a-block plus 1. The component halts in state s_b unless it is reset to its initial state by a query. In this case it reads the current a-block and the next b and enters state s_b again after a number of time steps that is the length of the a-block plus one (rules 7,4,5). Rule 6 is used when & appears in the input instead of b. After being reset into the initial state on the endmarker, the component enters state $s_\lhd$ and loops with λ-moves.

The master component A_1:

1. $\delta_1(s_{0,1},\$) = s_{1,1}$	5. $\delta_1(s_{4,1},\lambda) = s_{3,1}$	9. $\delta_1(s_\&,\&) = s_{5,1}$
2. $\delta_1(s_{1,1},\lambda) = s_{2,1}$	6. $\delta_1(s_{3,1},b) = q_2$	10. $\delta_1(s_{5,1},\lhd) = accept$
3. $\delta_1(s_{2,1},\lambda) = s_{3,1}$	7. $\delta_1(s_b,a) = s_{4,1}$	
4. $\delta_1(s_{3,1},a) = s_{4,1}$	8. $\delta_1(s_\&,a) = s_\&$	

The master reads the input prefix $\$ab$ in the first six time steps and enters the query state q_2 (rules 1–6). Exactly at that time the non-master component enters state s_b. Being in state s_b received the master reads the current a-block and the next b and enters state q_2 again after a number of time steps that is two times the length of the a-block plus one (rules 7,4,5,6). Exactly at this time the non-master component enters state s_b again provided that the a-block read by the non-master component is twice as long as the a-block read by the master. When the master receives state $s_\&$ instead of s_b, it reads the remaining suffix (rules 8,9), enters the accepting state on the endmarker (rule 10) and halts.

Finally, the length of a word $w \in L_{expo}$ is $|w| = m+2+\sum_{i=0}^{m} 2^i = 2^{m+1}+m+1$, for some $m \geq 1$. In its accepting computation, a communication takes place for every symbol b and the endmarker. So there are $m+1$ communications which is of order $O(\log(|w|))$. $\square$

The construction of the next example is similar to the one given in Example 1.

Example 2 The language $L_{poly} = \{\, \$aba^3ba^5b\cdots ba^{2m+1}\& \mid m \geq 0 \,\}$ belongs to $\mathscr{L}(f\text{-DRCPCFA}(2))$ with $f \in O(\sqrt{n})$. $\square$

3 Computational Capacity

In this section we consider aspects of the computational capacity of f-DRCPCFA(k). Examples 1 and 2 already revealed that there are non-semilinear languages accepted by systems with two components and sublinear communication. The next simple result is nevertheless important for the size of representations that will be used in connection with Kolmogorov arguments to separate language classes.

Lemma 3 *Let $k \geq 1$ and A be a DRCPCFA(k) with $S_1, S_2, \ldots, S_k$ being the state sets of the single components. If $w \in L(A)$, then w is accepted after at most $|S_1|\cdot|S_2|\cdots|S_k|\cdot(|w|+1)$ time steps, that is, in linear time.*

Proof During a computation some component A_i may be in $|S_i|$ different states. So after $|S_1|\cdot|S_2|\cdots|S_k|$ time steps the whole system runs through a loop if none of the components moves. Therefore, as long as no halting configuration is reached, at least one component must move after at most $|S_1|\cdot|S_2|\cdots|S_k|$ time steps. $\square$

The language of the next lemma combines the well-known non-context-free copy language with L_{expo} from above. It plays a crucial role in later proofs.

Lemma 4 *The language*

$$L_{expo,wbw} = \{\, \$w_1w_2\cdots w_m ba^{2^0}w_1w_1a^{2^1}w_2w_2\cdots a^{2^{m-1}}w_mw_m\& \mid m \geq 1, w_i \in \{0,1\}, 1 \leq i \leq m \}$$

belongs to $\mathscr{L}(O(\log(n))$-DRCPCFA$(3))$.

Proof A formal construction of a $O(\log(n))$-DRCPCFA(3) accepting $L_{expo,wbw}$ is given through the transition functions below, where $s_{0,i}$ is the initial state of component A_i, $1 \leq i \leq 3$, the sole accepting state is *accept*, and $\sigma \in \{0,1\}$.

The second non-master component A_3 initially passes over the \$ and, then, it reads a symbol, remembers it in its state, and loops without moving (rules 1,2,3,8,9). Whenever the component is reset into its initial state after a query, it reads the next symbol, remembers it, and loops without moving (rules 4–11). This component is used by the master to match the w_i from the prefix with the w_i from the suffix.

The non-master component A_3:

1. $\delta_3(s_{0,3},\$) = s_{1,3}$	5. $\delta_3(s_{0,3},1) = s_1$	9. $\delta_3(s_1,\lambda) = s_1$
2. $\delta_3(s_{1,3},0) = s_0$	6. $\delta_3(s_{0,3},b) = s_b$	10. $\delta_3(s_b,\lambda) = s_b$
3. $\delta_3(s_{1,3},1) = s_1$	7. $\delta_3(s_{0,3},a) = s_a$	11. $\delta_3(s_a,\lambda) = s_a$
4. $\delta_3(s_{0,3},0) = s_0$	8. $\delta_3(s_0,\lambda) = s_0$	

The first non-master component A_2 initially passes over the prefix $\$w_1w_2\cdots w_m$ (rules 1,2), the b (rule 3), and the adjacent infix $aw_1w_1aaw_2w_2$ (rules 4–13). On its way it checks whether the neighboring symbols w_i are in fact the same (rules 5–8 and 10–13). If the second check is successful the component enters state s_{ww}. Exactly at that time it has to be queried by the master, otherwise it blocks the computation. Subsequently, it repeatedly continues to read the input, where each occurrence of neighboring symbols w_i are checked for equality (rules 14 and 9–13), which is indicated by entering state s_{ww} again. This component is used to verify that all neighboring symbols w_i in the suffix are equal and, by the master, to check the lengths of the a-blocks in the same way as in Example 1. Note that the component is at time $m+9$ on the first symbol after w_2w_2. After being reset to its initial state, it takes a number of time steps equal to the length of the next a-block plus 2 to get on the first symbol after the next w_iw_i.

The non-master component A_2:

1. $\delta_2(s_{0,2},\$) = s_{1,2}$
2. $\delta_2(s_{1,2},\sigma) = s_{1,2}$
3. $\delta_2(s_{1,2},b) = s_{2,2}$
4. $\delta_2(s_{2,2},a) = s_{3,2}$
5. $\delta_2(s_{3,2},0) = s^0_{4,2}$
6. $\delta_2(s_{3,2},1) = s^1_{4,2}$
7. $\delta_2(s^0_{4,2},0) = s_{5,2}$
8. $\delta_2(s^1_{4,2},1) = s_{5,2}$
9. $\delta_2(s_{5,2},a) = s_{5,2}$
10. $\delta_2(s_{5,2},0) = s^0_{6,2}$
11. $\delta_2(s_{5,2},1) = s^1_{6,2}$
12. $\delta_2(s^0_{6,2},0) = s_{ww}$
13. $\delta_2(s^1_{6,2},1) = s_{ww}$
14. $\delta_2(s_{0,2},a) = s_{5,2}$
15. $\delta_2(s_{0,2},\&) = s_{\&}$
16. $\delta_2(s_{\&},\lambda) = s_{\&}$
17. $\delta_2(s_{0,2},\lhd) = s_{\lhd}$
18. $\delta_2(s_{\lhd},\lambda) = s_{\lhd}$

The master component A_1 initially passes over the prefix $\$w_1w_2\cdots w_m$ (rules 1,2), the b (rule 3), and the first a (rules 4–8). Then it reads the first of two adjacent symbols w_i and enters the query state q_3 (rule 9) (the equality of the symbols w_i has already been checked by component A_2). From component A_3 it receives the information about the matching symbol w_i from the prefix. If this symbol is the same as the next input symbol, then the computation continues (rules 10,11) by entering query state q_2. Note that this happens exactly at time step $m+9$. If the master receives state s_{ww} the length of the first two a-blocks are verified. Now the master repeatedly continues to read the input (rule 12,7,8), where on each occurrence of neighboring symbols w_i the equality with the corresponding symbol in the prefix is checked by querying component A_3 and the lengths of the a-blocks are compared by querying component A_2. After querying component A_2, it takes a number of time steps equal to the length of the adjacent a-block (processed by component A_2) plus 2 to get into state q_2 again. Finally, when the master component has checked the last symbol w_m and gets the information that A_2 has read symbol &, it queries component A_3 (rule 13). If it receives a b, the input is accepted (rule 14). In all other cases it is rejected.

The master component A_1:

1. $\delta_1(s_{0,1},\$) = s_{1,1}$
2. $\delta_1(s_{1,1},\sigma) = s_{1,1}$
3. $\delta_1(s_{1,1},b) = s_{2,1}$
4. $\delta_1(s_{2,1},\lambda) = s_{3,1}$
5. $\delta_1(s_{3,1},\lambda) = s_{4,1}$
6. $\delta_1(s_{4,1},\lambda) = s_{5,1}$
7. $\delta_1(s_{5,1},a) = s_{6,1}$
8. $\delta_1(s_{6,1},\lambda) = s_{5,1}$
9. $\delta_1(s_{5,1},\sigma) = q_3$
10. $\delta_1(s_0,0) = q_2$
11. $\delta_1(s_1,1) = q_2$
12. $\delta_1(s_{ww},a) = s_{6,1}$
13. $\delta_1(s_{\&},\&) = q_3$
14. $\delta_1(s_b,\lhd) = accept$

The length of a word $w \in L_{expo,wbw}$ is $|w| = 3m+3+\sum_{i=0}^{m-1} 2^i = 2^m+3m+2$, for some $m \geq 1$. In its accepting computation, two communications take place for every w_iw_i and one more communication on the endmarker. So there are $2m+1$ communications which is of order $O(\log(|w|))$. $\square$

For the proof of the following theorem we use an incompressibility argument. General information on Kolmogorov complexity and the incompressibility method can be found in [10]. Let $w \in \{0,1\}^+$ be an arbitrary binary string. The Kolmogorov complexity $C(w)$ of w is defined to be the minimal size of a program describing w. The following key argument for the incompressibility method is well known. There are binary strings w of *any* length so that $|w| \leq C(w)$.

Lemma 5 *The language $L_{wbw} = \{ w_1w_2 \cdots w_mbw_1w_2 \cdots w_m \mid m \geq 1, w_i \in \{0,1\}, 1 \leq i \leq m \}$ is accepted by some $O(n)$-DRCPCFA(2) but, for any $k \geq 1$, does not belong to $\mathscr{L}(f$-DRCPCFA$(k))$ if $f \in \frac{n}{\omega(\log(n))}$.*

Proof First, we sketch the construction of a $O(n)$-DRCPCFA(2) accepting L_{wbw}. Initially, the master component proceeds to the center marker b, while the non-master component reads the first input symbol w_1 and remembers this information in its state. Next, the master queries the non-master and matches the information received with the first symbol following b, while the non-master reads the next input symbol and remembers it in its state. Subsequently, this behavior is iterated, that is, the master queries the non-master again and matches its next input symbol, while the non-master reads and remembers the next symbol. The input is accepted when the master receives a b at the moment it reaches the right endmarker. Clearly, the number of communications on input length $n = 2m+1$ is $m+1 \in O(n)$.

Second, we turn to show that $L_{wbw} \notin \mathscr{L}(f\text{-DRCPCFA}(k))$ if $f \in \frac{n}{\omega(\log(n))}$. In contrast to the assertion, we assume that L_{wbw} is accepted by some f-DRCPCFA(k) $A = \langle \Sigma, A_1, A_2, \ldots, A_k, Q, \triangleleft \rangle$ with $f(n) \in \frac{n}{\omega(\log(n))}$. Let $z = wbw$, for some $w \in \{0,1\}^+$, and $K_0 \vdash \cdots \vdash K_{acc}$ be the accepting computation on input z, where K_0 is the initial configuration and K_{acc} is an accepting configuration.

Next, we consider snapshots of configurations at every time step at which the master component queries some other component or at which a component enters the middle marker b. For every such configuration, we take the time step t_i, the current states $s_1^{(i)}, s_2^{(i)}, \ldots, s_k^{(i)}$, and the positions $p_1^{(i)}, p_2^{(i)}, \ldots, p_k^{(i)}$ of the components. Thus, the ith snapshot is represented by the tuple $(t_i, s_1^{(i)}, p_1^{(i)}, s_2^{(i)}, p_2^{(i)}, \ldots, s_k^{(i)}, p_k^{(i)})$. Since there are altogether at most $f(2|w|+1)$ communications, the list of snapshots Λ contains at most $f(2|w|+1)+k$ entries.

We claim that each snapshot can be represented by at most $O(\log(|w|))$ bits. Due to Lemma 3 acceptance is in linear time and, therefore, each time step can be represented by at most $O(\log(|w|))$ bits. Each position of a component can also be represented by at most $O(\log(|w|))$ bits. Finally, each state can be represented by a constant number of bits. Altogether, each snapshot can be represented by $O(\log(|w|))$ bits. So, the list Λ can be represented by $(f(2|w|+1)+k) \cdot O(\log(|w|)) = \frac{|w|}{\omega(\log(|w|))} \cdot O(\log(|w|)) = o(|w|)$ bits.

Now we show that the list Λ of snapshots together with a snapshot of K_{acc} and the knowledge of A and $|w|$ is sufficient to reconstruct w. The reconstruction is implemented by the following algorithm P. First, P sequentially simulates A on all $2^{|w|}$ inputs xbx where $|x| = |w|$. Additionally, it is checked whether the computation simulated has the same snapshots as in the list Λ and the accepting configuration. In this way, the string w can be identified. We have to show that there is no other string $w' \neq w$ which can be identified in this way as well. Let us assume that such a w' exists. Then all snapshots of accepting computations on input wbw and $w'bw'$ are identical. This means that both computations end at the same time step and all components are in the same state and position. Additionally, in both computations communications take place at the same time steps, all components are in the same state and position at that moment. Moreover, the right half of the respective words is entered in the same states and in the same time steps on both input words wbw and wbw'. So, both computations are also accepting on input wbw' which is a contradiction.

Thus, w can be reconstructed given the above program P, the list of snapshots Λ, the snapshot of the accepting configuration, A, and $|w|$. Since the sizes of P and A are bounded by a constant, the size of Λ is bounded by $o(|w|)$, and $|w|$ as well as the size of the remaining snapshot is bounded by $O(\log(|w|))$ each, we can reconstruct w from a description of total size $o(|w|)$. Hence, the Kolmogorov complexity $C(w)$, that is, the minimal size of a program describing w is bounded by the size of the above description, and we obtain $C(w) \in o(|w|)$. On the other hand, we know that there are binary strings w of arbitrary length such that $C(w) \geq |w|$. This is a contradiction for w being long enough. □

The language of the next lemma is used in later proofs.

Lemma 6 *The language*

$$L_{poly,wbw} = \{\, \$w_1w_2\cdots w_m ba^1 w_1w_1a^3w_2w_2a^5w_3w_3\cdots a^{2m-1}w_mw_m\& \mid m \geq 1, w_i \in \{0,1\}, 1 \leq i \leq m \}$$

is accepted by some $O(\sqrt{n})$-DRCPCFA(3) but, for any $k \geq 1$, does not belong to $\mathscr{L}(f$-DRCPCFA$(k))$ if $f \in O(\log(n))$.

Proof Using the construction idea of Lemma 4, one shows $L_{poly,wbw} \in \mathscr{L}(O(\sqrt{n})$-DRCPCFA$(3))$.

The claimed non-containment is shown similarly to Lemma 5: in contrast to the assertion, we assume that $L_{poly,wbw}$ is accepted by some f-DRCPCFA(k) $A = \langle \Sigma, A_1, A_2, \ldots, A_k, Q, \triangleleft \rangle$ with $f(n) \in O(\log(n))$. Let

$$z = \$w_1w_2\cdots w_m ba^1 w_1w_1a^3w_2w_2a^5w_3w_3\cdots a^{2m-1}w_mw_m\& \in L_{poly,wbw},$$

where $w = w_1w_2\cdots w_m$, and $K_0 \vdash \cdots \vdash K_{acc}$ be the accepting computation on input z, where K_0 is the initial configuration and K_{acc} is an accepting configuration.

We use again an incompressibility argument and write down the list of snapshots of configurations in which communication takes place and the accepting configuration K_{acc}, and descriptions of A and $|w|$. Similar to the proof of Lemma 5, a program P can be described which reconstructs w uniquely from the information given.

Next, we determine the size of such a description. Program P and the system A can be represented by a constant number of bits. The length $|w|$ can be described by $\log(|w|) \in O(\log(m))$ bits. Since $|z| = 3m+3+\sum_{i=1}^{m} 2i-1 = 3m+3+m^2$ and acceptance is in linear time (Lemma 3), each time step can be represented by $O(\log(|z|)) = O(\log(m^2))$ bits. Moreover, the k states can be described by $O(1)$ bits, and the k positions by $k \cdot \log(|z|) = k \cdot \log(m^2+3m+3) \in O(\log(m))$ bits. So, altogether one snapshot can be represented by $O(\log(m))$ bits. Since at most $f(|z|) \in O(\log(|z|)) = O(\log(m))$ snapshots have to be listed, the list of all snapshots can be described by $O((\log(m))^2)$ bits. Therefore, the total size of a description of w is bounded by $O((\log(m))^2)$ as well. Thus, the Kolmogorov complexity $C(w)$ of w is bounded by $O((\log(m))^2)$. On the other hand, there are binary strings w of arbitrary length such that $C(w) \geq |w| = m$. This is a contradiction for w being long enough. □

The previous theorems showed that there are proper inclusions

$$\mathscr{L}(O(\log(n))\text{-DRCPCFA}(k)) \subset \mathscr{L}(O(\sqrt{n})\text{-DRCPCFA}(k))$$

for every $k \geq 3$, and

$$\mathscr{L}(O(\sqrt{n})\text{-DRCPCFA}(k)) \subset \mathscr{L}(O(n)\text{-DRCPCFA}(k))$$

for every $k \geq 2$.

Later, we will prove an infinite hierarchy in between the classes $\mathscr{L}(O(\log(n))$-DRCPCFA$(k))$ and $\mathscr{L}(O(\sqrt{n}))$-DRCPCFA(k), for every $k \geq 4$.

4 Decidability and Undecidability Results

4.1 Undecidability of Emptiness and Classical Questions

First, we show undecidability of the classical questions for models with a logarithmic amount of communication. To this end, we adapt the construction given in [1] which is based on the valid computations of *one-way cellular automata* (OCA), a parallel computational model (see, for example, [7, 8]). More precisely, the undecidability is shown by reduction of the corresponding problems for OCA which are known not even to be semi-decidable [11]. To this end, histories of OCA computations are encoded in single words that are called *valid computations* (cf., for example, [5]).

A one-way cellular automaton is a linear array of identical deterministic finite automata, sometimes called cells. Except for the leftmost cell each one is connected to its nearest neighbor to the left. The state transition depends on the current state of a cell itself and the current state of its neighbor, where the leftmost cell receives information associated with a boundary symbol on its free input line. The state changes take place simultaneously at discrete time steps. The input mode for cellular automata is called parallel. One can suppose that all cells fetch their input symbol during a pre-initial step.

More formally, an OCA is a system $M = \langle S, \#, T, \delta, F \rangle$, where S is the nonempty, finite set of cell states, $\# \notin S$ is the boundary state, $T \subseteq S$ is the input alphabet, $F \subseteq S$ is the set of accepting cell states, and $\delta : (S \cup \{\#\}) \times S \to S$ is the local transition function.

A configuration of an OCA at some time step $t \geq 0$ is a description of its global state, which is formally a mapping $c_t : \{1, 2, \dots, n\} \to S$, for $n \geq 1$. The initial configuration at time 0 on input $w = x_1 x_2 \dots x_n$ is defined by $c_{0,w}(i) = x_i$, $1 \leq i \leq n$. Let c_t, $t \geq 0$, be a configuration with $n \geq 2$, then its successor c_{t+1} is defined as follows: $c_{t+1}(1) = \delta(\#, c_t(1))$ and $c_{t+1}(i) = \delta(c_t(i-1), c_t(i))$, $2 \leq i \leq n$.

An input is accepted if at some time step during its computation the rightmost cell enters an accepting state. Without loss of generality and for technical reasons, one can assume that any accepting computation has at least three steps.

Now we turn to the valid computations of an OCA $M = \langle S, \#, T, \delta, F \rangle$. The computation of a successor configuration c_{t+1} of a given configuration c_t is written down in a sequential way as follows. Assume c_{t+1} is computed cell by cell from left to right. That is, we are concerned with subconfigurations of the form $c_{t+1}(1) \cdots c_{t+1}(i) c_t(i+1) \cdots c_t(n)$, where n is the length of the input. For technical reasons, in $c_{t+1}(i)$ we have to store both the successor state, which is entered in time step $t+1$ by cell i, and its former state. In this way, the computation of the successor configuration of M can be written as a sequence of n subconfigurations, and configuration c_{t+1} can be represented by $w^{(t+1)} = w_1^{(t+1)} \cdots w_n^{(t+1)}$ such that $w_i^{(t+1)} \in \#S^*(S \times S)S^*$, for $1 \leq i \leq n$, with $w_i^{(t+1)} = \#c_{t+1}(1) \cdots c_{t+1}(i-1)(c_{t+1}(i), c_t(i))c_t(i+1) \cdots c_t(n)$. The valid computations VALC(M) are now defined to be the set of words of the form $w^{(0)} w^{(1)} \cdots w^{(m)}$, where $m \geq 3$, $w^{(t)} \in (\#S^*(S \times S)S^*)^+$ are configurations of M, $1 \leq t \leq m$, $w^{(0)}$ is an initial configuration having the form $\#(T')^+$, where T' is a primed copy of the input alphabet T with $T' \cap S = \emptyset$, $w^{(m)}$ is an accepting configuration of the form $(\#S^*(S \times S)S^*)^* \#S^*(F \times S)$, and $w^{(t+1)}$ is the successor configuration of $w^{(t)}$, for $0 \leq t \leq m-1$.

For the constructions of DRCPCFA accepting the set VALC(M), we provide an additional technical transformation of the input alphabet. Let $S' = S \cup T'$ and $A = \{\#\} \cup S' \cup S'^2$ be the alphabet over which $\mathrm{VALC}(M)$ is defined. We consider the mapping $f : A^+ \to (A \times A)^+$ which is defined for words of length at least two by $f(x_1x_2\cdots x_n) = [x_1,x_2][x_2,x_3]\cdots[x_{n-1},x_n]$. From now on we consider $\mathrm{VALC}(M) \subseteq (A\times A)^+$ to be the set of valid computations to which f has been applied. The set of *invalid computations* $\mathrm{INVALC}(M)$ is then the complement of $\mathrm{VALC}(M)$ with respect to the alphabet $A \times A$.

The following example illustrates the definitions.

Example 7 We consider the following computation of an OCA M over the input alphabet $\{c,d\}$. The initial configuration is $c_0 = (c,d,d)$. Let the successor configurations be $c_1 = (p_1,r_1,s_1)$, $c_2 = (p_2,r_2,s_2)$, and $c_3 = (p_3,r_3,s_3)$. Furthermore, let s_3 be an accepting state, that is, cdd is an accepted input. These configurations are written down as sequences of subconfigurations as follows.

$$\begin{aligned}
w^{(0)} &= \#c'd'd' \\
w^{(1)} &= \#(p_1,c)dd\#p_1(r_1,d)d\#p_1r_1(s_1,d) \\
w^{(2)} &= \#(p_2,p_1)r_1s_1\#p_2(r_2,r_1)s_1\#p_2r_2(s_2,s_1) \\
w^{(3)} &= \#(p_3,p_2)r_2s_2\#p_3(r_3,r_2)s_2\#p_3r_3(s_3,s_2)
\end{aligned}$$

Then,

$$\begin{aligned}
&f(w^{(0)}w^{(1)}w^{(2)}w^{(3)}) = [\#,c'][c',d'][d',d'][d',\#][\#,(p_1,c)][(p_1,c),d][d,d][d,\#] \\
&[\#,p_1][p_1,(r_1,d)][(r_1,d),d][d,\#][\#,p_1][p_1,r_1][r_1,(s_1,d)][(s_1,d),\#][\#,(p_2,p_1)] \\
&[(p_2,p_1),r_1][r_1,s_1][s_1,\#][\#,p_2][p_2,(r_2,r_1)][(r_2,r_1),s_1][s_1,\#][\#,p_2][p_2,r_2] \\
&[r_2,(s_2,s_1)][(s_2,s_1),\#][\#,(p_3,p_2)][(p_3,p_2),r_2][r_2,s_2][s_2,\#][\#,p_3][p_3,(r_3,r_2)] \\
&[(r_3,r_2),s_2][s_2,\#][\#,p_3][p_3,r_3][r_3,(s_3,s_2)]
\end{aligned}$$

is a valid computation of M.

The length of a valid computation can be easily calculated.

Lemma 8 *Let M be an OCA on input $w_1w_2\cdots w_n$ which is accepted after t time steps. Then the length of the corresponding valid computation is $n+(n+1)\cdot n\cdot t$.*

The next lemma is the key tool for the reductions.

Lemma 9 *Let M be an OCA. Then language*

$$VALC'(M) = \{\, \$_1x_1x_2\cdots x_m\$_2a^{2^0}bba^{2^1}bb\cdots bba^{2^{m-1}}bb\& \mid m \geq 1, x_1x_2\cdots x_m \in VALC(M)\,\}$$

belongs to $\mathscr{L}(O(\log(n))$-DRCPCFA$(4))$.

Proof In [1] a $O(n)$-DRCPCFA(3) is constructed that accepts $\mathrm{VALC}(M)$. Basically, the master component A_1 and component A_2 are used to verify that after every subconfiguration the correct successor subconfiguration is given, whereas component A_3 is used to check the correct format of the input. This construction can be implemented identically for the present construction if we interpret $\$_2$ as the right endmarker. Additionally, component A_4 is used in the same way as component A_3 in the construction of Lemma 4, that is, initially it reads $\$_1$ and x_1, stores x_1 in its state, and waits at position 2 until it is queried. After being reset to its initial state, it again reads the next input symbol, stores it, and waits.

When $x_1x_2\cdots x_m \in \mathrm{VALC}(M)$ is tested, the master A_1 and component A_2 are both located at $\$_2$. The second part of the input is now tested along the line of the construction given in the proof of Lemma 4, where the master plays the role of the master, component A_2 the role of component A_2, and component A_4 the role of component A_3.

The length of a word $w \in \text{VALC}'(M)$ is $|w| = 3m + 3 + \sum_{i=0}^{m-1} 2^i = 2^m + 3m + 2$, for some $m \geq 1$. The test whether $x_1 x_2 \cdots x_m$ belongs to $\text{VALC}(M)$ requires $O(m)$ communications. For the remaining tests additional $O(m)$ communications are necessary as the proof of Lemma 4 shows. So, altogether, $O(m)$ communications are sufficient which is of order $O(\log(|w|))$. □

The set of *invalid computations* $\text{INVALC}'(M)$ is simply defined to be the complement of $\text{VALC}'(M)$ with respect to the alphabet $\{a, b, \$_1, \$_2, \&\} \cup (A \times A)$.

Lemma 10 *Let M be an OCA. Then language INVALC$'(M)$ belongs to $\mathscr{L}(O(\log(n))$-DRCPCFA$(4))$.*

Proof To accept the set of invalid computations $\text{INVALC}'(M)$ almost the same construction as for Lemma 9 can be used. The only adaption concerns acceptance and rejection. Since the only possibility to accept is that the master halts in state $accept$ while the other components are non-halting, accepting computations can be made rejecting by sending the master into a halting non-accepting state $reject$ instead. In order to make rejecting computations accepting, it is now sufficient to send the components into some halting accepting state whenever they would halt rejecting. □

Theorem 11 *For any degree $k \geq 4$, emptiness, finiteness, infiniteness, universality, inclusion, equivalence, regularity, and context-freeness are not semi-decidable for $O(\log(n))$-DRCPCFA(k).*

Proof All these problems are known to be non-semi-decidable for OCA [11]. By standard techniques (cf., for example, [1]) the OCA problems are reduced to $O(\log(n))$-DRCPCFA(k) via the valid and invalid computations and Lemmas 9 and 10. □

4.2 Undecidability of Communication Boundedness

This subsection is devoted to questions concerning the decidability or computability of the communication bounds. In principle, we deal with three different types of problems. The first type is to decide for a given DRCPCFA(k) A and a given function f whether or not A is communication bounded by f. The next theorem solves this problem negatively for all non-trivial communication bounds and all degrees $k \geq 3$.

Theorem 12 *Let $k \geq 3$ be any degree, $f \in o(n)$, and A be a DRCPCFA(k). Then it is not semi-decidable whether A is communication bounded by f.*

Proof Let A be a DRCPCFA(k) with $k \geq 3$ accepting some language $L(A) \subseteq \Sigma^*$. We take two new symbols $\{a, \$\} \cap \Sigma = \emptyset$ and construct a DRCPCFA(k) A' accepting language $a^*\$L(A)$. The idea of the construction is that, initially, all components move synchronously across the leading a-block. During this phase, the master component queries one of the non-master components in every time step. When all components have read the separating symbol \$, they enter the initial state of the corresponding component of A. Subsequently, A is simulated, thus testing whether the remaining input belongs to $L(A)$. So, on input $a^n\$w$ with $n \geq 1$ and $w \in L(A)$, A' performs at least n communications. In particular, for $n \geq |w|$ we obtain words that show that A' is not communication bounded by any function $f \in o(n)$, unless $L(A)$ is empty. So, A' is a f-DRCPCFA(k) if and only if $L(A) = \emptyset$.

Since in [1] it has been shown that emptiness is not semi-decidable for DRCPCFA with at least three components, the theorem follows. □

Mitrana considers in [13] the *degree of communication* of parallel communicating finite automata systems. The degree of communication of an accepting computation is defined as the number of queries posed. The degree of communication $Comm(x)$ of a nondeterministic PCFA A on input x is defined as the minimal number of queries posed in accepting computations on x. The degree of communication $Comm(A)$ of a PCFA A is then defined as $\sup\{Comm(x) \mid x \in L(A)\}$. Here we have the second type of problems we are dealing with. Mitrana raised the question whether the degree of communication $Comm(A)$ is computable for a given nondeterministic PCFA(k) A. Since $Comm(A)$ is either finite or infinite, in our terms the question is to decide whether or not A is communication bounded by some function $f \in O(1)$ and, if it is, to compute the precise constant. The next theorem solves the problem.

Theorem 13 *Let $k \geq 3$ be an integer. Then the degree of communication $Comm(A)$ is not computable for DRCPCFA(k).*

Proof For a given DRCPCFA(k) A and new input symbols a and \$, we construct a DRCPCFA(k) A' accepting the language $a^*\$L(A)$ as in the proof of Theorem 12.

Now, we claim that $Comm(A') = 0$ if and only if $L(A) = \emptyset$. If $L(A)$ is empty, then A' accepts the empty set and, thus, $Comm(A') = 0$. On the other hand, if $L(A)$ is not empty, then $Comm(A') > 0$ by construction of A'. Since emptiness is not semi-decidable for DRCPCFA(k) with $k \geq 3$ [1], the theorem follows. □

Now we turn to the last type of problems we are dealing with in this section. The question is now whether the degree of communication is computable for the *language* accepted by a given nondeterministic PCFA(k) A. In [13] the degree of communication $Comm_X(L)$ of a language L is defined as $\inf\{Comm(A) \mid A \text{ is device of type } X \text{ and } L(A) = L\}$. Mitrana showed in [13] that $Comm_{CPCFA}(L(A))$ for some nondeterministic CPCFA A is not computable. He leaves as an open question whether the degree is computable for RCPCFA. Here we are going to show that the degree is not even computable for deterministic RCPCFA.

Lemma 14 *Let $k \geq 3$ be an integer. Then the degree of communication $Comm_{DRCPCFA(k)}(L(A))$ is not computable.*

Proof For a given DRCPCFA(k) A over alphabet Σ and new input symbols $b, 0, 1, \$_1, \$_2$, we construct a DRCPCFA$(k)$ A' accepting the language

$$\{w_1w_2\cdots w_m b w_1w_2\cdots w_m \mid m \geq 1, w_i \in \{0,1\}, 1 \leq i \leq m\}\$_1\$_2L(A).$$

We present the construction for $k = 3$. The generalization to larger k is straightforward.

The idea of the construction is that in a first phase master component A_1 and a non-master component A_2 check the correctness of the prefix $w_1w_2\cdots w_m b w_1w_2\cdots w_m$. This is done as in the construction of Lemma 5. Component A_3 checks the correct format of the input up to the separating symbol $\$_1$ and waits on symbol $\$_2$ until it is queried. At the end of this phase, the master is on the $\$_1$ and component A_2 stays on the symbol b.

In a second phase, the master component stays on $\$_1$ and repeatedly queries component A_2 until this one has read $\$_1$ and, thus, stays on $\$_2$. Now the master reads $\$_1$ and queries component A_2. After being reset to its initial state, component A_2 reads $\$_2$ and performs one λ-step. Then it changes to the initial state of A_2 in A. During this λ-step, the master component reads $\$_2$ and queries component A_3. Then it changes to the initial state of the master of A. Finally, after being reset to its initial state, component A_3 reads $\$_2$ and changes into the initial state of A_3 in A.

Now, all components are in their initial states on the first symbol of the input of A and in a third phase A is simulated. We claim that $Comm(L(A')) = 0$ if and only if $L(A) = \emptyset$. If $L(A)$ is empty, then A' accepts the empty set and $Comm(L(A')) = Comm(\emptyset) = 0$. If $L(A)$ is not empty, we fix some $x \in L(A)$. Assume contrarily that $Comm(L(A')) = 0$. Then there exists a DRCPCFA(k) B accepting $L(A')$ such that $Comm(B) = 0$. From B a DRCPCFA($k+1$) B' is constructed by providing an additional component which checks whether the suffix is precisely x, and halts non-accepting if an error is found. So, B' accepts the language

$$\{\, w_1w_2\cdots w_m b w_1 w_2 \cdots w_m \mid m \geq 1, w_i \in \{0,1\}, 1 \leq i \leq m \,\}\$_1\$_2 x$$

and we still have $Comm(B') = 0$. Similar as in the proof of Lemma 5, it follows by an incompressibility argument that this conclusion leads to a contradiction.

Since emptiness is not semi-decidable for DRCPCFA(k) with $k \geq 3$ [1], the degree of communication $Comm_{DRCPCFA(k)}(L(A))$ is not computable. □

5 An Infinite Hierarchy

In this section, we are going to show that there is an infinite strict hierarchy of language classes in between $\mathscr{L}(O(\log(n))$-DRCPCFA$(k))$ and $\mathscr{L}(O(\sqrt{n})$-DRCPCFA$(k))$, for any $k \geq 4$. To this end, we consider functions $f : \mathbb{N} \to \mathbb{N}$ that are time-computable by one-way cellular automata. That means, given any unary input of length $n \geq 1$, say a^n, the rightmost cell has to enter an accepting state exactly after $f(n)$ time steps and never before. Time-computable functions in OCA have been studied in [3], where it is shown that, for any $r \geq 1$, there exists an OCA-time-computable function $f \in \Theta(n^r)$. We will use this result in the sequel. So, let M_r be an OCA that time-computes $f \in \Theta(n^r)$, for $r \geq 1$. We will use

$$\begin{aligned} L_r = \{\, & \$_1 x_1 x_2 \cdots x_\ell \$_2 w'_1 w'_2 \cdots w'_m w_{m+1} \cdots w_\ell \$_3 w'_1 w'_2 \cdots w'_m w_{m+1} \cdots w_\ell \$_4 a^{2^0} bba^{2^1} bb \cdots a^{2^{m-1}} bb\& \mid \\ & m \geq 1, x_1x_2\cdots x_\ell \text{ is the valid computation of } M_r \text{ on input } a^m, \\ & w'_i \in \{0',1'\}, 1 \leq i \leq m,\ w_i \in \{0,1\}, m+1 \leq i \leq \ell \,\} \end{aligned}$$

as witness languages for the infinite hierarchy.

Lemma 15 *Let $r \geq 1$ be an integer. Then language L_r belongs to $\mathscr{L}(O(\log(n)^{r+2})$-DRCPCFA$(4))$.*

Proof An $O(\log(n)^{r+2})$-DRCPCFA(4)) A accepting L_r works in five phases.

As mentioned before, in [1] an $O(n)$-DRCPCFA(3) is constructed that accepts VALC(M), where the master component A_1 and component A_2 are used to verify the subconfigurations, and component A_3 is used to check the correct format of the input. In the first phase, A simulates this behavior where $\$_2$ plays the role of the endmarker. When $x_1x_2\cdots x_\ell \in$ VALC(M) has been tested, the master A_1 and component A_2 are both located on the symbol after $\$_2$, that is, on w'_1. Additionally, component A_4 initially reads $\$_1$ and waits on x_1 to be queried. The total number of communications in this phase is of order $O(\ell)$.

In the second phase, it is verified that there are as many symbols in between $\$_1$ and $\$_2$ as in between $\$_2$ and $\$_3$, that is, the length ℓ is matched. Furthermore, it is checked whether there are exactly m symbols of the second infix primed. Since $x_1x_2\cdots x_\ell$ describes an OCA computation on some unary input a^m, the initial configuration of the OCA is of the form $\#(a')^m$. Therefore, the valid computation begins with $[\#,a'][a',a']^{m-1}[a',\#]$ followed by symbols not containing primed versions of other symbols. As in the constructions before, the master A_1 moves to the right while querying component A_4 in every step. Whenever component A_4 is reset to its initial state, it reads the next input symbol, remembers it, and waits. In this way, component A_4 is tracked over the valid computations. Moreover, the master A_1 receives information about the symbols read by A_4 and can check the number of primed symbols to be m. The phase ends successfully when A_1 has read $\$_3$ and receives the information that A_4 has read $\$_2$ in this moment, that is, both infixes have the same length ℓ. This phase takes $O(\ell)$ communications. At its end, the master A_1 is located on the symbol after $\$_3$ and components A_2 and A_4 are both located on the symbol after $\$_2$.

The third phase is used to compare the word in between $\$_2$ and $\$_3$ with the word in between $\$_3$ and $\$_4$. Similar as in the phase before, to this end, the master A_1 moves to the right while querying component A_2 in every step. Whenever component A_2 is reset to its initial state, it reads the next input symbol, remembers it, and waits. So, A_1 can check whether the currently read symbols are identical. The phase ends successfully when A_1 has read $\$_4$ and receives the information that A_2 has read $\$_3$ in this moment. Now, the master A_1 is located on the symbol after $\$_4$, A_2 is located on the symbol after $\$_3$, and A_4 still on the symbol after $\$_2$. The total number of communications in this phase is of order $O(\ell)$.

The fourth phase is used to track component A_2 to the position of A_1. So, the master A_1 loops on its position while it queries A_2 in every step. In this way, A_2 moves to the right. The phase ends when A_1 receives the information that A_2 has read $\$_4$. At this time step the master A_1 and component A_2 are located on the symbol after $\$_4$ and A_4 still on the symbol after $\$_2$. During this phase $O(\ell)$ communications take place.

The fifth and final phase is to check the suffix. The master knows that this phase starts and changes into some appropriate state in a λ-step. The situation is similar for component A_2. It is in its initial state on a symbol a for the first time. So, both synchronously start the phase. Basically, here we can use again the construction of the proof of Lemma 9. That is, the master component and component A_2 check that the lengths of a-blocks are doubling. Communication takes place at both symbols b. Reading the first b, component A_4 is queried and forced to proceed one input symbol in order to check the correct number m of a-blocks. Since component A_4 is tracked over an infix whose first m symbols are primed this can be done almost as before. Reading the second b, the master queries component A_2 to ensure that the a-blocks ended correctly. The total number of communications in this phase is of order $O(m)$. This concludes the construction of A.

The length ℓ of the valid computation of M_r on input a^m is of order $\Theta(m^2 \cdot m^r) = \Theta(m^{r+2})$ by Lemma 8. The length of an input is $n = 3\ell + 2^m - 1 + 2m + 5 \in \Theta(2^m)$. The total number of communications is of order $O(\ell) + O(\ell) + O(\ell) + O(\ell) + O(m) = O(m^{r+2})$. So, the number of communications is of order $O(\log(n)^{r+2})$. □

Lemma 16 *Let $r \geq 1$ be an integer. Then language L_r does not belong to $\mathscr{L}(O(\log(n)^r)$-DRCPCFA$(4))$.*

Proof The proof is along the line of the proof of Lemma 6. By way of contradiction, we assume that L_r is accepted by some $O(\log(n)^r)$-DRCPCFA(4).

Let z be a word in L_r whose infix $x = x_1x_2\cdots x_\ell$ is the valid computation of M_r on input a^m. Then $|z|$ is of order $\Theta(2^m)$ and ℓ is of order $\Theta(m^{r+2})$. We will use an incompressibility argument and choose a string $w = w_1w_2\cdots w_\ell \in \{0,1\}^*$ so that the Kolmogorov complexity is $C(w) \geq |w| = \ell \in \Theta(m^{r+2})$. Then the word $z' = \$_1x\$_2w'_1w'_2\cdots w'_mw_{m+1}\cdots w_\ell\$_3w'_1w'_2\cdots w'_mw_{m+1}\cdots w_\ell\$_4a^{2^0}bba^{2^1}bb\cdots a^{2^{m-1}}bb\&$ belongs to L_r as well.

With the help of the accepting computation on z' we write down a program that uniquely reconstructs w. The order of magnitude of the size of the program is given by the product of the size of one snapshot and the number of all snapshots. Since one snapshot can be described by $O(m)$ bits and the number of snapshots is bounded by $O(m^r)$, we derive that $C(w)$ is of order $O(m^{r+1})$, a contradiction. □

Combining Lemma 15 and Lemma 16 the desired infinite hierarchy of the next theorem follows.

Theorem 17 *Let $r \geq 1$ be an integer. Then the class $\mathscr{L}(O(\log(n)^r)$-DRCPCFA$(4))$ is properly included in the class $\mathscr{L}(O(\log(n)^{r+2})$-DRCPCFA$(4))$.*

Since the proofs of Lemma 15 and Lemma 16 do not rely on a specific number of components as long as at least four components are provided, the hierarchy follows for any number of components $k \geq 4$.

Corollary 18 *Let $k \geq 4$ and $r \geq 1$ be two integers. Then the class $\mathscr{L}(O(\log(n)^r)$-DRCPCFA$(k))$ is properly included in the class $\mathscr{L}(O(\log(n)^{r+2})$-DRCPCFA$(k))$.*

References

[1] Henning Bordihn, Martin Kutrib & Andreas Malcher (2011): *Undecidability and Hierarchy Results for Parallel Communicating Finite Automata*. *Int. J. Found. Comput. Sci.* 22, pp. 1577–1592, doi:10.1142/S0129054111008891.

[2] Henning Bordihn, Martin Kutrib & Andreas Malcher (2012): *On the Computational Capacity of Parallel Communicating Finite Automata*. *Int. J. Found. Comput. Sci.* 23, pp. 713–732, doi:10.1142/S0129054112500062.

[3] Thomas Buchholz & Martin Kutrib (1998): *On time computability of functions in one-way cellular automata*. *Acta Inform.* 35, pp. 329–352, doi:10.1007/s002360050123.

[4] Ashish Choudhary, Kamala Krithivasan & Victor Mitrana (2007): *Returning and non-returning parallel communicating finite automata are equivalent*. *RAIRO Inform. Théor.* 41, pp. 137–145, doi:10.1051/ita:2007014.

[5] John E. Hopcroft & Jeffrey D. Ullman (1979): *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley.

[6] Oscar H. Ibarra (1973): *On Two-way Multihead Automata*. *J. Comput. System Sci.* 7, pp. 28–36, doi:10.1016/S0022-0000(73)80048-0.

[7] Martin Kutrib (2008): *Cellular Automata – A Computational Point of View*. In: *New Developments in Formal Languages and Applications*, chapter 6, Springer, pp. 183–227, doi:10.1007/978-3-540-78291-9_6.

[8] Martin Kutrib (2009): *Cellular Automata and Language Theory*. In: *Encyclopedia of Complexity and System Science*, Springer, pp. 800–823, doi:10.1007/978-0-387-30440-3_54.

[9] Martin Kutrib & Andreas Malcher (2011): *Two-Party Watson-Crick Computations*. In: *Implementation and Application of Automata (CIAA 2010)*, *LNCS* 6482, Springer, pp. 191–200, doi:10.1007/978-3-642-18098-9_21.

[10] Ming Li & Paul M. B. Vitányi (1993): *An Introduction to Kolmogorov Complexity and Its Applications*. Springer, doi:10.1007/978-1-4757-3860-5

[11] Andreas Malcher (2002): *Descriptional Complexity of Cellular Automata and Decidability Questions*. *J. Autom., Lang. Comb.* 7, pp. 549–560.

[12] Carlos Martín-Vide, Alexandru Mateescu & Victor Mitrana (2002): *Parallel Finite Automata Systems Communicating by States*. *Int. J. Found. Comput. Sci.* 13, pp. 733–749, doi:10.1142/S0129054102001424.

[13] Victor Mitrana (2000): *On the Degree of Communication in Parallel Communicating Finite Automata Systems*. *J. Autom., Lang. Comb.* 5, pp. 301–314.

[14] Friedrich Otto (2013): *Asynchronous PC systems of pushdown automata*. In: *Language and Automata Theory and Applications (LATA 2013)*, *LNCS* 7810, Springer, pp. 456–467, doi:10.1007/978-3-642-37064-9_40.

[15] Marcel Vollweiler (2013): *Asynchronous systems of parallel communicating finite automata*. In: *Fifth Workshop on Non-Classical Models for Automata and Applications (NCMA 2013)*, *books@ocg.at* 294, Austrian Computer Society, Vienna, pp. 243–257.

Languages of lossless seeds

Karel Břinda
Laboratoire d'Informatique Gaspard Monge
Université Paris-Est Marne-la-Vallée
Paris, France
karel.brinda@univ-mlv.fr

Several algorithms for similarity search employ seeding techniques to quickly discard very dissimilar regions. In this paper, we study theoretical properties of lossless seeds, i.e., spaced seeds having full sensitivity. We prove that lossless seeds coincide with languages of certain sofic subshifts, hence they can be recognized by finite automata. Moreover, we show that these subshifts are fully given by the number of allowed errors k and the seed margin ℓ. We also show that for a fixed k, optimal seeds must asymptotically satisfy $\ell \in \Theta(m^{\frac{k}{k+1}})$.

1 Introduction

The annual volume of data produced by the Next-Generation Sequencing technologies has been rapidly increasing; even faster than growth of disk storage capacities. Thus, new efficient algorithms and data-structures for processing, compressing and storing these data, are needed.

Similarity search represents the most frequent operation in bioinformatics. In huge DNA databases, a two-phase scheme is the most widely used approach to find all occurrences of a given string up to some Hamming or Levenshtein distance. First of all, most of dissimilar regions are discarded in a fast *filtration phase*. Then, in a *verification phase*, only "hot candidates" on similarity are processed by classical time-consuming algorithms like Smith-Waterman [23] or Needleman-Wunsch [17].

Algorithms for the filtration phase are often based on so-called *seed filters* which make use of the fact that two strings of the same length m being in Hamming distance k must necessarily share some exact patterns. These patterns are represented as strings over the alphabet $\{\texttt{\#},\texttt{-}\}$ called *seeds*, where the "matching" symbol `#` corresponds to a matching position and the "joker" symbol `-` to a matching or a mismatching position.

For instance, for two strings of length 15, matching within two errors, shared patterns are, e.g., `##-#--##-#` or `#####`. For illustration, if we consider that two strings match as `===X=====X=====` (where the symbols `=` and `X` represent respectively matching and mismatching positions), then the corresponding seed positions can be following:

```
===X=====X=====
.##-#--##-#....
....#####......
```

As the second seed is the longest possible contiguous seed in this case, we observe the main advantage of spaced seeds in comparison to contiguous seeds: for the same task, there exist spaced seeds with higher number of `#`'s (so-called *weight*).

Two basic characteristics of every seed are *selectivity* and *sensitivity*. Selectivity measures restrictivity of a filter created from the seed. In general, higher weight implies better selectivity of the filter. *Lossless seeds* are those seeds having full sensitivity. They are easier to handle mathematically on one hand, but attain lower weight on the other hand. *Lossy seeds* are employed for practical purposes more since a small decrease in sensitivity can be compensated by considerable improvement of selectivity.

Nevertheless, only lossless seeds are considered in this paper. For a given length m of strings to be compared and a given number of allowed mismatches k (such setting is called (m,k)-*problem*), the aim is to design fully sensitive seeds with highest possible weight.

1.1 Literature

The idea of lossless seeds was originally introduced by Burkhardt and Karkkäinen [3, 4]. Let us remark that lossy spaced seed were used in the same time in the PatternHunter program [16]. Generalization of lossless seeds was studied by Kucherov et al. [11]. given seed are required (the pattern is shared at more positions). The authors also proved that, for a fixed number k of mismatches, *optimal seeds* (i.e., seeds with the highest possible weight among all seeds solving the given problem) must asymptotically satisfy $m - w(m) \in \Theta(m^{\frac{k}{k+1}})$, where $w(m)$ denotes the maximal possible weight of a seed solving the (m,k)-problem. They also started a systematic study of seeds created by repeating of short patterns. Afterwards, the results on asymptotic properties of optimal seeds were generalized by Farach-Colton et al. [10]. Computational complexity of optimal seed construction was derived by Nicolas and Rivals [18, 19].

Further, the theory on lossless seed was significantly developed by Egidi and Manzini. First, they studied seeds designed from mathematical objects called perfect rulers [6, 9]. The idea of utilization of some type of "rulers" was later independently extended by KB [2] (cyclic rulers) and, again, Edigi and Manzini [8] (difference sets). In [8], these ideas were extended also to seed families. Cyclic rulers and difference sets mathematically correspond to each other. Edigi and Manzini [7] also showed possible usage of number-theoretical results on quadratic residues for seed design.

In practice, seeds often find their use in short-read mappers implementing hash tables (for more details on read mapping, see, e.g., [12, 22]). ZOOM [13] and PerM [5] are examples of mappers utilizing lossless seeds.

A list of papers on spaced seed is regularly maintained by Noé [20].

1.2 Our object of study

One of the most important theoretical aspects of lossless seeds are their structural properties. Whereas good lossy seeds usually show irregularity, it was observed that good lossless seeds are often repetitions of short patterns ([11, 5, 2, 8]). The question whether optimal seeds can be constructed in all cases by repeating patterns, which would be short with respect to seed length, remains open (see [2, Conjecture 1]). Its answering would have practical impacts in development of bioinformatical software tools since the search space of programs for lossless seeds design could be significantly cut and also indexes in programs using lossless seeds for approximate string matching could be more memory efficient (like [5]).

1.3 Paper organization and results

In this paper, we follow and further develop ideas from [2]. We concentrate on a parameter ℓ called seed margin, which is the difference between the size m of compared strings and the length of a seed.

In Section 2 we recall the notation used in combinatorics on words and symbolic dynamics. In Section 3 we formally define seeds and (m,k)-problems. Then we transform the problem of seed detection into another criterion (Theorem 1) and also show asymptotic properties of ℓ for optimal seeds (Proposition 1). In Section 4 we prove that sets of seeds, obtained by fixing the parameters k and ℓ, coincide with languages of some sofic subshifts. Therefore, those sets of seeds are recognized by finite automata. In Section 5 we show applications of obtained results for seed design. These results provide a new view on lossless seeds and explain their periodic properties.

2 Preliminaries

Throughout the paper, we use a standard notation of combinatorics on words and symbolic dynamics.

2.1 Combinatorics on words

An *alphabet* $\mathcal{A} = \{a_0, \dots, a_{m-1}\}$ is a finite set of symbols called *letters*. In this paper, we will work exclusively with the alphabet $\{\texttt{\#}, \texttt{-}\}$ A finite sequence of letters from $\mathcal{A}$ is called a *finite word* (over $\mathcal{A}$). The set $\mathcal{A}^*$ of all finite words (including the empty word ε) provided with the operation of concatenation is a free monoid. The concatenation is denoted multiplicatively. If $w = w_0 w_1 \cdots w_{n-1}$ is a finite word over $\mathcal{A}$, we denote its length by $|w| = n$. We deal also with bi-infinite sequences of letters from $\mathcal{A}$ called *bi-infinite words* $\mathbf{w} = \cdots \mathbf{w}_{-2}\mathbf{w}_{-1}|\mathbf{w}_0\mathbf{w}_1\mathbf{w}_2\cdots$ over $\mathcal{A}$. The sets of all bi-infinite words over $\mathcal{A}$ is denoted by $\mathcal{A}^{\mathbb{Z}}$.

A finite word w is called a *factor* of a word $\mathbf{u}$ ($\mathbf{u}$ being finite or bi-infinite) if there exist words p and s (finite or one-side infinite) such that $\mathbf{u} = pws$. For given indexes i and j, the symbol $\mathbf{u}[i, j]$ denotes the factor $\mathbf{u}_i\mathbf{u}_{i+1}\cdots\mathbf{u}_j$ if $i \le j$, or ε if $i > j$. A concatenation of k words w is denoted by w^k. The set of all factors of a word $\mathbf{u}$ ($\mathbf{u}$ being finite or bi-infinite) is called the language of $\mathbf{u}$ and denoted by $\mathcal{L}(\mathbf{u})$. Its subset $\mathcal{L}(\mathbf{u}) \cap \mathcal{A}^n$ containing all factors of $\mathbf{u}$ of length n is denoted by $\mathcal{L}_n(\mathbf{u})$.

Let us remark that this notation will be used extensively in the whole text. For instance $\mathbf{w}[2,5]\texttt{-}^4$ denotes the word created by concatenation of the factor $\mathbf{w}_2\mathbf{w}_3\mathbf{w}_4\mathbf{w}_5$ of a bi-infinite word $\mathbf{w}$ and the word $\texttt{----}$. Similarly, for a finite word v of length n, by $\cdots\texttt{--}|v\texttt{--}\cdots$ we denote the bi-infinite word $\mathbf{u}$ such that for all $i \in \{0, \dots, n-1\}(\mathbf{u}_i = w_i)$ and for all $i \in \mathbb{Z} \setminus \{0, \dots, n-1\}(\mathbf{u}_i = \texttt{-})$. For more information about combinatorics on words, we can refer to Lothaire I [15].

2.2 Symbolic dynamics

Consider an alphabet $\mathcal{A}$. On the set $\mathcal{A}^{\mathbb{Z}}$ of bi-infinite words over $\mathcal{A}$, we define a so-called Cantor metric d as

$$d(\mathbf{u}, \mathbf{v}) = \begin{cases} 0 & \text{if } \mathbf{u} = \mathbf{v}, \\ 2^{-s} & \text{if } \mathbf{u} \neq \mathbf{v}, \text{ where } s := \min\left\{|i| \;\middle|\; \mathbf{u}_i \neq \mathbf{v}_i\right\}. \end{cases}$$

We define a *shift* operation σ as $[\sigma(\mathbf{u})]_i = \mathbf{u}_{i+1}$ for all $i \in \mathbb{Z}$. The map σ is invertible, and the power σ^k is defined by composition for all $k \in \mathbb{Z}$. The map σ is continuous on $\mathcal{A}^{\mathbb{Z}}$, therefore, $(\mathcal{A}^{\mathbb{Z}}, \sigma)$ is a dynamical system, which is called a *full shift*.

A bi-infinite word $\mathbf{u} \in \mathcal{A}^{\mathbb{Z}}$ *avoids* a set of finite words X if $\mathcal{L}(\mathbf{u}) \cap X = \emptyset$. By S_X we denote the set of all bi-infinite words that avoid X and we call it a *subshift*. If X is a regular language, S_X is called *sofic subshift*; if X is finite, S_X is called a *subshift of finite type*. The *language* $\mathcal{L}(S)$ of a subshift S is the union of languages of all bi-infinite words from S. By $\mathcal{L}_n(S)$ we denote the set $\mathcal{L}(S) \cap \mathcal{A}^n$. It holds that a set $S \subseteq \mathcal{A}^{\mathbb{Z}}$ is a subshift if and only if it is invariant under the shift map σ (that means $\sigma(S) = S$) and it is closed with respect to the Cantor metric. A general theory of subshifts is well summarized in [14].

3 Lossless seeds

In this section, we introduce basic definition formalizing lossless seeds. Then we introduce a parameter ℓ called seed margin and show its asymptotic properties for optimal seeds. Let us recall that m denotes the length of strings to be compared and k denotes the number of allowed mismatches.

Definition 1. *The binary alphabet $\mathcal{A} = \{\texttt{\#}, \texttt{-}\}$ is called* seed alphabet. *Every finite word over this alphabet is a* seed. *The* weight *of a seed Q is the number of occurrences of the letter # in Q.*

Definition 2. *Let m and k be positive integers. Every set $\{i_1, \dots, i_k\} \subseteq \{0, \dots, m-1\}$ is called* error combination *of k errors.*

Consider a seed Q such that $|Q| < m$ and denote $\ell := m - |Q|$, which is the so-called seed margin. *Then Q* detects *an error combination $\{i_1, \dots, i_k\} \subseteq \{0, \dots, m-1\}$ at position $t \in \{0, \dots, \ell\}$ if for all $j \in \{0, \dots, |Q|-1\}$ it holds $\left(Q_j = \texttt{\#} \implies j + t \notin \{i_1, \dots, i_k\}\right)$.*

The seed Q is said to solve *the (m,k)-problem if every error combination $\{i_1, \dots, i_k\} \subseteq \{0, \dots, m-1\}$ of k errors is detected by Q at some position $t \in \{0, \dots, \ell\}$.*

Many combinatorial properties of seeds can be studied from the perspective of bi-infinite words. First, we need a seed analogy of the logical function `OR` applied on bi-infinite words and producing, again, a bi-infinite word.

Definition 3. *Consider k bi-infinite words $\mathbf{u}^{(1)}, \dots, \mathbf{u}^{(k)}$ over $\mathcal{A}$. We define a k-nary operation $\oplus$ as*

$$\forall i \in \mathbb{Z}: \quad (\oplus(\mathbf{u}^{(1)}, \dots, \mathbf{u}^{(k)}))_i = \begin{cases} \texttt{\#} & \text{if } (\mathbf{u}^{(j)})_i = \texttt{\#} \text{ for some } j \in \{1, \dots, k\} \\ \texttt{-} & \text{otherwise} \end{cases}$$

The following theorem will be crucial for seed analysis in the rest of the text. It is mainly a translation of basic definitions to the formalism of shifts and logical operations, but it enables us to easily observe on which parameters (and how) the structure of lossless seeds really depends.

Theorem 1. *Let m and k be positive integers and Q be a seed such that $|Q| < m$. Denote $\ell := m - |Q|$ and $\mathbf{w} := \cdots \texttt{--}|\texttt{-}^{\ell} Q \texttt{--} \cdots$. Then Q detects an error combination $\{i_1, \dots, i_k\} \subseteq \{0, \dots, m-1\}$ at a position $t \in \{0, \dots, \ell\}$ if and only if*

$$\left(\oplus(\sigma^{i_1}(\mathbf{w}), \dots, \sigma^{i_k}(\mathbf{w})\right)_{\ell - t} = \texttt{-}. \tag{1}$$

Proof. Q detects $\{i_1, \dots, i_k\}$ at position t if $\forall j \in \{0, \dots, |Q|-1\}\left(Q_j = \texttt{\#} \implies j + t \notin \{i_1, \dots, i_k\}\right)$. This is equivalent to $\forall p \in \{i_1, \dots, i_k\}(\mathbf{w}_{p-t+\ell} = \texttt{-})$, which is equivalent to (1). $\square$

Corollary 1. *Q does not detect a combination $\{i_1, \dots, i_k\}$ at any position $t \in \{0, \dots, \ell\}$ if and only if $(\oplus(\sigma^{i_1}(\mathbf{w}), \dots, \sigma^{i_k}(\mathbf{w}))[0, \ell] = \texttt{\#}^{\ell+1}$.*

Let us mention that in the case of two errors, Corollary 1 corresponds to the Laser method [2, Section 4.1] (a JavaScript implementation is available at [1]) as we illustrate in the following example.

Example 1. *Consider a seed $Q = \texttt{\#\#-\#-----\#-\#\#}$ of length* 14 *and the $(19,2)$-problem. In Figure 1 we show a corresponding schematic table. Denote $\mathbf{w} := \cdots \texttt{--}|\texttt{-}^5 Q \texttt{--} \cdots$. The words $\oplus(\sigma_i(\mathbf{w}), \sigma_j(\mathbf{w}))$ occur diagonally. It is easily seen from Corollary 1 that Q does not detect the error combination $\{5, 13\}$ since $\ell = 5$ and $\left(\oplus(\sigma^5(\mathbf{w}), \sigma^{13}(\mathbf{w}))\right)[0,5] = \texttt{\#}^{\ell+1}$.*

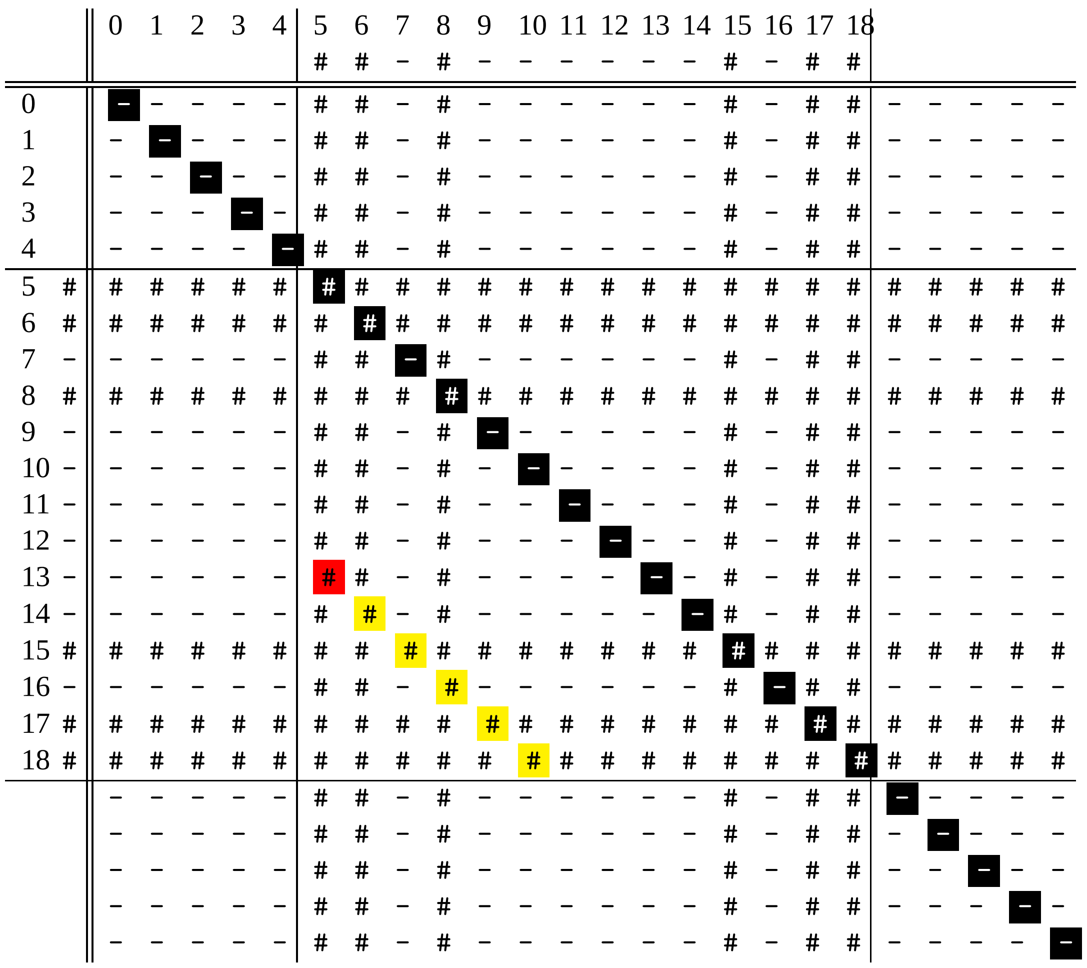


Figure 1: The Laser method for the $(19,2)$-problem and the seed $Q = $ ##-#------#-## in Example 1.

Even though the basic parameters in the concept of (m,k)-problems are m and k; as follows from Theorem 1, the parameters determining structure of seeds are ℓ and k. Therefore, in the next section, we will fix them and study seeds Q solving $(|Q|+\ell,k)$-problems. Hence, when increasing m, the seed must be extended in order to keep ℓ constant.

To complete this section, we show the asymptotic relation of m, ℓ, and k for optimal seeds. Let us fix the parameter k. Let $w(m)$ denote the maximal weight of a seed solving the (m,k)-problem. It was proved in [11, Lemma 4] that $m - w(m) \in \Theta(m^{\frac{k}{k+1}})$. We show that ℓ has the same asymptotic behavior.

Proposition 1. *Let k be a fixed positive integer and $w(m)$ denote the maximal weight of a seed solving the (m,k)-problem. Let $H(m)$ be the set of all seeds with weight $w(m)$ solving the (m,k)-problem. For every positive m, set $\ell(m) := m - |Q|$, where Q is an arbitrary seed from $H(m)$. Then $\ell(m) \in \Theta(m - w(m))$.*

Proof. Since $m \geq \ell(m) + w(m)$, we get trivially the upper bound as $\ell(m) \in \mathcal{O}(m - w(m))$. Now let us prove the lower bound. Let k be fixed. Since $m - w(m) \in \Theta(m^{\frac{k}{k+1}})$ for optimal seeds, it also holds that $(m - w(m))^{k+1} \in \mathcal{O}(m^k)$. From combinatorial considerations on seed detection, we get $\binom{m}{k} \leq \binom{m-w(m)}{k}(\ell+1)$. By combining the last two formulas, we obtain $\ell(m) \in \Omega(m - w(m))$, which concludes the proof. $\square$

4 Seed subshifts

In this section, we show the relation between lossless seeds and subshifts. First, we denote sets of seeds obtained by fixing the parameters ℓ and k. Afterwards, we prove that they coincide with languages of certain sofic subshifts. After defining functions checking the criterion given by Corollary 1 globally on bi-infinite words, we show that the subshift are exactly the sets of bi-infinite words, for which these functions have the upper bound ℓ.

Definition 4. *Let ℓ and k be positive integers. The set of all seeds such that each seed Q solves the $(|Q|+\ell,k)$-problem is denoted by* Seed_k^ℓ.

Example 2. $\mathrm{Seed}_2^3 = \{\varepsilon, \texttt{\#}, \texttt{-}, \texttt{\#-}, \texttt{-\#}, \texttt{--}, \texttt{\#--}, \texttt{-\#-}, \texttt{--\#}, \texttt{---}, \texttt{\#--\#}, \texttt{\#---}, \texttt{-\#--}, \texttt{--\#-}, \texttt{---\#}, \texttt{----}, \ldots\}$.

4.1 Functions sh_k and (ℓ,k)-valid bi-infinite words

Definition 5. *Consider a positive integer k. We define a function* $\mathrm{sh}_k : (\mathcal{A}^{\mathbb{Z}})^k \to \mathbb{N}_0 \cup \{+\infty\}$ *as:*

$$\mathrm{sh}_k(\mathbf{u}^{(1)},\ldots,\mathbf{u}^{(k)}) = \sup_{i_1,\ldots,i_k\in\mathbb{Z}} \sup_{p\in\mathbb{N}_0} \left\{ p \mid \mathbf{v}[0,p-1] = \texttt{\#}^p,\ \textit{where } \mathbf{v} = \oplus\left(\sigma^{i_1}(\mathbf{u}^{(1)}),\ldots,\sigma^{i_k}(\mathbf{u}^{(k)})\right)\right\}. \quad (2)$$

We extend the range of the function $\mathrm{sh}_k(\cdot,\ldots,\cdot)$ to $(\mathcal{A}^*)^k$. Finite words w are transformed into bi-infinite words $\mathbf{v}$ as $\mathbf{v} := \cdots\texttt{--}|w\texttt{--}\cdots$.

Informally said, $sh_k(\mathbf{u}^{(1)},\ldots,\mathbf{u}^{(k)})$ is equal to

- a finite $s\in\mathbb{N}_0$ if after arbitrary "aligning" of the words followed by the logical `OR` operation (in the Laser method the diagonal bi-infinite words), each run of #'s has length at most s and the value s is attained for some "alignment";
- $+\infty$ if there exists an "alignment" with run of infinitely many #'s (e.g., $\mathrm{sh}_2(\cdots vv|vv\cdots, \cdots ww|ww\cdots)$ with $v = \texttt{\#\#-}$ and $w = \texttt{\#--}$).

Every function sh_k is symmetric and shift invariant with respect to all variables. The following observations show how to estimate their values for given k bi-infinite words.

Observation 1 (Lower estimate). *Let $\mathbf{u}^{(1)},\ldots,\mathbf{u}^{(k)}$ be bi-infinite words. If $\oplus(\sigma^{i_1}(\mathbf{u}^{(1)}),\ldots,\sigma^{i_k}(\mathbf{u}^{(k)}))$ has a factor $\texttt{\#}^p$ for some $i_1,\ldots,i_k$; then $\mathrm{sh}_k(\mathbf{u}^{(1)},\ldots,\mathbf{u}^{(k)}) \geq p$.*

Observation 2 (Upper estimate). *Let $\mathbf{u}^{(1)},\ldots,\mathbf{u}^{(k)},\mathbf{v}^{(1)},\ldots,\mathbf{v}^{(k)}$ be bi-infinite words such that $\mathbf{u}^{(1)} \preceq \mathbf{v}^{(1)},\ldots,\mathbf{u}^{(k)} \preceq \mathbf{v}^{(k)}$, where $\preceq$ is a relation defined as*

$$\mathbf{u} \preceq \mathbf{v} \qquad \iff \qquad (\mathbf{u}_i = \texttt{\#} \implies \mathbf{v}_i = \texttt{\#}) \ \textit{ holds for all } i\in\mathbb{Z}. \quad (3)$$

Then $\mathrm{sh}_k(\mathbf{u}^{(1)},\ldots,\mathbf{u}^{(k)}) \leq \mathrm{sh}_k(\mathbf{v}^{(1)},\ldots,\mathbf{v}^{(k)})$.

Bi-infinite words for which the sh_k function is bounded by some ℓ, will be the "bricks" of our subshifts. Their factors Q are exactly those seeds solving $(|Q|+\ell,k)$-problems.

Definition 6. *A bi-infinite word* $\mathbf{u}$ *satisfying* $\mathrm{sh}_k(\mathbf{u},\ldots,\mathbf{u}) \leq \ell$ *is called an* $(\ell;k)$-valid bi-infinite word. *For fixed positive integers ℓ and k, we denote the set of all $(\ell;k)$-valid words by* V_k^ℓ.

Lemma 1. *A seed Q solves the $(|Q|+\ell,k)$-problem if and only if it is a factor of an (ℓ,k)-valid bi-infinite word.*

Proof. $\Longrightarrow$: The word $\mathbf{w} := \cdots\texttt{--}|\texttt{-}^\ell Q\texttt{--}\cdots$ must be (ℓ,k)-valid since otherwise Q would not solve the $(|Q|+\ell,k)$-problem by Corollary 1.

$\Longleftarrow$: For a contradiction assume that there exists a factor Q of a bi-infinite word $\mathbf{u}$, which does not solve the $(|Q|+\ell,k)$-problem. Let the non-detected error combination be $\{i_1,\dots,i_k\}$. Denote $\mathbf{w} = \cdots\text{--}\text{--}|\text{-}^{\ell}Q\text{--}\text{--}\cdots$.

We use shift invariance of sh_k and Observation 2 to get

$$\mathrm{sh}_k(\mathbf{w},\dots,\mathbf{w}) \le \mathrm{sh}_k(\mathbf{u},\dots,\mathbf{u}) \le \ell. \tag{4}$$

Since Q does not detect the error combination $\{i_1,\dots,i_k\}$, it follows from Corollary 1 that

$$(\oplus(\sigma^{i_1}(\mathbf{w}),\dots,\sigma^{i_k}(\mathbf{w}))[0,\ell] = \#^{\ell+1}.$$

Nevertheless, this gives us a lower estimate on $\mathrm{sh}_k(\mathbf{w},\dots,\mathbf{w})$, which is contradicting (4). □

4.2 Subshifts of (ℓ,k)-valid words

The property of (ℓ,k)-validity is preserved under the shift operation. Moreover, the sets V_k^{ℓ} of (ℓ,k)-valid words are subshifts. To prove it, we need to find a criterion for verifying (ℓ,k)-validity based on comparing finite factors of a given bi-infinite word.

Lemma 2. *Let* $\mathbf{u}$ *be a bi-infinite word over the seed alphabet* $\mathcal{A}$. *Then the following statements are equivalent:*

1. $\mathbf{u}$ *is* $(\ell;k)$*-valid;*
2. $\forall v^{(1)},\dots,v^{(k)} \in \mathcal{L}_{\ell+1}(\mathbf{u})\big(\mathrm{sh}_k(v^{(1)},\dots,v^{(k)}) \le \ell\big)$*;*
3. $\forall w^{(1)},\dots,w^{(k)} \in \mathcal{L}_{\ell+1}(\mathbf{u})\big(\oplus(w^{(1)},\dots,w^{(k)}) \ne \#^{\ell+1}\big)$.

Proof. We prove three implications.

$1 \Longrightarrow 2$: Consider any such factors $v^{(1)},\dots,v^{(k)}$. Find their positions $i_1,\dots,i_k$ in $\mathbf{u}$. It holds that

$$\cdots\text{--}\text{--}|v^{(1)}\text{--}\text{--}\cdots \preceq \sigma^{i_1}(\mathbf{u}),\ \ \dots,\ \ \cdots\text{--}\text{--}|v^{(k)}\text{--}\text{--}\cdots \preceq \sigma^{i_k}(\mathbf{u}),$$

where $\preceq$ is the relation defined by (3). By combining the assumption, shift invariance of sh_k, and Observation 2, we obtain $\mathrm{sh}_k(v^{(1)},\dots,v^{(k)}) \le \mathrm{sh}_k(\mathbf{u},\dots,\mathbf{u}) \le \ell$.

$2 \Longrightarrow 3$: It is an easy consequence of the definition of the sh_k function.

$3 \Longrightarrow 1$: For a contradiction assume that $\mathbf{u}$ is not (ℓ,k)-valid. Then there exist integers $i_1,\dots,i_k$ such that $\oplus(\sigma^{i_1}(\mathbf{u}),\dots,\sigma^{i_1}(\mathbf{u}))[0,\ell] = \#^{\ell+1}$. □

The main consequence of Lemma 2 is the fact that every seed must be constructed from reciprocally compatible tiles of length $\ell+1$. To describe this property, we define a relation of compatibility on the set $\mathcal{A}^{\ell+1}$.

Definition 7. *For given positive integers* ℓ *and* k*, we define the* k*-nary* compatibility relation C_k^{ℓ} *on* $\mathcal{A}^{\ell+1}$ *as*

$$\mathrm{C}_k^{\ell}(v^{(1)},\dots,v^{(k)}) \quad\Longleftrightarrow\quad \mathrm{sh}_k(v^{(1)},\dots,v^{(k)}) \le \ell.$$

Corollary 2. *Let* $\mathbf{u}$ *be a bi-infinite word over the seed alphabet* $\mathcal{A}$. *The word* $\mathbf{u}$ *is* (ℓ,k)*-valid if and only if* $\forall v^{(1)},\dots,v^{(k)} \in \mathcal{L}_{\ell+1}(\mathbf{u})\left(\mathrm{C}_k^{\ell}(v^{(1)},\dots,v^{(k)})\right)$.

Now let us prove that (ℓ,k)-valid words really form subshifts. We only need to show that (ℓ,k)-valid words are exactly those words, which can be created from compatible "tiles".

Lemma 3. *Let ℓ and k be positive integers. The set V_k^ℓ of all (ℓ,k)-valid words is a subshift.*

Proof. We prove the lemma by construction of a set X of forbidden words (as they are introduced in 2.2). Take

$$X := \left\{x \in \mathcal{A}^* \mid \exists v^{(1)},\dots,v^{(k)} \in \mathcal{L}_{\ell+1}(x)\left(\neg \mathrm{C}_k^\ell(v^{(1)},\dots,v^{(k)})\right)\right\}.$$

The set X contains all possible finite words having some factors, which are "incompatible" with respect to the given ℓ and k. Hence, the subshift S_X contains exactly all bi-infinite words $\mathbf{u}$ satisfying $\forall v_1,\dots,v_k \in \mathcal{L}_{\ell+1}(\mathbf{u})\left(\mathrm{C}_k^\ell(v_1,\dots,v_k)\right)$ and we obtain $S_X = \mathrm{V}_k^\ell$ by Corollary 2. □

Example 3. *Even though both of the seeds $Q^{(1)} =$ ##-#-- and $Q^{(2)} =$ --#-## solve the $(11,2)$-problem, the seed $Q = Q^{(1)}$--$Q^{(2)}$ does not solve the $(19,2)$-problem as we have seen in Example 1. Since $Q^{(1)} \oplus Q^{(2)} = \#^6$, any seed $\tilde{Q}$ of the form $\tilde{Q} = Q^{(1)}\text{-}^p Q^{(2)}$ cannot solve the $(|\tilde{Q}|+5,2)$-problem.*

It follows from the last example that the subshift V_2^5 of all $(5,2)$-valid words is not of finite type. Nevertheless, every subshift V_k^ℓ must be a union of subshifts of finite type, which can be constructed from so-called (ℓ,k)-generating sets.

Definition 8. *For given positive integers ℓ and k, a subset G of $\mathcal{A}^{\ell+1}$ is called* (ℓ,k)-generating set *if the following conditions are satisfied:*

1. *for all $v^{(1)},\dots,v^{(k)} \in G$, it holds $\mathrm{C}_k^\ell(v^{(1)},\dots,v^{(k)})$;*
2. *it is maximal possible (i.e., it cannot contain any other word from $\mathcal{A}^{\ell+1}$).*

Observation 3. *Let us take a word from an (ℓ,k)-generating set G. If we remove the last or the first letter and concatenate the letter - to the beginning or to the end of the word, we obtain again a word from G. Therefore, every (ℓ,k)-generating set G must contain, e.g., the word $\text{-}^{\ell+1}$.*

Every generating set G fully determines a subshift of finite type, we will denote it by $S(G)$. This subshift contains all bi-infinite words $\mathbf{u}$ such that $\mathcal{L}_{\ell+1}(\mathbf{u}) \subseteq G$.

Definition 9. *Consider a seed Q and an (ℓ,k)-generating set G. By $S(G)$, we denote the subshift S_X of finite type given by $X = \mathcal{A}^{\ell+1}\setminus G$. We say that a seed Q is* generated *by G if $Q \in \mathcal{L}(S(G))$.*

In other words, a seed Q satisfying $|Q| \geq \ell+1$ is generated by G if $\mathcal{L}_{\ell+1}(Q) \subseteq G$. A seed Q such that $|Q| < \ell+1$ is generated by G if $\exists w \in G\big(Q \in \mathcal{L}(w)\big)$. We can also observe that every (ℓ,k)-valid word is generated by some (ℓ,k)-generating set.

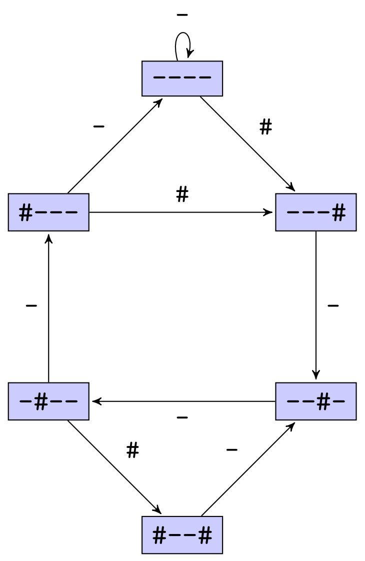
Observation 4. *For every (ℓ,k)-valid bi-infinite word $\mathbf{u}$, there exists an (ℓ,k)-generating set G such that $\mathbf{u} \in S(G)$.*

Example 4. *Continue with the setting from Example 2. Consider the only one $(3,2)$-generating set $G = \{$#--#, #---, -#--, --#-, ---#, ----$\}$. Since $S(G)$ is of finite type, it follows from theory of symbolic dynamics that there exists a strongly connected labeled graph H such that $S(G)$ coincide with labels of all bi-infinite paths in H (for details, see [14]). This graph also determines a finite automaton recognizing the set $\mathcal{L}(S(G))$, i.e., the set of labels of finite paths in H. Such automaton can be created from a de-Bruijn graph. However, it would not be minimal as it is shown in Figure 2.*

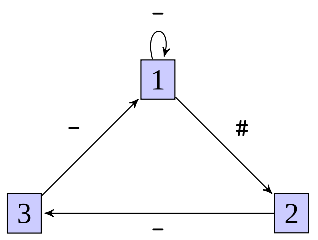
Theorem 2. *Let k and ℓ be positive integers. The set Seed_k^ℓ is a regular language.*

Proof. There can be only finite number of (ℓ,k)-generating sets; denote them $G_1,\dots,G_d$. It follows from Observation 4 that $S(G_1)\cup\ldots\cup S(G_d) = \mathrm{V}_k^\ell$ and, from Lemma 1, we know that $\mathcal{L}(\mathrm{V}_k^\ell) = \mathrm{Seed}_k^\ell$.

For every $i \in \{1,\dots,d\}$, the set $S(G_i)$ is a subshift of finite type, so every set $\mathcal{L}(S(G_i))$ is a regular language. Since the set Seed_k^ℓ is a union of finitely many regular languages, it is a regular language. □



(a) A graph created as a de-Bruijn graph from the set of vertices G.



(b) The previous graph after minimization.

Figure 2: Labeled graphs H for the subshift $S(G)$ in Example 4.

5 Application for seed design

In this section, we describe how to design seeds with knowledge of an (ℓ,k)-generating set. Then we show how to search $(\ell,1)$ and $(\ell,2)$-generating sets.

5.1 Seed design using generating sets

Let us have an (ℓ,k)-generating set G and let us consider a task of designing a seed Q of length s, which would solve the $(\ell+s,k)$-problem. If $s \leq \ell+1$, we can take an arbitrary factor of length s of any word from G.

If $s > \ell+1$, we need to construct the seed in $s-\ell$ steps by extending letter by letter. In the first step, we take an arbitrary word $w \in G$ and set $Q := w$. In every other step, we take any word w from G such that the last ℓ letters of Q are equal to ℓ first letters of w and concatenate the last letter of w to Q. Existence of such word w is guaranteed since we can use at least the letter - in every step.

5.2 Generating sets for $k = 1$

As a simple consequence of Corollary 1, we get a full characterization of all seeds solving $(m,1)$-problems ([2, Theorem 5]).

Proposition 2. $\mathrm{Seed}_1^\ell = \{Q \in \mathcal{A}^* \mid \#^{\ell+1}$ *is not a factor of* $Q\}$

Proof. Denote $\mathbf{v} = \cdots\text{--}|\text{-}^\ell Q\text{--}\cdots$ and $\ell = m - |Q|$. Then from Corollary 1 follows that: Q solves the $(m,1)$-problem $\iff \forall i \in \mathbb{Z}\left((\sigma^i(\mathbf{v}))[0,\ell] \neq \#^{\ell+1}\right) \iff Q$ does not contain $\#^{\ell+1}$. □

Thus, for every positive ℓ, the only $(\ell,1)$-generating set is $\mathcal{A}^{\ell+1} \setminus \{\#^{\ell+1}\}$, i.e., the set of all words of length $\ell+1$ except $\#^{\ell+1}$.

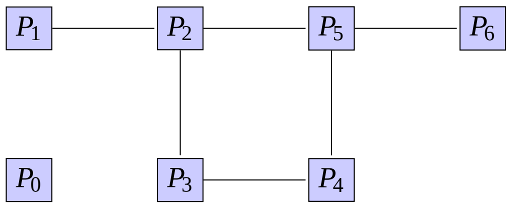


Figure 3: The simplified graph of sets of equivalent seeds for $(5,2)$-generating sets search in Example 6.

5.3 Generating sets for $k = 2$

Let $k = 2$ and ℓ be an arbitrary fixed positive integer. We can derive all $(\ell,2)$-generating sets using graph theoretical methods by transformation to independent sets search. Let $V := \{w^{(1)},\dots,w^{(q)}\}$ denote the set of all seeds of length $\ell+1$ solving the $(2\ell+1,2)$-problem. Consider a graph R given by the adjacency matrix $(M_R)_{i,j} = \begin{cases} 0 & \text{if } \mathrm{C}_2^\ell(w^{(i)},w^{(j)}), \\ 1 & \text{otherwise}. \end{cases}$

Then the generating sets are "maximal" independent sets (maximal with respect to inclusion) in the graph R. We require maximality here since it is already required by the second property in Definition 8.

We can partially simplify the graph R. We say that two vertices v and w in this graph are equivalent if $\forall x \in V \big(\mathrm{C}_k^\ell(x,v) \iff \mathrm{C}_k^\ell(x,w)\big)$. Then we can put all equivalent vertices into one vertex, i.e., every vertex will contain a set of words instead of only one word. The step with searching "maximal" independent sets stays unchanged.

Example 5. *Let $k = 2$. For every $\ell \in \{1,\dots,4\}$, all seeds solving the $(\ell+1,2)$-problem are mutually compatible, which means that there exists a unique $(\ell,2)$-generating set. We list them out in the following table.*

ℓ	G
1	{--}
2	{#--, -#-, --#, ---}
3	{#--#, #---, -#--, --#-, ---#, ---}
4	{##---, -##--, --##-, ---##, #-#--, -#-#-, --#-#, #--#-, -#--#, #---#, #----, -#---, --#--, ---#-, ----#, -----}

Example 6. *Let $k = 2$ and $\ell = 5$. We find the graph R by the procedure above. After its simplification, we obtain the graph in Figure 3, where*

P_0 = {------; -----#, ----#-, ---#--, --#---, -#----, #-----;
----##, ---##-, --##--, -##---, ##----;
---#-#, --#-#-, -#-#--, #-#---;
--#--#, -#--#-, #--#--; -#---#, #---#-;
#---##; ##---#; #----#},

P_1 = {#--#-#}, P_2 = {--##-#, -##-#-, ##-#--},

P_3 = {-##--#, ##--#-}, P_4 = {-#--##, #--##-},

P_5 = {--#-##, -#-##-, #-##--}, P_6 = {#-#--#}.

By finding "maximal" independent sets in the graph in Figure 3, we get all $(5,2)$*-generating sets:*

$$G_1 = P_0 \cup P_1 \cup P_3 \cup P_5, \qquad G_2 = P_0 \cup P_1 \cup P_3 \cup P_6,$$
$$G_3 = P_0 \cup P_2 \cup P_4 \cup P_6, \qquad G_4 = P_0 \cup P_1 \cup P_4 \cup P_6.$$

To conclude the section, let us remark that a similar derivation can be done using hypergraphs also for $k > 2$.

6 Conclusion

In this paper, we have studied lossless seeds from the perspective of symbolic dynamics. We have concentrated on the seed margin ℓ defined as a difference of the length m of compared strings and the length of a seed. We have derived asymptotic behavior of ℓ for optimal seeds (Proposition 1), which must satisfy $\ell \in \Theta(m^{\frac{k}{k+1}}) = \Theta(m - w(m))$. We have shown another criterion for errors detection by seeds (Theorem 1). From this criterion we have proved that lossless seeds coincide with languages of certain sofic subshifts, therefore, they are recognized by finite automata (Theorem 2). We have presented that these subshifts are fully given by the number of allowed errors k and the seed margin ℓ and that they can be further decomposed into subshifts of finite type.

These facts explain why periodically repeated patterns often appear in lossless seeds. This is caused by the fact that these patterns correspond to cycles in recognizing automata (which correspond to seeds for cyclic (m,k)-problems in [11]). Nevertheless, it remains unclear what is the upper bound on the length of cycles to obtain at least some optimal seeds. In the case case $k = 2$, it was conjectured in [2, Conjecture 1] that it is sufficient to consider patterns having length at most $\ell + 1$ to obtain some of optimal seeds.

Acknowledgements. The author is supported by the ABS4NGS grant of the French government (program *Investissement d'Avenir*). He is grateful to Gregory Kucherov and Karel Klouda for helpful ideas. He also thanks three anonymous referees for valuable comments.

References

[1] Karel Břinda (2013): *Laser method on-line*. Available at `http://brinda.cz/laser-method/`.

[2] Karel Břinda (2013): *Lossless seeds for approximate string matching*. Master's thesis, FNSPE Czech Technical University in Prague, Czech Republic. Available at `http://brinda.cz/publications/diplomka.pdf`.

[3] Stefan Burkhardt & Juha Kärkkäinen (2001): *Better Filtering with Gapped q-Grams*. In: *Proceedings of the 12th Symposium on Combinatorial Pattern Matching (CPM), Lecture Notes in Computer Science* 2089, Springer, pp. 73–85, doi:10.1007/3-540-48194-X_6.

[4] Stefan Burkhardt & Juha Kärkkäinen (2002): *Better filtering with gapped q-grams*. *Fundamenta Informaticae* 56(1-2), pp. 51–70.

[5] Yangho Chen, Tate Souaiaia & Ting Chen (2009): *PerM: efficient mapping of short sequencing reads with periodic full sensitive spaced seeds*. *Bioinformatics* 25(19), pp. 2514–2521, doi:10.1093/bioinformatics/btp486.

[6] Lavinia Egidi & Giovanni Manzini (2011): *Spaced Seeds Design Using Perfect Rulers*. In: *Proceedings of the 18th International Symposium on String Processing and Information Retrieval (SPIRE), Pisa (Italy), Lecture Notes in Computer Science* 7024, Springer, pp. 32–43, doi:10.1007/978-3-642-24583-1_5.

[7] Lavinia Egidi & Giovanni Manzini (2013): *Better spaced seeds using quadratic residues*. *Journal of Computer and System Sciences* 79(7), pp. 1144–1155, doi:10.1016/j.jcss.2013.03.002.

[8] Lavinia Egidi & Giovanni Manzini (2014): *Design and analysis of periodic multiple seeds*. *Theoretical Computer Science* 522, pp. 62–76, doi:10.1016/j.tcs.2013.12.007.

[9] Lavinia Egidi & Giovanni Manzini (2014): *Spaced Seeds Design Using Perfect Rulers*. *Fundamenta Informaticae* 131(2), pp. 187–203, doi:10.3233/FI-2014-1009.

[10] Martin Farach-Colton, Gad M. Landau, Süleyman Cenk Sahinalp & Dekel Tsur (2007): *Optimal spaced seeds for faster approximate string matching*. *Journal of Computer and System Sciences* 73(7), pp. 1035–1044, doi:10.1016/j.jcss.2007.03.007.

[11] Gregory Kucherov, Laurent Noé & Mikhail A. Roytberg (2005): *Multiseed lossless filtration*. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)* 2(1), pp. 51–61, doi:10.1109/tcbb.2005.12.

[12] Heng Li & Nils Homer (2010): *A survey of sequence alignment algorithms for next-generation sequencing*. *Briefings in bioinformatics* 11(5), pp. 473–83, doi:10.1093/bib/bbq015.

[13] Hao Lin, Zefeng Zhang, Michael Q. Zhang, Bin Ma & Ming Li (2008): *ZOOM! Zillions Of Oligos Mapped*. *Bioinformatics* 24(21), pp. 2431–2437, doi:10.1093/bioinformatics/btn416.

[14] Douglas Lind & Brian Marcus (1995): *An Introduction to Symbolic Dynamics and Coding*. Cambridge University Press, doi:10.1017/CBO9780511626302.

[15] M. Lothaire (1983): *Combinatorics on Words*. *Encyclopedia of Mathematics and its Applications* 17, Addison-Wesley Publishing Co., Reading, Mass., doi:10.1017/CBO9780511566097. Reprinted in the Cambridge University Press, Cambridge, UK, 1997.

[16] Bin Ma, John Tromp & Ming Li (2002): *PatternHunter: Faster and more sensitive homology search*. *Bioinformatics* 18(3), pp. 440–445, doi:10.1093/bioinformatics/18.3.440.

[17] Saul B Needleman & Christian D Wunsch (1970): *A general method applicable to the search for similarities in the amino acid sequence of two proteins*. *Journal of Molecular Biology* 48(3), pp. 443–453, doi:10.1016/0022-2836(70)90057-4.

[18] François Nicolas & Éric Rivals (2005): *Hardness of Optimal Spaced Seed Design*. In A. Apostolico, M. Crochemore & K. Park, editors: *Proceedings of the 16th Annual Symposium on Combinatorial Pattern Matching (CPM), Jeju Island (Korea)*, *Lecture Notes in Computer Science* 3537, Springer, pp. 144–155, doi:10.1007/b137128.

[19] François Nicolas & Éric Rivals (2008): *Hardness of Optimal Spaced Seed Design*. *Journal of Computer and System Sciences* 74(5), pp. 831–849, doi:10.1016/j.jcss.2007.10.001.

[20] Laurent Noé (2014): *Spaced seeds bibliography*. Available at `http://www.lifl.fr/~noe/spaced_seeds.html`.

[21] Laurent Noé & Gregory Kucherov (2005): *YASS: enhancing the sensitivity of DNA similarity search*. *Nucleic Acids Research* 33(suppl. 2), pp. W540–W543, doi:10.1093/nar/gki478.

[22] Paolo Ribeca (2012): *Short-Read Mapping*. In Naiara Rodríguez-Ezpeleta, Michael Hackenberg & Ana M. Aransay, editors: *Bioinformatics for High Throughput Sequencing*, Springer New York, pp. 107–125, doi:10.1007/978-1-4614-0782-9_7.

[23] Temple F. Smith & Michael S. Waterman (1981): *Identification of common molecular subsequences*. *Journal of molecular biology* 147(1), pp. 195–7, doi:10.1016/0022-2836(81)90087-5.

Maximally Atomic Languages[*]

Janusz Brzozowski
David R. Cheriton School of Computer Science, University of Waterloo
Waterloo, ON, Canada N2L 3G1
brzozo@uwaterloo.ca

Gareth Davies
Department of Pure Mathematics, University of Waterloo
Waterloo, ON, Canada N2L 3G1
gdavies@uwaterloo.ca

The atoms of a regular language are non-empty intersections of complemented and uncomplemented quotients of the language. Tight upper bounds on the number of atoms of a language and on the quotient complexities of atoms are known. We introduce a new class of regular languages, called the *maximally atomic languages*, consisting of all languages meeting these bounds. We prove the following result: If L is a regular language of quotient complexity n and G is the subgroup of permutations in the transition semigroup T of the minimal DFA of L, then L is maximally atomic if and only if G is transitive on k-subsets of $\{1,\dots,n\}$ for $0 \le k \le n$ and T contains a transformation of rank $n-1$.

Keywords: atom, átomaton, finite automaton, quotient complexity, regular language, set-transitive group, state complexity, transition semigroup

1 Introduction

The *state/quotient complexity* of a regular language is the number of states in the minimal deterministic finite automaton (DFA) of the language, or equivalently, the number of left quotients of the language. An *atom* of a regular language is a non-empty intersection of the language's left quotients, some of which may be complemented. Brzozowski and Tamm have found tight upper bounds on the number of atoms of a language [4] and on the quotient complexities of atoms [3]. This lets us define a new class of regular languages which we call *maximally atomic*: these are regular languages whose atoms meet these bounds.

The *transition semigroup* of a DFA is the semigroup of transformations induced by the transition function of the DFA on its set of states. Our main result (stated formally in Section 3) is the following relationship between maximally atomic languages and transition semigroups:

A regular language with quotient complexity n is maximally atomic if and only if the transition semigroup of its minimal DFA contains permutations that can map any subset of $\{1,\dots,n\}$ to any other subset of the same size, as well as at least one transformation with an image of size $n-1$.

In the process of proving this, we establish several other relationships between transition semigroups and atoms; in particular, we give sufficient conditions for a language to have the maximal number of atoms, and necessary and sufficient conditions for certain individual atoms to have maximal complexity. We also derive a general formula for the transition functions of "átomata" (nondeterministic finite automata whose states correspond to the atoms of the language they recognize).

[*]This work was supported by the Natural Sciences and Engineering Research Council of Canada under grant No. OGP0000871.

2 Definitions and Terminology

2.1 Partially Ordered Sets

A *partially ordered set* (poset) is a pair $(S,\leq)$ where S is a set and $\leq$ is a partial order on S. A *subposet* of $(S,\leq)$ is a poset $(T,\leq)$ such that $T\subseteq S$. We often abbreviate $(S,\leq)$ to simply S.

If T is a subposet of S, then for $a,b\in S$, the *interval* of T between a and b, denoted $[a,b]_T$, is the set of all $t\in T$ such that $a\leq t$ and $t\leq b$. Note that if $b<a$, then the interval $[a,b]_T$ is empty.

Let $Q_n=\{1,2,\ldots,n\}$, let $P=(2^{Q_n},\subseteq)$, and let X be a subposet of P. For each non-empty interval $[V,U]_X$, define the *type* of $[V,U]_X$ to be the pair of integers $(|\bigcap[V,U]_X|,|\bigcup[V,U]_X|)$. Let the type of the empty interval be $(-1,-1)$.

2.2 Transformations

A *transformation* of a set X is a mapping $t\colon X\to X$. Since we deal only with finite sets, we assume without loss of generality that $X=Q_n$ for some n. A *permutation* is an invertible (one-to-one and onto) transformation. A *singular transformation* is a non-invertible transformation.

If a transformation t maps i to j, we say the *image* of i under t is j and write $t(i)=j$. The image of $S\subseteq Q_n$ is $t(S)=\{t(i)\mid i\in S\}$. The image of t itself is $\operatorname{im} t=t(Q_n)$. The *coimage* of t is $\operatorname{coim} t=\overline{\operatorname{im} t}$, where $\overline{S}=Q_n\setminus S$. The *preimage* of an element i under t is $t^{-1}(i)=\{j\mid t(j)=i\}$. The preimage of $S\subseteq Q_n$ under t is $t^{-1}(S)=\bigcup_{i\in S}t^{-1}(i)$. The *rank* of a transformation is $|\operatorname{im} t|$. The *composition* or *product* of two transformations s and t is $s\circ t$, defined by $(s\circ t)(i)=s(t(i))$.

A *transposition* (i,j) for $i\neq j$ is a transformation such that $t(i)=j$, $t(j)=i$, and $t(\ell)=\ell$ for all $\ell\notin\{i,j\}$. A permutation is *even* if it can be written as a product of an even number of transpositions and it is *odd* otherwise. A *unitary transformation*, denoted by $(i\to j)$ (with $i\neq j$), is a transformation such that $t(i)=j$ and $t(\ell)=\ell$ for all $\ell\neq i$.

2.3 Semigroups, Monoids, and Groups

A *semigroup* is a pair $(S,\cdot)$, where S is a non-empty set and $\cdot$ is an associative binary operation. We often abbreviate $(S,\cdot)$ to S. A *monoid* $M=(M,\cdot,e)$ is a semigroup with identity e, and a *group* $G=(G,\cdot,e)$ is a monoid in which each element has an inverse. A *subsemigroup* of $(S,\cdot)$ is a semigroup $(T,\cdot)$ where $T\subseteq S$. If $(S,\cdot,e)$ and $(M,\cdot,e)$ are monoids with $M\subseteq S$, then M is a *submonoid* of S. A *subgroup* of S is a submonoid G of S such that G is a group.

The *full transformation semigroup* of degree n, denoted T_n, is the set of all transformations $t\colon Q_n\to Q_n$ under the binary operation $\circ$. Note that T_n is a monoid, since the identity transformation of Q_n acts as the identity element. The *symmetric group* of degree n, denoted by S_n, is the subgroup of permutations in T_n. A *transformation semigroup* of degree n is a subsemigroup of T_n, and a *permutation group* of degree n is a subgroup of S_n. A *conjugate* of a permutation group G of degree n is a group of the form $\{p\circ g\circ p^{-1}\mid g\in G\}$, where $p\in S_n$.

Let G be a permutation group of degree n and let X be a set. For $x\in X$, the *orbit* of x under G is the set $\{g(x)\mid g\in G\}$. We say that G *acts transitively* or *is transitive* on a set X if for all $x,y\in X$ there exists $g\in G$ such that $g(x)=y$, or equivalently, if G has only one orbit when it acts on X. We say G is *k-set-transitive* if it is transitive on the set of k-subsets (subsets of cardinality k) of Q_n. If G is k-set-transitive for $0\leq k\leq n$, we say G is *set-transitive*.

The set-transitive permutation groups have been fully classified by Beaumont and Peterson [1]. In general, a set-transitive group is either the symmetric group S_n or the alternating group A_n (the subgroup of even permutations in S_n). When n is small there are four exceptions (up to conjugation):

Proposition 1. *A set-transitive permutation group of degree n is S_n or A_n or a conjugate of one of the following permutation groups:*

1. *For $n = 5$, the affine general linear group* $\mathrm{AGL}(1,5)$.
2. *For $n = 6$, the projective general linear group* $\mathrm{PGL}(2,5)$.
3. *For $n = 9$, the projective special linear group* $\mathrm{PSL}(2,8)$.
4. *For $n = 9$, the projective semilinear group* $\mathrm{P\Gamma L}(2,8)$.

2.4 Finite Automata

A *nondeterministic finite automaton* (*NFA*) is a tuple $\mathcal{N} = (Q,\Sigma,\eta,I,F)$, where Q is a finite, non-empty set of *states*, Σ is a finite, non-empty *alphabet*, $\eta\colon Q \times \Sigma \to 2^Q$ is a *transition function*, $I \subseteq Q$ is a set of *initial* states, and $F \subseteq Q$ is a set of *final* states. We extend η to $\eta\colon 2^Q \times \Sigma^* \to 2^Q$ as follows: for $S \subseteq Q$ and $w = xa$, $x \in \Sigma^*$, $a \in \Sigma$, we define $\eta(S,w)$ inductively by $\eta(S,\varepsilon) = S$ and $\eta(S,xa) = \eta(\eta(S,x),a) = \bigcup_{s\in\eta(S,x)} \eta(s,a)$. We define $\eta_w\colon 2^Q \to 2^Q$ by $\eta_w(S) = \eta(S,w)$.

A word w is *accepted* by $\mathcal{N}$ if $\eta_w(I) \cap F \neq \emptyset$. The *language accepted by* $\mathcal{N}$ is the set of all words accepted by $\mathcal{N}$. The *language of a state* $q \in Q$ is the language accepted by the modified NFA $\mathcal{N}_q = (Q,\Sigma,\eta,\{q\},F)$. For $S,T \subseteq Q$, we say S is *reachable from* T in $\mathcal{N}$ if there exists $w \in \Sigma^*$ such that $\eta_w(T) = S$. If S is reachable from I, we simply say S is *reachable* in $\mathcal{N}$. An NFA that accepts a language L is *minimal* if the number of states is minimal among all NFAs that accept L.

A *deterministic finite automaton* (*DFA*) is a tuple $\mathcal{D} = (Q,\Sigma,\delta,q_1,F)$, where Q, Σ and F have the same meaning as in an NFA, $\delta\colon Q \times \Sigma \to Q$ is a transition function, and $q_1 \in Q$ is an initial state. Since DFAs are special cases of NFAs, all the definitions above apply also to DFAs. While minimal NFAs need not be unique, there is a unique (up to isomorphism) minimal DFA for each regular language.

For all $w \in \Sigma^*$, $\delta_w\colon Q \to Q$ is a transformation of the set of states of $\mathcal{D}$; we call this the *transformation induced by* w in $\mathcal{D}$. The *transition semigroup* of $\mathcal{D}$ is the semigroup $(T,\circ)$, where $T = \{\delta_w \mid w \in \Sigma^+\}$. This is the semigroup of transformations of Q induced by non-empty words over Σ in $\mathcal{D}$.

For an NFA $\mathcal{N} = (Q,\Sigma,\eta,I,F)$, define the *reverse* of $\mathcal{N}$ to be the NFA $\mathcal{N}^R = (Q,\Sigma,\eta^R,F,I)$, where $\eta^R(q,a) = \{p \in Q \mid q \in \eta(p,a)\}$. Note that if $\mathcal{N} = \mathcal{D}$ is a DFA with transition function δ, then δ_w is a transformation and we have $\delta_w^R = \delta_w^{-1}$. Define the *determinization* of an NFA $\mathcal{N}$ to be the DFA $\mathcal{N}^D = (Q',\Sigma,\eta^D,I,F')$, where $Q' = \{S \in 2^Q \mid S \text{ is reachable in } \mathcal{N}\}$, $F' = \{S \in Q' \mid S \cap F \neq \emptyset\}$, and $\eta^D(S,a) = \bigcup_{s\in S} \eta(s,a)$.

2.5 Languages, Quotients, and Atoms

Let L be a regular language over the alphabet Σ and let $\mathcal{D} = (Q_n,\Sigma,\delta,q_1,F)$ be the minimal DFA of L. The *left quotient* (or simply *quotient*) of L by the word $w \in \Sigma^*$ is $w^{-1}L = \{x \mid wx \in L\}$. There is a one-to-one correspondence between quotients of L and states of the minimal DFA of L: the languages of distinct states of $\mathcal{D}$ are distinct quotients of L. We use the following convention when discussing quotients of L: the set of quotients is $\{K_1,K_2,\dots,K_n\}$, where K_i is the language of state i of $\mathcal{D}$. Due to the one-to-one correspondence between states and quotients, the *complexity* of L can be equivalently defined as the number of states in the minimal DFA of L (*state complexity*) or the number of distinct quotients of L (*quotient complexity*).

From now on we deal with non-empty languages only. Denote the complement of a language L by $\overline{L} = \Sigma^* \setminus L$. For $S \subseteq Q_n$, let A_S denote the intersection $\bigcap_{i \in S} K_i \cap \bigcap_{i \in \overline{S}} \overline{K_i}$. If A_S is non-empty, then A_S is called an *atom* of L. Let $\mathbf{A}$ be the set of all atoms of L. The *atom map* $\phi \colon \mathbf{A} \to 2^{Q_n}$ is defined by $\phi(A_S) = S$. This map is well-defined, since for each atom A there is precisely one subset S of Q_n such that $A_S = A$. The *basis* of an atom A is $\mathscr{B}(A) = \{K_i \mid i \in \phi(A)\}$.

The *átomaton* of L is the NFA $\mathscr{A} = (\mathbf{A}, \Sigma, \eta, I, F)$, where $\eta(A_i, a) = \{A_j \mid aA_j \subseteq A_i\}$, $I = \{A \in \mathbf{A} \mid q_1 \in \phi(A)\}$, and $F = \{A \in \mathbf{A} \mid \varepsilon \in A\}$. Note that the initial atoms are those that contain L in their bases. Also, there is precisely one final atom: the atom for which all the quotients in its basis contain ε and all other quotients do not. The language of state A of $\mathscr{A}$ is the atom A [4].

The *atomic poset* of L is $\phi(\mathbf{A}) = (\phi(\mathbf{A}), \subseteq)$; this is the set of all subsets S of Q_n such that A_S is an atom. An *atomic interval* of L is an interval in $\phi(\mathbf{A})$, that is, an interval of the form $[V, U]_{\phi(\mathbf{A})}$. We denote an atomic interval using double brackets, since this makes the notation cleaner: we write $[[V, U]]$ instead of $[V, U]_{\phi(\mathbf{A})}$. Since $\phi(\mathbf{A})$ is a subposet of $(2^{Q_n}, \subseteq)$, any two subsets of Q_n can act as endpoints of an atomic interval. Furthermore, every atomic interval $[[V, U]]$ has an associated type (v, u), as defined in the section on posets.

Note that, if $[[V, U]]$ contains both of its endpoints (i.e., $V, U \in [[V, U]]$), then the type of $[[V, U]]$ is $(|V|, |U|)$. However, we cannot always use the sizes of the endpoints to determine the type of an interval, since there may be multiple ways to choose the endpoints of an interval. For example, if $A_{\{1\}}$ is an atom but $A_\emptyset$ and $A_{\{1,2\}}$ are not, then $[[\{1\}, \{1\}]] = [[\emptyset, \{1\}]] = [[\{1\}, \{1,2\}]] = \{\{1\}\}$. But this interval has type $(1, 1)$, not $(0, 1)$ or $(1, 2)$.

Some basic facts about atoms and átomata follow. The following proposition, proved in [3], shows that we may view the states of $\mathscr{A}$ as subsets of Q_n:

Proposition 2. *Let L be a regular language with átomaton $\mathscr{A}$ and minimal DFA $\mathscr{D}$. Then the atom map ϕ is an NFA isomorphism between $\mathscr{A}$ and $\mathscr{D}^{RDR}$.*

The next proposition relates the number of atoms of L to the complexity of the reverse L^R. The proof follows easily from Proposition 2.

Proposition 3 (Number of Atoms)**.** *Let L be a regular language with complexity n, and let the minimal DFA of L be $\mathscr{D} = (Q_n, \Sigma, \delta, q_1, F)$. Then for $S \subseteq Q_n$, the intersection A_S is an atom of L if and only if S is reachable in $\mathscr{D}^R$, i.e, if and only if there exists $w \in \Sigma^*$ such that $\delta_w^{-1}(F) = S$. Thus there is a bijection between atoms of L and states of $\mathscr{D}^{RD}$, the minimal DFA of L^R.*

It is well-known that if the complexity of L is n, then the complexity of L^R is at most 2^n, and for $n \geq 2$ this bound is tight. Thus 2^n is also a tight bound on the number of atoms of a regular language when $n \geq 2$.

In [3], a tight upper bound on the complexity of individual atoms was derived and a formula for the bound was given. We give a different (but equivalent) formula below:

Proposition 4 (Complexity of Atoms)**.** *Let L be a regular language with complexity n. Define the function Ψ as follows:*

$$\Psi(n,k) = \begin{cases} 2^n - 1, & \text{if } k = 0 \text{ or } k = n; \\ 1 + \sum_{v=1}^{k} \sum_{u=k}^{n-1} \binom{n}{u} \binom{u}{v}, & \text{if } 1 \leq k \leq n-1. \end{cases}$$

If A_S is an atom of L, $\Psi(n, |S|)$ is a tight upper bound on the complexity of A_S.

With these bounds established, we can formally define the class of maximally atomic languages. A non-empty regular language L of complexity n is *maximally atomic* if it has the maximal number of atoms (1 if $n = 1$, 2^n if $n \geq 2$) and if for each atom A_S of L, A_S has the maximal complexity $\Psi(n, |S|)$.

3 Main Results

Note that when $n = 1$, the only nonempty language over Σ is Σ^*, and it is maximally atomic. The following proposition characterizes the maximally atomic languages of complexity $n = 2$:

Proposition 5. *Let L be a regular language of complexity 2 and let $\mathscr{D}$ be its minimal DFA with state set Q_2. Let T be the transition semigroup of $\mathscr{D}$. Then:*

- *There are four transformations of Q_2: the identity transformation, the transposition $(1,2)$, and the unitary transformations $(1 \to 2)$ and $(2 \to 1)$.*
- *T contains all four transformations of Q_2 if and only if T contains $(1,2)$ and at least one unitary transformation.*
- *All subsets of Q_2 are reachable in $\mathscr{D}^R$ (and hence L has all 2^2 atoms) if and only if T contains all four transformations of Q_2.*
- *Each atom of L has maximal complexity if and only if T contains all four transformations of Q_2.*
- *Thus, L is maximally atomic if and only if T contains all four transformations of Q_2.*

The computations required to prove this proposition can be easily done by hand. Henceforth we will be concerned only with languages of complexity $n \geq 3$.

Our main theorem is the following:

Theorem 1. *Let L be a regular language over Σ with complexity $n \geq 3$, and let T be the transition semigroup of the minimal DFA of L. Then L is maximally atomic if and only if the subgroup of permutations in T is set-transitive and T contains a transformation of rank $n-1$.*

In view of this, let us consider how the class of maximally atomic languages relates to other language classes. Let **FTS** denote the class of languages whose minimal DFAs have the *full transformation semigroup* as their transition semigroup, let **STS** denote the class whose minimal DFAs have transition semigroups with a *set-transitive subgroup* of permutations and a transformation of rank $n-1$, let **MAL** denote the class of *maximally atomic languages*, let **MNA** denote the class of languages with the *maximal number of atoms*, and let **MCR** denote the class of languages with a *maximally complex reverse*.

1. **FTS** is properly contained in **STS**, by Proposition 1.
2. **STS** is equal to **MAL**, by Theorem 1.
3. **MAL** is contained in **MNA**. Figure 1 in [2] shows the containment is proper.
4. **MNA** is equal to **MCR**, by Proposition 3.

To summarize, we have: **FTS** $\subset$ **STS** = **MAL** $\subset$ **MNA** = **MCR**.

The proof of Theorem 1 relies on two intermediate results. The first gives a condition that is sufficient (but not necessary) for L to have 2^n atoms:

Theorem 2. *Let L be a regular language over Σ with complexity $n \geq 3$, and let T be the transition semigroup of the minimal DFA of L. If T contains all unitary transformations, then L has 2^n atoms.*

The second result establishes Theorem 1 in all but a few cases; it gives necessary and sufficient conditions for individual atoms of L to have maximal complexity, but only when the bases of the atoms are in a certain size range.

Theorem 3. *Let L be a regular language over Σ with complexity $n \geq 3$, and let T be the transition semigroup of the minimal DFA of L. Let A_S be an atom of L and suppose that either $n \geq 4$ and $2 \leq |S| \leq n-2$, or $n = 3$ and $1 \leq |S| \leq 2$. Then A_S has maximal complexity if and only if the subgroup of permutations in T is $|S|$-set-transitive and T contains a transformation of rank $n-1$.*

The rest of the paper consists of the proofs of these three theorems. Shortly before the deadline for this paper, we were informed that the proof of our main result can be simplified by replacing the átomaton with a different construction [5]. Below we present our original proofs, which use the átomaton.

4 Proof of Theorem 2

Let L be a language of complexity $n \geq 3$ and let $\mathscr{D} = (Q_n, \Sigma, \delta, q_1, F)$ be its minimal DFA. Let T be the transition semigroup of $\mathscr{D}$ and assume it contains all unitary transformations. By Proposition 3, L has 2^n atoms if and only if for all $S \subseteq Q_n$, S is reachable in $\mathscr{D}^R$, i.e., there exists $w \in \Sigma^*$ such that $\delta_w^R(F) = \delta_w^{-1}(F) = S$.

Suppose $X \subseteq Q_n$, with $1 \leq |X| \leq n-1$. Let $t = (i \to j)$ and $s = (i \to k)$ for $i \in Q_n$, $j \in X$, $k \not\in X$; then $t^{-1}(X) = X \cup \{i\}$ and $s^{-1}(X) = X \setminus \{i\}$. Since T contains all unitary transformations, it contains t and s. Thus for every non-empty $X \subset Q_n$ and every $i \in Q_n$, there are words $w, x \in \Sigma^*$ such that $\delta_w^{-1}(X) = X \cup \{i\}$ and $\delta_x^{-1}(X) = X \setminus \{i\}$.

In other words, from any non-empty proper subset X of Q_n, we can reach (in $\mathscr{D}^R$) all subsets that differ from X by the addition or removal of a single element. Repeatedly applying this fact, we see that from X we can reach any subset S of Q_n: shrink X to a singleton $\{i\} \subseteq X$, expand $\{i\}$ to $\{i, j\}$ for $j \in S$, shrink again to $\{j\} \subseteq S$, and then expand to S (or shrink to $\emptyset$ for $S = \emptyset$).

Now, if $|F| = 0$ then $L = \emptyset$, and if $|F| = n$ then $L = \Sigma^*$; since $\mathscr{D}$ is minimal, $n = 1$ in either case. Since $n \geq 3$, we have that F is a non-empty proper subset of Q_n. Thus by the argument above, we can reach all subsets of Q_n in $\mathscr{D}^R$; hence L has 2^n atoms. □

5 Proof of Theorem 3

5.1 The Átomaton and Minimal DFAs of Atoms

In this section we prove the $\Rightarrow$ direction of Theorem 3. Two results on átomata and atoms are needed for this. We first describe the transition function of the átomaton, in the case where the states are viewed as subsets of Q_n. Define $\Delta_w \colon 2^{Q_n} \to 2^{Q_n}$ by $\Delta_w(S) = \overline{\delta_w(\overline{S})} = Q_n \setminus \delta_w(Q_n \setminus S)$.

Lemma 1. *Let L be a regular language over Σ. Let $\mathscr{D} = (Q_n, \Sigma, \delta, q_1, F)$ be the minimal DFA of L. Let $\mathscr{A}$ be the átomaton of L with transition function η. If $[[V,U]]$ is an atomic interval of L and a set of states of $\mathscr{A}$, then for all $w \in \Sigma^*$, we have $\eta_w([[V,U]]) = [[\delta_w(V), \Delta_w(U)]]$.*

Proof. It was shown in [3] that $\eta_a(S) = \{T \mid A_T$ is an atom of L, $T \supseteq \delta_a(S)$ and $\delta_a(\overline{S}) \cap T = \emptyset\}$. If $\delta_a(\overline{S}) \cap T = \emptyset$, then $T \subseteq \overline{\delta_a(\overline{S})} = \Delta_a(S)$. Thus $\eta_a(S)$ is the set of $T \subseteq Q_n$ such that A_T is an atom of L and $\delta_a(S) \subseteq T \subseteq \Delta_a(S)$, which is precisely $[[\delta_a(S), \Delta_a(S)]]$. One verifies that this can be extended to words, giving $\eta_w(S) = [[\delta_w(S), \Delta_w(S)]]$.

Next, we want to show $\eta_w([[V,U]]) = [[\delta_w(V), \Delta_w(U)]]$. For $T \in [[V,U]]$, consider $\eta_w(T)$. Since $V \subseteq T$, $\delta_w(V) \subseteq \delta_w(T)$. Since $T \subseteq U$, we have $\overline{T} \supseteq \overline{U}$, and thus $\delta_w(\overline{T}) \supseteq \delta_w(\overline{U})$. It follows that $\Delta_w(T) \subseteq \Delta_w(U)$. Hence $\eta_w(T) = [[\delta_w(T), \Delta_w(T)]] \subseteq [[\delta_w(V), \Delta_w(U)]]$, and $\eta_w([[V,U]]) \subseteq [[\delta_w(V), \Delta_w(U)]]$.

For containment in the other direction, suppose that T is in $[[\delta_w(V), \Delta_w(U)]]$; then $\delta_w(V) \subseteq T \subseteq \Delta_w(U)$ and A_T is an atom. Let $S = \delta_w^{-1}(T)$; then we claim $S \in [[V,U]]$. Since $T \supseteq \delta_w(V)$, we have $\delta_w^{-1}(T) = S \supseteq V$. If $i \in T \subseteq \Delta_w(U)$, then $i \not\in \delta_w(\overline{U})$. Hence $\delta_w^{-1}(i)$ is disjoint from $\overline{U}$ for all $i \in T$, and so $\delta_w^{-1}(T)$ is disjoint from $\overline{U}$. It follows that $\delta_w^{-1}(T) = S \subseteq U$. It remains to show A_S is an atom; but since A_T is an atom, by Proposition 3, there exists $x \in \Sigma^*$ such that $\delta_x^{-1}(F) = T$. Thus $S = \delta_w^{-1}(T) = \delta_w^{-1}(\delta_x^{-1}(F)) = \delta_{xw}^{-1}(F)$, so by Proposition 3, A_S is also an atom.

Hence $S \in [[V,U]]$, and it follows that $\eta_w(S) = [[T,\Delta_w(S)]] \subseteq \eta_w([[V,U]])$. To complete the proof, we must show $\eta_w(S)$ is non-empty (and thus contains T) by showing that $T \subseteq \Delta_w(S) = \overline{\delta_w(\overline{\delta_w^{-1}(T)})}$. Observe that if $i \in T$, then $\delta_w^{-1}(i) \subseteq \delta_w^{-1}(T)$. Thus $\delta_w^{-1}(i) \cap \overline{\delta_w^{-1}(T)} = \emptyset$, and so $i \notin \delta_w(\overline{\delta_w^{-1}(T)})$, which gives $i \in \Delta_w(S)$ as required. Thus $T \in \eta_w(S) = [[T,\Delta_w(S)]]$, and it follows that if $T \in [[\delta_w(V),\Delta_w(U)]]$, then $T \in \eta_w([[V,U]])$. This proves that the two intervals must be equal. $\square$

Table 1: $\mathscr{D}$.

	δ	a
$\rightarrow$	1	2
$\leftarrow$	2	3
$\leftarrow$	3	4
	4	4

Table 2: $\mathscr{D}^R$.

	δ^R	a
$\leftarrow$	1	
$\rightarrow$	2	$\{1\}$
$\rightarrow$	3	$\{2\}$
	4	$\{3,4\}$

Table 3: $\mathscr{D}^{RD}$.

	δ^{RD}	a
$\rightarrow$	$\{2,3\}$	$\{1,2\}$
$\leftarrow$	$\{1,2\}$	$\{1\}$
$\leftarrow$	$\{1\}$	$\emptyset$
	$\emptyset$	$\emptyset$

Table 4: $\mathscr{A}$.

	η	a
$\leftarrow$	$\{2,3\}$	
$\rightarrow$	$\{1,2\}$	$\{\{2,3\}\}$
$\rightarrow$	$\{1\}$	$\{\{1,2\}\}$
	$\emptyset$	$\{\emptyset,\{1\}\}$

Example 1. The minimal DFA $\mathscr{D}$ of Table 1 accepts the language $\{a,aa\}$. The NFA $\mathscr{D}^R$ is in Table 2 and the DFA $\mathscr{D}^{RD}$, in Table 3. The átomaton $\mathscr{A}$ is in Table 4. In NFAs $\mathscr{D}^R$ and $\mathscr{A}$, a blank in an entry (q,a) indicates that there is no transition from q under a. However, when determinization is used in Table 3, the empty set of states of $\mathscr{D}^R$ becomes a state of the resulting DFA $\mathscr{D}^{RD}$. A right arrow ($\rightarrow$) indicates an initial state and a left arrow ($\leftarrow$) indicates a final state.

Consider the atomic interval $[[\emptyset,\{1,2\}]] = \{\emptyset,\{1\},\{1,2\}\}$; we have $\delta_a(\emptyset) = \emptyset$, and $\Delta_a(\{1,2\}) = \overline{\delta_a(\overline{\{1,2\}})} = \overline{\delta_a(\{3,4\})} = \overline{\{4\}} = \{1,2,3\}$. Thus to determine the result of $\eta_a([[\emptyset,\{1,2\}]])$, we take the interval $[\emptyset,\{1,2,3\}]_{2^{Q_4}} = \{\emptyset,\{1\},\{2\},\{3\},\{1,2\},\{1,3\},\{2,3\},\{1,2,3\}\}$ and we remove the sets that do not represent atoms. After this removal, we get $\{\emptyset,\{1\},\{1,2\},\{2,3\}\}$. Hence $\eta_a(\{\emptyset,\{1\},\{1,2\}\}) = \eta_a([[\emptyset,\{1,2\}]]) = [[\emptyset,\{1,2,3\}]] = \{\emptyset,\{1\},\{1,2\},\{2,3\}\}$. ■

Remark 1. If we treat the set of states of $\mathscr{A}$ as a subset of 2^{Q_n}, then it is possible that the empty set is a state of $\mathscr{A}$, as in Example 1. Since we use the same symbol for η and its extension to subsets of states, an ambiguity arises when η is applied to the empty set. Specifically, $\eta_w(\emptyset)$ may mean "η_w applied to the state $\emptyset \in 2^{Q_n}$", in which case $\eta_w(\emptyset) = \eta_w([[\emptyset,\emptyset]]) = [[\emptyset, \operatorname{coim}\delta_w]]$, or it may mean "$\eta_w$ applied to the empty subset of states $\emptyset \subseteq 2^{Q_n}$", in which case $\eta_w(\emptyset) = \emptyset$. We avoid this ambiguity by adopting the convention that $\eta_w(\emptyset)$ always means "η_w applied to $\emptyset \subseteq 2^{Q_n}$" and $\eta_w([[\emptyset,\emptyset]])$ has the other meaning.

A corollary of this is that every reachable subset of states in the átomaton $\mathscr{A} = (\mathbf{A},\Sigma,\eta,I,F)$ is an atomic interval of L. The same holds for every reachable subset of states in the NFA $\mathscr{A}_S = (\mathbf{A},\Sigma,\eta,\{S\},F)$ recognizing the atom A_S. Since the determinization $\mathscr{A}_S^D$ is the minimal DFA of A_S [3], it follows that *the states of minimal DFAs of atoms of L may be represented as atomic intervals of L.*

If A_S is an atom of L with maximal complexity, certain restrictions apply to the *types* of the atomic intervals in $\mathscr{A}_S^D$. For $S \subseteq Q_n$, define an *S-type* to be a pair of integers (v,u) satisfying:

1. If $|S| = 0$, then $v = 0$ and $0 \le u \le n-1$.
2. If $|S| = n$, then $1 \le v \le n$ and $u = n$.
3. If $1 \le |S| \le n-1$, then $1 \le v \le |S|$ and $|S| \le u \le n-1$.

A non-empty interval that has an S-type is called an *S-interval*. The empty interval is a special case: it is an S-interval if and only if $1 \le |S| \le n-1$. The significance of S-types and S-intervals is as follows:

Lemma 2. *Let L be a regular language with complexity n and let A_S be an atom of L. If A_S has maximal complexity $\Psi(n,|S|)$, then the set of states of $\mathscr{A}_S^D$ equals the set of atomic S-intervals of L.*

Proof. A simple counting argument shows that the number of intervals of type (v,u) in a subposet of $P = (2^{Q_n}, \subseteq)$ is bounded from above by $\binom{n}{u}\binom{u}{v}$. Combining this fact with the definition of an S-type gives that $\Psi(n,|S|)$ is an upper bound on the number of S-intervals in a subposet of P. Now, we know that $\mathscr{A}_S^D$ has exactly $\Psi(n,|S|)$ states; if we show that these states are all atomic S-intervals of L, then this implies the state set of $\mathscr{A}_S^D$ contains exactly $\Psi(n,|S|)$ distinct atomic S-intervals of L, and nothing else. Since the atomic poset of L is a subposet of P, there can be no more than $\Psi(n,|S|)$ atomic S-intervals of L, and this proves the result. Thus we just need to show that every state of $\mathscr{A}_S^D$ is an S-interval.

Let $[[V,U]]$ be a state of $\mathscr{A}_S^D$ and suppose $[[\delta_w(S),\Delta_w(S)]] = [[V,U]]$. If $[[V,U]]$ is the empty interval, then it is automatically an S-interval for $1 \le |S| \le n-1$, by definition. For $|S| = 0$ or $|S| = n$, the empty interval is not an S-interval, but this does not matter since it is not reachable in $\mathscr{A}_S^D$. In the $|S| = 0$ case, a state of $\mathscr{A}_S^D$ has the form $[[\delta_w(\emptyset),\Delta_w(\emptyset)]] = [[\emptyset, \operatorname{coim}\delta_w]]$, which always contains $\emptyset$ (since $A_S = A_\emptyset$ is an atom); thus every state of $\mathscr{A}_S^D$ is a non-empty interval. For $|S| = n$, a similar argument works.

Next, suppose $[[V,U]]$ is non-empty. For this case, some setup is needed. Define the set $X_S = \{[\delta_w(S),\Delta_w(S)]_P \mid w \in \Sigma^*\}$ of intervals in P. One can verify that $(|\delta_w(S)|,|\Delta_w(S)|)$ is an S-type for all S and w, and thus X_S is a set of S-intervals of P. This means $|X_S|$ is bounded from above by $\Psi(n,|S|)$. Now, let $Y_S = \{[[\delta_w(S),\Delta_w(S)]] \mid w \in \Sigma^*\}$; this is just the set of states of $\mathscr{A}_S^D$, and thus it has size $\Psi(n,|S|)$. Define $\alpha\colon X_S \to Y_S$ by $\alpha([X,Y]_P) = [[X,Y]]$; this is clearly a surjection, and thus $|X_S| \ge |Y_S| = \Psi(n,|S|)$. Since we also have $|X_S| \le \Psi(n,|S|)$, we get $|X_S| = |Y_S| = \Psi(n,|S|)$ and hence α is a bijection.

Now, assume without loss of generality that the type of $[[V,U]]$ is $(|V|,|U|)$. Suppose for a contradiction that $|V| > |S|$. Then since $[[V,U]] = [[\delta_w(S),\Delta_w(S)]]$, we have $\delta_w(S) \subset V$. We can find a set X such that $|X| = |S|$ and $\delta_w(S) \subseteq X \subset V$. Now, since $\delta_w(S) \subseteq X \subseteq \Delta_w(S)$, we have $X \in [[\delta_w(S),\Delta_w(S)]]$ if and only if A_X is an atom. But $X \notin [[V,U]]$ since $X \subset V$, and thus A_X is not an atom. It follows that the interval $[[X,X]]$ is empty. If $|S| = 0$ or $|S| = n$, then in fact $|S| = |X|$ and $[[S,S]]$ is clearly non-empty, a contradiction. If $1 \le |S| \le n-1$, observe that $\alpha([X,X]_P) = [[X,X]] = \emptyset$. But $\emptyset \in X_S$ since $\emptyset$ is an S-interval of P for $1 \le |S| \le n-1$, so also $\alpha(\emptyset) = \emptyset$. This is a contradiction, since α is a bijection. Thus for S of any size, we always have $|V| \le |S|$. A similar argument to the above shows that $|U| \ge |S|$.

Thus, if $|S| = 0$ we have $|V| = 0$ and $0 \le |U| \le |\Delta_w(S)| = n-1$. If $|S| = n$ we have $1 \le |\delta_w(S)| \le |V| \le n$ and $|U| = n$. If $1 \le |S| \le n-1$, then $1 \le |V| \le |S|$ and $|S| \le |U| \le n-1$. Thus we have proved $(|V|,|U|)$ is an S-type. Hence every state $[[V,U]]$ of $\mathscr{A}_S^D$ is an atomic S-interval of L, and the number of states equals the upper bound $\Psi(n,|S|)$ on the number of atomic S-intervals of L, proving the lemma. □

Lemma 2 has two particularly useful consequences. Let A_S be an atom of maximal complexity, and suppose $V,U \subseteq Q_n$ are sets such that $(|V|,|U|)$ is an S-type. Then:

1. $[[V,U]]$ has type $(|V|,|U|)$. In particular, this means $[[V,U]]$ contains its endpoints V and U.
2. $[[V,U]]$ is a state of $\mathscr{A}_S^D$.

(1) follows since $(|V|,|U|)$ is an S-type, and so if $[[V,U]]$ does not have type $(|V|,|U|)$, the number of atomic S-intervals of type $(|V|,|U|)$ is not maximal and hence A_S is not maximally complex. (2) follows since if $[[V,U]]$ has the S-type $(|V|,|U|)$, it is an atomic S-interval and thus a state of $\mathscr{A}_S^D$.

These facts are sufficient to prove one direction of Theorem 3:

Theorem 3 ($\Rightarrow$ Direction). Let L be a language of complexity $n \geq 3$, let T be the transition semigroup of the minimal DFA of L, and let A_S be an atom of L. Suppose either $n = 3$ and $1 \leq |S| \leq 2$, or $n \geq 4$ and $2 \leq |S| \leq n-2$. We prove that if A_S has maximal complexity, then the subgroup of permutations in T is $|S|$-set-transitive and T contains a transformation of rank $n-1$.

The minimal DFA of A_S is $\mathscr{A}_S^D$, and its initial state is $[[S,S]]$. For all $X \subseteq Q_n$ with $|X| = |S|$, $(|X|,|X|)$ is an S-type. Thus by Lemma 2, $[[X,X]]$ is a state of $\mathscr{A}_S^D$ of type $(|X|,|X|)$. Thus $\eta_w([[S,S]]) = [[\delta_w(S),\Delta_w(S)]] = [[X,X]]$ for some $w \in \Sigma^*$. Applying Lemma 2 again gives $(|\delta_w(S)|,|\Delta_w(S)|) = (|X|,|X|)$. Hence $|X| = |\delta_w(S)| = |S|$, and so $\delta_w \in T$ is a permutation. It follows for all $X \subseteq Q_n$ with $|X| = |S|$, there is a permutation that sends S to X; thus the subgroup of permutations in T is $|S|$-set-transitive.

Now, let $\delta_w \in T$ have rank $n-k$ and consider $[[\delta_w(S),\Delta_w(S)]]$. By Lemma 2 this interval has type $(|\delta_w(S)|,|\Delta_w(S)|)$, so it is a non-empty interval. This implies $\delta_w(S)$ and $\delta_w(\overline{S})$ are disjoint. It follows that $|\operatorname{im}\delta_w| = |\delta_w(Q_n)| = |\delta_w(S)| + |\delta_w(\overline{S})|$. Since the rank of δ_w is $n-k$, $|\delta_w(\overline{S})| = (n-k) - |\delta_w(S)|$. Thus $|\Delta_w(S)| = n-(n-k-|\delta_w(S)|) = |\delta_w(S)|+k$, which gives $|\Delta_w(S)| - |\delta_w(S)| = k$.

Consider $[[S,S\cup\{i\}]]$ for $i \notin S$. Since $(|S|,|S|+1)$ is an S-type, by Lemma 2 this interval is reachable in $\mathscr{A}_S^D$. Thus there is a $\delta_w \in T$ such that $(|\delta_w(S)|,|\Delta_w(S)|) = (|S|,|S|+1)$. By the argument above, this δ_w must have rank $n-(|\Delta_w(S)|-|\delta_w(S)|) = n-1$. Hence T contains a transformation of rank $n-1$. ☐

5.2 Semigroups and Groups

To prove the other direction of Theorem 3, we use some results from semigroup and group theory. The first is a result of Livingstone and Wagner [6]:

Proposition 6. *Let G be a permutation group of degree $n \geq 4$. If $2 \leq k \leq \frac{n}{2}$, then the number of orbits when G acts on k-subsets of Q_n is at least the number of orbits when G acts on $(k-1)$-subsets of Q_n.*

Using this proposition, we can easily prove

Lemma 3. *Let G be a k-set-transitive permutation group of degree $n \geq 4$ and suppose $2 \leq k \leq \frac{n}{2}$. Then:*

1. *G is $(n-k)$-set-transitive.*
2. *G is ℓ-set-transitive for each ℓ such that $0 \leq \ell \leq k$ or $n-k \leq \ell \leq n$.*

Proof. (1): Suppose G is k-set-transitive. If U and V are $(n-k)$-subsets of Q_n, then $\overline{U}$ and $\overline{V}$ are k-subsets, and there exists a permutation $p \in G$ mapping $\overline{U}$ to $\overline{V}$. But if p maps $\overline{U}$ to $\overline{V}$, then it maps U to V; thus G can map any $(n-k)$-subset to any other $(n-k)$-subset, and so is $(n-k)$-set-transitive.

(2): Suppose G is k-set-transitive and $2 \leq k \leq \frac{n}{2}$. Then there is one orbit when G acts on k-subsets. By Proposition 6, there is one orbit when G acts on $(k-1)$-subsets. This implies G is $(k-1)$-set-transitive. Repeating this argument we conclude that G is ℓ-set-transitive for $0 \leq \ell \leq k$. By (1), G is also ℓ-set-transitive for $n-k \leq \ell \leq n$. ☐

Note that for $n = 3$, a permutation group of degree 3 is set-transitive if and only if it is transitive.

The second result we use is a theorem of Ruškuc, published by McAlister [7]:

Proposition 7. *Let G be a permutation group of degree $n \geq 3$ and let $t\colon Q_n \to Q_n$ be a unitary transformation. Let T be the transformation semigroup generated by $G \cup \{t\}$. Then T contains all singular transformations if and only if G is 2-set-transitive.*

We can use this to prove the following lemma:

Lemma 4. *Let G be a 2-set-transitive permutation group of degree $n \geq 3$ and let $t\colon Q_n \to Q_n$ be a transformation of rank $n-1$. Then the transformation semigroup T generated by $G \cup \{t\}$ contains all singular transformations.*

Proof. By Proposition 7, if G is 2-set-transitive and t is a unitary transformation, then T contains all singular transformations. Thus it suffices to show that if t is any transformation of rank $n-1$, then $G \cup \{t\}$ generates a unitary transformation.

For each transformation $s\colon Q_n \to Q_n$, we define a set of tuples called *s-paths*. For $k \geq 2$, a tuple $(i_1, \dots, i_k)$ of distinct elements of Q_n is an *s-path of length k* if $s(i_j) = i_{j+1}$ for $1 \leq j < k$ and $s(i_k) = i_\ell$ for some $\ell < k$. An s-path $(i_1, \dots, i_k)$ is *incomplete* if there exists a in Q_n such that $(a, i_1, \dots, i_k)$ is an s-path, and *complete* otherwise. An s-path $(i_1, \dots, i_k)$ is *cyclic* if $s(i_k) = i_1$ and *acyclic* otherwise. The element i_1 of the acyclic s-path $(i_1, \dots, i_k)$ is called the *head*.

Let t be a transformation of rank $n-1$, and consider the t-paths. If a t-path is complete and acyclic, its head must be an element of $\operatorname{coim} t$. Since t has rank $n-1$, $|\operatorname{coim} t| = 1$, and so there is precisely one complete acyclic t-path. Let $(a_1, \dots, a_k)$ be that complete acyclic t-path, and suppose $t(a_k) = a_\ell$.

Since G is 2-set-transitive, it is 1-set-transitive by Lemma 3. Thus there exists a permutation $p \in G$ with $p(a_1) = a_{\ell-1}$. Let $pt = p \circ t$, and consider pt-paths. Since pt has rank $n-1$, there is only one complete acyclic pt-path; the head of this path must be $a_{\ell-1}$, since $\operatorname{coim} pt = \{a_{\ell-1}\}$. Observe that $pt(a_k) = p(t(a_k)) = p(a_\ell) = p(t(a_{\ell-1})) = pt(a_{\ell-1})$; it follows that $(a_{\ell-1}, p(a_\ell), pt(p(a_\ell)), \dots, a_k)$ is the complete acyclic pt-path.

Now, let n be the product of the lengths of all the complete cyclic pt-paths and the incomplete cyclic pt-path $(p(a_\ell), pt(p(a,\ell)), \dots, a_k)$. Then we have $(pt)^n = (a_{\ell-1} \to (pt)^n(a_{\ell-1}))$, where $(pt)^n$ is pt composed with itself n times. This proves that T must contain all singular transformations, since it is 2-set-transitive and contains a unitary transformation. □

These results are sufficient to prove the other direction of Theorem 3:

Theorem 3 ($\Leftarrow$ Direction). Let L be a language of complexity $n \geq 3$, let T be the transition semigroup of the minimal DFA of L, and let A_S be an atom of L. Suppose either $n = 3$ and $1 \leq |S| \leq 2$, or $n \geq 4$ and $2 \leq |S| \leq n-2$. We prove that if the subgroup of permutations in T is $|S|$-set-transitive and T contains a transformation of rank $n-1$, then A_S has maximal complexity.

By Lemmas 3 and 4, T contains all singular transformations. By Theorem 2, L has 2^n atoms. From the proof of Lemma 2, $\Psi(n, |S|)$ is a tight bound on the number of S-intervals in the atomic poset of L. Since L has 2^n atoms (the maximal possible), the number of atomic S-intervals of L meets the bound $\Psi(n, |S|)$. It remains to show that all these intervals are reachable in the minimal DFA $\mathscr{A}_S^D$ of A_S. From the inital state $[[S,S]]$ of $\mathscr{A}_S^D$, we can reach the empty interval by $(i \to j)$ where $i \in S$ and $j \notin S$; thus it suffices to consider non-empty intervals.

Let $[[V,U]]$ be a non-empty atomic S-interval of L with type $(|V|, |U|)$. By the definition of an atomic S-interval, $1 \leq |V| \leq |S|$ and $|S| \leq |U| \leq n-1$ and $V \subseteq U$. Thus there exists a set X such that $|X| = |S|$ and $V \subseteq X \subseteq U$. Since the subgroup of permutations in T is $|S|$-set-transitive, there is a permutation $\delta_w \in T$ that sends S to X; thus $\eta_w([[S,S]]) = [[X,X]]$. If $V = X = U$, we are done, so assume that $V \subset X$ or $X \subset U$. If $V \subset X$ and $|V| \geq 2$, we can shrink the lower bound of $[[X,X]]$ as follows: select distinct $i, j \in Q_n$ such that $i \in X \setminus V$ and $j \in V$. Since T contains all unitary transformations, there is a $\delta_x \in T$ such that $\delta_x = (i \to j)$. Since $i \notin \overline{X}$, $\delta_x(\overline{X}) = \overline{X}$ and thus $\Delta_x(X) = X$. It follows that $\eta_x([[X,X]]) = [[X \setminus \{i\}, X]]$. Repeating this process, we can reach $[[V,X]]$ for all V with $1 \leq |V| \leq |S|$. By a similar process, we can repeatedly enlarge the upper bound of $[[V,X]]$ to reach $[[V,U]]$. Thus all $\Psi(n, |S|)$ atomic S-intervals of L are reachable in $\mathscr{A}_S^D$. By Lemma 2, A_S has maximal complexity. □

Remark 2. The proof above works for the $|S| = 1$ and $|S| = n-1$ cases if we assume that T contains all unitary transformations, rather than only assuming it contains some transformation of rank $n-1$.

6 Proof of Theorem 1

Having proved Theorems 2 and 3, we need only a bit more work to prove our main theorem.

Let L be a language with complexity $n \geq 3$ and let T be the transition semigroup of the minimal DFA of L. If L is maximally atomic, then by Theorem 3 and Lemma 3, the subgroup of permutations in T is k-set-transitive for $1 \leq k \leq n-1$, and hence is set-transitive; also, by Theorem 3, T contains a transformation of rank $n-1$. This proves one direction of the theorem.

For the other direction, suppose the subgroup of permutations in T is set-transitive and contains a transformation of rank $n-1$. By Theorem 2, L has 2^n atoms. By Theorem 3, if $n \geq 4$ and $2 \leq |S| \leq n-2$ or $n = 3$ and $1 \leq |S| \leq 2$, then A_S has maximal complexity. By Lemma 4, T contains all singular transformations and hence all unitary transformations; so by Remark 2, A_S has maximal complexity if $|S| = 1$ or $|S| = n-1$. The only remaining cases are $|S| = 0$ and $|S| = n$.

Let $\mathscr{A}_S^D$ be the minimal DFA of A_S. By Lemma 2, to show that A_S has maximal complexity, it suffices to show that all atomic S-intervals of L are reachable in $\mathscr{A}_S^D$. If $|S| = 0$, then $S = \emptyset$, and the atomic $\emptyset$-intervals of L are those with type $(0,i)$ where $0 \leq i \leq n-1$. The initial state of $\mathscr{A}_\emptyset^D$ is $[[\emptyset,\emptyset]]$; thus a reachable state looks like $[[\delta_w(\emptyset),\Delta_w(\emptyset)]] = [[\emptyset, \operatorname{coim} \delta_w]]$ for some $w \in \Sigma^*$.

Since T contains all singular transformations, for all $U \subset Q_n$, there exists $t \in T$ such that $\operatorname{coim} t = U$. Hence for all $U \subset Q_n$, $[[\emptyset,U]]$ is reachable in $\mathscr{A}_\emptyset^D$. Thus all intervals of type $(0,i)$ are reachable, for $0 \leq i \leq n-1$. By Lemma 2, $A_\emptyset$ has maximal complexity. By a similar argument, when $|S| = n$, the atom A_{Q_n} has maximal complexity. Thus all 2^n atoms have maximal complexity; this completes the proof. $\square$

7 Conclusions

We have defined a new class of regular languages – the maximally atomic languages – and proven that a language of complexity n is maximally atomic if and only if the transition semigroup of its minimal DFA is set-transitive and contains a transformation of rank $n-1$. Since the set-transitive groups have been fully classified, it is easy to construct examples of maximally atomic languages and study them. We have also derived a formula for the transition functions of átomata and minimal DFAs of atoms.

Acknowledgements: We thank a referee for giving many suggestions to improve our proofs.

References

[1] Ross A. Beaumont & Raymond P. Peterson (1955): *Set-transitive permutation groups*. *Canadian Journal of Mathematics* 7, pp. 35–42, doi:10.4153/CJM-1955-005-x.

[2] Janusz Brzozowski & Gareth Davies (2013): *Maximal Syntactic Complexity of Regular Languages Implies Maximal Quotient Complexities of Atoms*. Available at `http://arxiv.org/abs/1302.3906`.

[3] Janusz Brzozowski & Hellis Tamm (2013): *Complexity of Atoms of Regular Languages*. *Int. J. Found. Comput. Sci.* 24(7), pp. 1009–1027, doi:10.1142/S0129054113400285.

[4] Janusz Brzozowski & Hellis Tamm (2014): *Theory of Átomata*. *Theoret. Comput. Sci.*, doi:10.1016/j.tcs.2014.04.016. In press.

[5] Szabolcs Iván (2014): *Handle Atoms with Care*. Available at `http://arxiv.org/abs/1404.6632`.

[6] Donald Livingstone & Ascher Wagner (1965): *Transitivity of finite permutation groups on unordered sets*. *Mathematische Zeitschrift* 90(5), pp. 393–403, doi:10.1007/BF01112361.

[7] Donald B. McAlister (1998): *Semigroups generated by a group and an idempotent*. *Communications in Algebra* 26(2), pp. 243–254, doi:10.1080/00927879808826145.

Simplifying Nondeterministic Finite Cover Automata

Cezar Câmpeanu
Department of Computer Science and Information Technology,
The University of Prince Edward Island, Canada
ccampeanu@upei.ca

The concept of Deterministic Finite Cover Automata (DFCA) was introduced at WIA '98, as a more compact representation than Deterministic Finite Automata (DFA) for finite languages. In some cases representing a finite language, Nondeterministic Finite Automata (NFA) may significantly reduce the number of states used. The combined power of the succinctness of the representation of finite languages using both cover languages and non-determinism has been suggested, but never systematically studied. In the present paper, for nondeterministic finite cover automata (NFCA) and l-nondeterministic finite cover automaton (l-NFCA), we show that minimization can be as hard as minimizing NFAs for regular languages, even in the case of NFCAs using unary alphabets. Moreover, we show how we can adapt the methods used to reduce, or minimize the size of NFAs/DFCAs/l-DFCAs, for simplifying NFCAs/l-NFCAs.

1 Introduction

The race to find more compact representation for finite languages was started in 1959, when Michael O. Rabin and Dana Scott introduced the notion of Nondeterministic Finite Automata, and showed that the equivalent Deterministic Finite Automaton can be, in terms of number of states, exponential larger than the NFA. Since, it was proved in [25] that we can obtain a polynomial algorithm for minimizing DFAs, and in [16] was proved that an $O(n\log n)$ algorithm exists. In the meantime, several heuristic approaches have been proposed to reduce the size of NFAs [2, 18], but it was proved by Jiang and Ravikumar [19] that NFA minimization problems are hard; even in case of regular languages over a one letter alphabet, the minimization is NP-complete [10, 19].

On the other hand, in case of finite languages, we can obtain minimizing algorithms [22, 26] that are in the order of $O(n)$, where n is the number of states of the original DFA. In [4, 6, 21] it has been shown that using Deterministic Finite Cover Automata to represent finite languages, we have minimization algorithms as efficient as the best known algorithm for minimizing DFAs for regular languages.

The study of the state complexity of operations on regular languages was initiated by Maslov in 1970 [22, 23], but has not become a subject of systematic study until 1992 [27]. The special case of state complexity of operations on finite languages was studied in [5].

Nondeterministic state complexity of regular languages was also subject of interest, for example in [12, 13, 14, 15]. To find lower bounds for the nondeterministic state complexity of regular languages, the fooling set technique, or the extended fooling set technique may be used [3, 9, 10].

In this paper we show that NFCA state complexity for a finite language L can be exponentially lower than NFA or DFCA state complexity of the same language. We modify the fooling set technique for cover automata, to help us prove lower bounds for NFCA state complexity in section 3. We also show that the (extended) fooling set technique is not optimal, as we have minimal NFCAs with arbitrary number of states, and the largest fooling set has constant size. In section 4 we show that minimizing NFCAs is hard, and in section 5 we show that heuristic approaches for minimizing DFAs or NFAs need a special treatment when applied to NFCAs, as many results valid for the DFCAs are no longer true for NFCAs. In section 6, we formulate a few open problems and future research directions.

2 Notations and definitions

The number of elements of a set T is denoted by $\#T$. In case Σ is an alphabet, i.e, finite non-empty set, the free monoid generated by Σ is Σ^*, and it is the set of all words over Σ. The length of a word $w = w_1w_2\dots w_n$, $n \geq 0$, $w_i \in \Sigma$, $1 \leq i \leq n$, is $|w| = n$. The set of words of length equal to l is Σ^l, the set of words of length less than or equal to l is denoted by $\Sigma^{\leq l}$. In a similar fashion, we define $\Sigma^{\geq l}$, $\Sigma^{<l}$, or $\Sigma^{>l}$. A finite automaton is a structure $A = (Q,\Sigma,\delta,q_0,F)$, where Q is a finite non-empty set called the set of states, Σ is an alphabet, $q_0 \in Q$, $F \subseteq Q$ is the set of final states, and δ is the transition function. For delta, we distinguish the following cases:

- if $\delta : Q \times \Sigma \overset{\circ}{\longrightarrow} Q$, the automaton is deterministic; in case δ is always defined, the automaton is complete, otherwise it is incomplete;
- if $\delta : Q \times \Sigma \longrightarrow 2^Q$, the automaton is non-deterministic.

The language accepted by an automaton is defined by: $L(A) = \{w \in \Sigma^* \mid \delta(\{q_0\},w) \cap F \neq \emptyset\}$, where $\delta(S,w)$ is defined as follows:

$$\delta(S,\varepsilon) = S,$$

$$\delta(S,wa) = \bigcup_{q\in\delta(S,w)} \delta(\{q\},a).$$

Of course, $\delta(\{q\},a) = \{\delta(q,a)\}$ in case the automaton is deterministic, and $\delta(\{q\},a) = \delta(q,a)$, in case the automaton is non-deterministic.

Definition 1 *Let L be a finite language, and l be the length of the longest word w in L, i.e., $l = \max\{|w| \mid w \in L\}$*[1]*. If L is a finite language, L' is a cover language for L if $L' \cap \Sigma^{\leq l} = L$.*

A cover automaton for a finite language L is an automaton that recognizes a cover language, L', for L. An l-NFCA A is a cover automaton for the language $L(A) \cap \Sigma^{\leq l}$.

One could plainly see that any automaton that recognizes L is also a cover automaton.

The level of a state $s \in Q$ in a cover automaton $A = (Q,\Sigma,\delta,q_0,F)$ is the length of the shortest word that can reach the state s, i.e., $level_A(s) = min\{|w| \mid s \in \delta(q_0,w)\}$.

Let us denote by $x_A(s)$ the smallest word w, according to quasi-lexicographical order, such that $s \in \delta(q_0,w)$, see [6] for a similar definition in case of DFCA. Obviously, $level_A(s) = |x_A(s)|$.

For a regular language L, $\equiv_L$ denotes the Myhil-Nerode equivalence of words [17].

The similarity relation induced by a finite language L is defined as follows[6]: $x \sim_L y$, if for all $w \in \Sigma^{\leq l-\max\{|x|,|y|\}}$, $xw \in L$ iff $yw \in L$. A dissimilar sequence for a finite language L is a sequence $x_1,\dots,x_n$ such that $x_i \not\sim_L x_j$, for all $1 \leq i,j \leq n$ and $i \neq j$.

Now, we need to define the similarity for states in an NFCA, since it was the main notion used for DFCA minimization.

Definition 2 *In an NFCA $A = (Q,\Sigma,\delta,q_0,F)$, two states $p,q \in Q$ are similar, written $s \sim_A q$, if $\delta(p,w) \cap F \neq \emptyset$ iff $\delta(q,w) \cap F \neq \emptyset$, for all $w \in \Sigma^{\leq l-\max\{level(p),level(q)\}}$.*

[1] We use the convention that $max\emptyset = 0$.

In case the NFCA A is understood, we may omit the subscript A, i.e., we write $p \sim q$ instead of $p \sim_A q$, also we can write $level(p)$ instead of $level_A(p)$.

We consider only non-trivial NFCAs for L, i.e., NFCAs such that $level(p) \le l$ for all states p. In case $level(p) > l$, p can be eliminated, and the resulting NFA is still a NFCA for L. In this case, if $p \sim q$, then either $p,q \in F$, or $p,q \in Q \setminus F$, because $|\varepsilon| \le l - \max\{level(p), level(q)\}$.

Deterministic state complexity of a regular language L is defined as the number of states of the minimal deterministic automaton recognizing L, and it is denoted by $sc(L)$:

$$sc(L) = \min\{\#Q \mid A = (Q,\Sigma,\delta,q_o,F),\ \text{deterministic, complete, and } L = L(A)\}.$$

Non-deterministic state complexity of a regular language L is defined as the number of states of the minimal non-deterministic automaton recognizing L, and it is denoted by $nsc(L)$:

$$nsc(L) = \min\{\#Q \mid A = (Q,\Sigma,\delta,q_o,F),\ \text{non-deterministic and } L = L(A)\}.$$

For finite languages L, we can also define deterministic cover state complexity $csc(L)$ and non-deterministic cover state complexity $ncsc(L)$:

$$\begin{aligned} csc(L) &= \min\{\#Q \mid A = (Q,\Sigma,\delta,q_o,F),\ \text{deterministic, complete, and} \\ &\quad L = L(A) \cap \Sigma^{\le l}\}, \\ ncsc(L) &= \min\{\#Q \mid A = (Q,\Sigma,\delta,q_o,F),\ \text{non-deterministic, and} \\ &\quad L = L(A) \cap \Sigma^{\le l}\}. \end{aligned}$$

Obviously, $ncsc(L) \le nsc(L) \le sc(L)$, but also $ncsc(L) \le csc(L) \le sc(L)$. Thus, non-deterministic finite cover automata can be considered to be one of the most compact representation of finite languages.

3 Lower-bounds and Compression Ratio for NFCAs

We start this section analyzing few examples where nondeterminism, or the use of cover language, reduce the state complexity. Let us first analyze the type of languages where non-determinism, combined with cover properties, reduce significantly the state complexity.

We choose the language $L_{F_{m,n}} = \{a,b\}^{\le m} a \{a,b\}^{n-2}$, where $m,n \in \mathbb{N}$. In Figure 1 we can see an NFA recognizing L with $m+n$ states. We must note that the longest word in the language has $m+n-1$ letters. Let us analyze if the automaton in Figure 1 is minimal. The fooling set technique, introduced in [7] and [8], and used to prove the lower-bound for state complexity of NFAs, is stated in [3, 7] as follows:

Lemma 1 *Let $L \subseteq \Sigma^*$ be a regular language, and suppose there exists a set of pairs $S = \{(x_i, y_i) \mid 1 \le i \le n\}$, with the following properties:*

1. *If $x_i y_i \in L$, for $1 \le i \le n$ and $x_i y_j \notin L$, for all $1 \le i,j \le n$, $i \ne j$, then $nsc(L) \ge n$. The set S is called* a fooling set *for L.*
2. *If $x_i y_i \in L$, for $1 \le i \le n$ and for $1 \le i,j \le n$, if $i \ne j$, implies either $x_i y_j \notin L$ or $x_j y_i \notin L$, then $nsc(L) \ge n$. The set S is called* an extended fooling set *for L.*

Now consider the following set of pairs of words: $S = \{(b^m a b^j, b^{n-2-j}) \mid 0 \le j \le n-2\} \cup \{(a^i, b^{m-i} a b^{n-2}) \mid 0 \le i \le m\} = \{(x_k, y_k) \mid 1 \le k \le m+n\}$.

For $(x_k, y_k) \in S$, we have that

1. $x_k y_k = b^m ab^j b^{n-2-j} = b^m ab^{n-2} \in L$, or
2. $x_k y_k = a^i b^{m-i} ab^{n-2} \in L$.

Let us examine for each $1 \leq k, h \leq m+n$, $k \neq h$ if the words $x_k y_h$ and $x_h y_k$ are also in L. We have the following possibilities:

1. Case I
   (a) $x_k y_h = b^m ab^i b^{n-2-j} \notin L$, and
   (b) $x_h y_k = b^m ab^j b^{n-2-i} \notin L$.
2. Case II
   (a) $x_k y_h = a^i b^{m-j} ab^{n-2} \in L$, if $i < j$, but
   (b) $x_h y_k = a^j b^{m-i} ab^{n-2} \notin L$, if $i < j$ (because $|a^j b^{m-i} ab^{n-2}| = m+n-1+j-i > m+n-1$).
3. Case III
   (a) $x_k y_h = b^m ab^j b^{m-i} ab^{n-2} \notin L$ (because $|b^m ab^j b^{m-i} ab^{n-2}| = m+1+j+m+1+n-2 > m+n-1$), or
   (b) $x_h y_k = a^i b^{n-2-j} \in L$ if $n-2-j+1+i > n$, or $x_h y_k = a^i b^{n-2-j} \notin L$ $n-2-j+1+i < n$.

From the statement 2. of Lemma 1, it follows that the NFA is minimal. We must note the following:

1. we cannot use the weak form 1 to prove the lower-bound;
2. when proving the lower-bound, we concatenate words to obtain a word of length greater than the maximum length of the words in the language, and that's why $x_i y_j$ is rejected. Since in case of cover automata such words will be automatically rejected, there is no doubt that any fooling set type technique we may use to prove the lower-bound for NFCAs must consider the length, and we should ignore the cases when the length exceeds the maximal one.

Hence, the fooling set technique introduced in [7] and [8], and used to prove the lower-bound for state complexity of NFAs, can be modified to prove a lower-bound for minimal NFCAs, and it can be formulated for cover languages as an adaptation of Theorem 1 in [10].

Lemma 2 *Let $L \subseteq \Sigma^{\leq l}$ be a finite language such that the longest word in L has the length l, and suppose there exists a set of pairs $S = \{x_i, y_i) \mid 1 \leq i \leq n\}$, with the following properties:*

1. *If $x_i y_i \in L$ for $1 \leq i \leq n$ and for $1 \leq i, j \leq n$, $i \neq j$, and $x_i y_j \in \Sigma^{\leq l}$, we have that $x_i y_j \notin L$, then $ncsc(L) \geq n$.*

   The set S is called a fooling set *for L.*
2. *If $x_i y_i \in L$, for $1 \leq i \leq n$ and for $1 \leq i, j \leq n$, if $i \neq j$, implies either $x_i y_j \in \Sigma^{\leq l}$ and $x_i y_j \notin L$, or $x_j y_i \in \Sigma^{\leq l}$ and $x_j y_i \notin L$ for all, then $ncsc(L) \geq n$.*

   The set S is called an extended fooling set *for L.*

Proof Assume there exists an NFCA $A = (Q, \Sigma, \delta, q_0, F)$, with m states accepting L. For each i, $1 \leq i \leq n$, $x_i y_i \in L$, therefore we must have a state $s_i \in \delta(q_0, x_i)$ and $\delta(s_i, y_i) \cap F \neq \emptyset$. In other words, there exists a state $f_i \in F$ and $f_i \in \delta(s_i, y_i)$.

1. We claim $s_i \notin \delta(q_0, x_j)$ for all $j \neq i$. If $s_i \in \delta(q_0, x_j)$, then $f_i \in \delta(s_i, y_i) \subseteq \delta(q_0, x_j y_i)$, and because $|x_j y_i| \leq l$, it follows that $x_j y_i \in L$, a contradiction.

2. We consider the function $f : \{1,\dots,n\} \longrightarrow Q$ defined by $f(i) = s_i$, s_i as above. We claim that f is injective. If $f(i) = f(j)$, then $\delta(f(i),y_i) = \delta(f(j),y_i)$, also $\delta(f(j),y_j) = \delta(f(i),y_j)$. Because $\delta(f(i),y_i) \cap F \neq \emptyset$, we also have that $\delta(f(j),y_i) \cap F \neq \emptyset$, and because $|x_i y_j| \leq l$, it follows that $x_i y_j \in L$, a contradiction. If $|x_j y_i| \leq l$, using the same reasoning, will follow that $x_j y_i \in L$. In both cases we have a contradiction, thus Q must have at least n elements.□

For the example above, we discover that we cannot have more than one pair of the form $(a^i, b^{m-i}ab^{n-2})$, thus, applying the extended fooling set technique for NFCAs, the minimum number of states in a minimal NFCA is at least $n-2+1+1 = n$. This proves that the NFCA presented in Figure 2 is minimal.

It is easy to check that any two distinct words $w_1, w_2 \in \Sigma^{\leq n-1}$, $w_1 \neq w_2$, are not similar with respect to $\sim_L$. It follows that for the language presented in Figure 1, $csc(L) \geq 2^{n-1}$. One can also verify that for two distinct words uay and wax, if $|y| \neq |x|$, $|x|, |y| \leq n-2$, they are distinguishable; also, in case $|x| = |y| \leq n-2$, the word $a^{n-2-|x|}$ will distinguish between all the words for which $|u| < n-2-|x|$ or $|w| < n-2-|x|$, thus the number of states in the minimal DFA is even larger than $csc(L)$. In case $m = 2$ and $n = 4$, the minimal DFCA is presented in Figure 3. A simple computation shows us that the corresponding minimal DFA has 15 states.

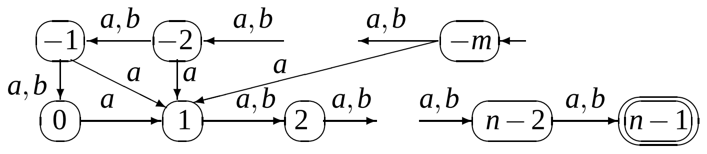


Figure 1: An NFA with $m+n$ states for the language $L_{F_{m,n}} = \{a,b\}^{\leq m}a\{a,b\}^{n-2}$.

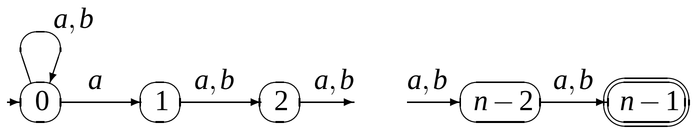


Figure 2: An NFCA with n states for the language $L_{F_{m,n}} = \{a,b\}^{\leq m}a\{a,b\}^{n-2}$, that is the same as the one in Figure 1. In case $m = 2$ and $n = 4$, the language is the same as the one described in Figure 3. An equivalent minimal NFA has $m+n$ states.

This language example shows that NFCAs may be a much more compact representation for finite languages than NFAs, or even DFCAs, and motivates the study of such objects. In terms of compression, clearly the number of states in the NFCA is exponentially smaller than the number of states in the DFA, and in some cases, even exponentially smaller than in an NFA.

Let's set $\Sigma = \{a\}$, $l > k \geq 2$, and choose the following language:

$$L_{l,k} = a(\Sigma^{\leq l} - \{(a^k)^n \mid n \geq 0\}). \tag{1}$$

In Figure 4, the NFCA A_k accepts the language $L_{l,k}$, therefore $ncsc(L_{l,k}) \leq csc(L_{l,k}) \leq sc(L_{l,k}) \leq \min(l+1, k+1) = k+1$. It is known [7, 13, 24] that the automaton A_k is minimal NFA for $\bigcup_{l \in \mathbb{N}} L_{l,k}$, if k is

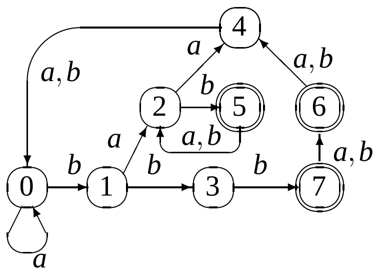


Figure 3: A minimal DFCA with 8 states for the language $L_{F_{2,4}} = \{a,b\}^{\leq 2}a\{a,b\}^2$, $l = 5$. The equivalent minimal DFA has 15 states.

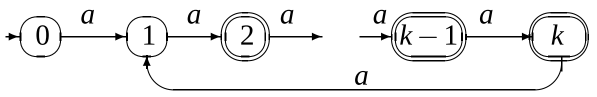


Figure 4: An NFA/NFCA A_k for $L_{l,k}$.

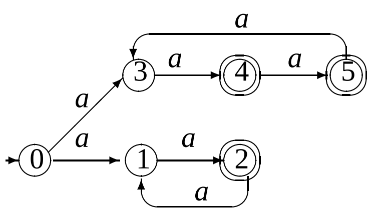
a prime number. However, this may not be a minimal NFCA, as illustrated by the example in Figure 5, where A_7 is not minimal for $L_{9,7}$, even if it is minimal NFA for the cover language.

We apply the extended fooling set technique for the language $L_{l,k}$. Because the alphabet is unary, all the words in an extended fooling set S are powers of a: $S \supseteq \{(a^{i_1}, a^{j_1}), (a^{i_2}, a^{j_2}), (a^{i_3}, a^{j_3}), \ldots, (a^{i_r}, a^{j_r})\}$, for some $r \in \mathbb{N}$. A simple computation shows that if $i_1, \ldots, i_r > 1$, and $i_1 + j_2 = z_{12}k + 1$ and $i_1 + j_3 = z_{13}k + 1$ for some $z_{12}, z_{13} \in \mathbb{N}$, then $i_2 + j_3 \neq z_{23}k + 1$ and $i_3 + j_2 \neq z_{32}k + 1$, for any $z_{23}, z_{32} \in \mathbb{N}$. It follows that $r \leq 3$.

Let A be an NFA accepting $L \supseteq L_k$, and we can consider that it is already in Chrobak normal form, as it is ultimately periodic. Thus, for each L, $nsc(L) \geq p_1 + \ldots p_s$, where p_i are primes, and each cycle has $p_i^{k_i}$ states, $1 \leq i \leq s$. Now, let us prove that A_k is minimal for some language $L_{l,k}$, $l \geq k$.

Assume there exists an automaton $B = (Q_B, \Sigma, \delta_B, q_{0,B}, F_B)$ with m states, $m \leq k+1$ such that $L(B) = L_{l,k}$. It follows that the language $L(B)$ will contain words with a length $x + hy$ for $x, y \leq k$, and all $h \in \mathbb{N}$. For h large enough, one of these words will be of length multiple of k plus 1, therefore, for large enough l, i.e., greater than some l_0, $L_{l,k} \neq L(B)$. Thus, the number of states in B is at least k. A_k is also a minimal NFCA for languages $L_{l,k}$, $l \geq l_0$, hence it follows that Theorem 7 in [10] is also valid for cover automata:

Theorem 1 *There is a sequence of languages $(L_{l,k})_{k \geq 2}$ such that the nondeterministic cover complexity*



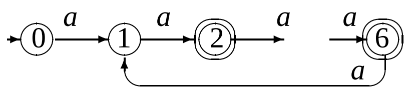


Figure 5: A minimal NFCA for $L_{9,7}$, left, and a minimal NFA for a cover language, right.

of $L_{l,k}$ is at least k, but the extended fooling set for $L_{l,k}$ is of size c, where c is a constant.

Now, we are ready to check how hard is to obtain this minimal representation of a finite language.

4 Minimization Complexity

In this section we show that minimizing NFCAs is hard, and we'll show it with the exact same arguments from [11], used to prove that minimizing NFAs is hard. We will describe the construction from [8, 11], showing that we can also use it with only a minor addition for cover NFAs. To keep the paper self contained, we include a complete description, and emphasize the changes required for the cover automata, rather than just presenting the differences.

Let us consider a logical formula $F \in 3SAT$, in the conjunctive normal form, i.e., $F = \bigwedge_{i=1}^{m} C_i$, where each clause C_i, $1 \le i \le m$, is defined using variables $x_1, \ldots, x_n$, $C_i = u_1 \vee u_2 \vee u_3$, and each u_j, $1 \le j \le 3$ are either x_i or $\neg x_i$. Let $p_1, p_2, \ldots, p_n$ be distinct prime numbers such that $p_1 < p_2 < \ldots < p_n$. We set $k = \prod_{i=1}^{n} p_i$, and using Chinese Remainder Theorem [20][2], it follows that the function $f : \mathbb{Z}_k \longrightarrow \prod_{i=1}^{n} \mathbb{Z}_{p_i}$ is bijective. We need to define a language L_F and a natural number l such that $L_F = \{a\}^*$, if and only if F is unsatisfiable, therefore, the finite language $L_F \cap \Sigma^{\le l}$ has $\{a\}^*$ as a cover language. We can construct an automaton B_i in $O(p_n)$ in a similar fashion as we build automata A_k that recognizes the language $L(B_i) = \{a^n \mid n \bmod p_i \notin \{0,1\}\}$. Let B be an automaton recognizing $\bigcup_{i=1}^{n} L(B_i)$. It is clear that it can be constructed in $O(n \cdot p_n)$ time. For each clause C_i such that a_1, a_2, a_3 is an assignment of its variables for which C_i is not satisfied, we define $L_{C_i} = \cap_{i=1}^{3} \{a^n \mid n \bmod p_i = a_i\}$. An automaton C_i accepting L_{C_i} can be constructed in $O(p_n^3)$ time[3]. Setting $L_F = \bigcup_{i=1}^{m} L_{C_i} \cup L(B)$, it follows that $L_F = \{a\}^*$ iff F is satisfiable. Moreover, L_F is a cyclic language with period at most k, thus setting $l = k$, we have that $L_F \cap \{a\}^{\le l}$ has $\{a\}^*$ as a cover language iff F is satisfiable. Since according to [1], primality test can be done in polynomial time, we can find the first n prime numbers in polynomial time, which means that our NFA construction can also be done in polynomial time. If F is unsatisfiable, then $ncsc(L) = 1$, if F is satisfiable, then the minimal period of L_F is $\frac{l}{2}$, according to [7, 8], and the minimal number of states in an NFA is at least equal to the largest prime number dividing its period, which is p_n. Using the same argument as in [11], it follows that the existence of a polynomial algorithm to decide if $ncsc(L) = o(n)$ implies that $nsc(L) = o(n)$, therefore we can solve $3SAT$ in polynomial time, i.e., $P = NP$. Consequently, we proved that

Theorem 2 *Minimizing either NFCAs or l-NFCAs is at least NP-hard.*

5 Reducing the Number of States of NFCAs

Assume the DFA $A = (Q, \Sigma, \delta, q_0, F)$ is minimal for L, and the minimal NFA is $A' = (Q', \Sigma, \delta', q_0, F)$, where $Q' = Q - \{d\}$, $\delta'(s,p) = \delta(s,p)$, if $\delta(s, p \in Q')$ and $\delta'(s,p) = \emptyset$ if $\delta(s,p) = d$. In other words, the minimal NFA is the same as the DFA, except that we delete the dead state. We may have a minimal DFCA as A, and A' as a minimal NFA, but not as a minimal NFCA, as illustrated by A_7 and $L_{9,7}$.

We need to investigate if classical methods to reduce the number of states in an NFA or DFA/DFCA can also be applied to NFCAs, thus, we first analyze the state merging technique. For NFAs, we distinguish between two main ways of merging states: (1) a weak method, where two states are merged

[2]Theorem I.3.3, page 21

[3]Using Cartesian product construction, for example.

by simply collapsing one into the other, and consolidate all their input and output transitions, and (2), a strong method, where one state is merged into another one by redirecting its input transitions toward the other state, and completely deleting it and all its output transitions. The same methods are considered for NFCAs.

Definition 3 *Let $A = (Q, \Sigma, \delta, q_0, F)$ be a NFCA for the finite language L.*

1. *We say that the state q is* weakly mergible *in state p if the automaton $A' = (Q', \Sigma, \delta', q_0, F')$, where $Q' = Q - \{q\}$, $F' = F \cap Q'$, and*

$$\delta(s,a) = \begin{cases} \delta(s,a), & \text{if } \delta(s,a) \subseteq Q' \text{ and } s \neq p, \\ (\delta(s,a) \setminus \{q\}) \cup \{p\}, & \text{if } q \in \delta(s,a) \text{ and } s \neq p, \\ (\delta(s,a) \cup \delta(q,a)) \setminus \{q\}, & \text{if } s = p \end{cases}$$

   is also a NFCA for L. In this case we write $p \precapprox q$.

2. *We say that the state q is* strongly mergible *in state p, if the automaton $A' = (Q', \Sigma, \delta', q_0, F')$, where $Q' = Q - \{q\}$, $F' = F \cap Q'$, and*

$$\delta(s,a) = \begin{cases} \delta(s,a), & \text{if } \delta(s,a) \subseteq Q' \\ (\delta(s,a) \setminus \{q\}) \cup \{p\}, & \text{if } q \in \delta(s,a), \end{cases}$$

   is also a NFCA for L. In this case we write $p \precsim q$.

In case $p \precapprox q$, $(L_p^L L_p^R \cup L_p^L L_q^R \cup L_q^L L_p^R \cup L_q^L L_q^R) \cap \Sigma^{\leq l} \subseteq L$ and in case $p \precsim q$, $L_q^L L_q^R \cap \Sigma^{\leq l} \subseteq (L_p^L L_p^R \cup L_q^L L_p^R) \cap \Sigma^{\leq l} \subseteq L$, where for $s \in Q$ $L_s^L = \{w \in \Sigma^* \mid s \in \delta(q_0, w)\}$ and $L_s^R = \{w \in \Sigma^* \mid \delta(s,w) \cap F \neq \emptyset\}$.

For the case of DFCAs, if A is a DFCA for L and two states are similar with respect to the similarity relation induced by A, then all the words reaching these states are similar. Moreover, if two words of minimal length reach two distinct states in a DFCA, and the words are similar with respect to L, then the states in the DFCA must be similar with respect to the similarity relation induced by A. These results are used for DFCA minimization, and we need to verify if they can be used in case of NFCAs. In the following lemmata we show that the corresponding results are no longer true.

Lemma 3 *Let $A = (Q, \Sigma, \delta, q_0, F)$ be a NFCA for the finite language L. It is possible that $x_A(s) \sim_L x_A(q)$, but s and q are not mergible.*

Proof For the automaton in Figure 5, left, $x_A(3) = x_A(1)$, but the states 1 and 3 are not mergible, as the resulting automaton would not reject a^7.

Lemma 4 *Let $A = (Q, \Sigma, \delta, q_0, F)$ be a NFCA for the finite language L, and $p, q \in Q$, $p \neq q$. It is possible to have $x, y \in \Sigma^*$, $p \in \delta(q_0, x)$, $q \in \delta(q_0, y)$, $p \sim q$, and $x \not\sim_L y$.*

Proof Consider the language $L = L(A) \cap \{a,b\}^{\leq 14}$, where A is depicted in Figure 5.

We have that:

- $aa \not\sim_L ba$, because $aaa \notin L$, but $baa \in L$;
- $2 \in \delta(0, ba)$, $7 \in \delta(0, aa)$, and
- $2 \sim_A 7$, because $\delta(2, a^{2k}) = \{2\} \subseteq F$, $\delta(2, a^{2k+1}) = \{1\} \cap F = \emptyset$, $\delta(7, a^{2k}) = \{7\} \subseteq F$, $\delta(7, a^{2k+1}) = \{6\} \cap F = \emptyset$, and $\delta(2, w) = \delta(7, w) = \emptyset$, for all $w \in \Sigma^* - \{a\}^*$.□

Let us verify the case when two states p, q are similar, or we can distinguish between them.

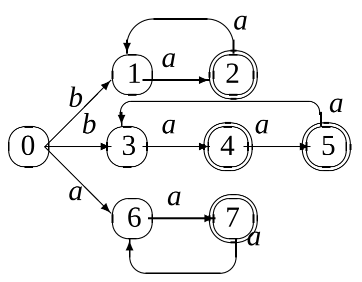


Figure 6: An example where $p \sim_A q$, $x \not\sim_L y$, but $p \in \delta(q_0,x)$ and $q \in \delta(q_0,y)$, $aa \not\sim_L ba$, $2 \in \delta(0,ba)$, $7 \in \delta(0,aa)$, and $2 \sim_A 7$.

Lemma 5 *Let $A = (Q,\Sigma,\delta,q_0,F)$ be a NFCA for the finite language L, $p,q \in Q$, $p \neq q$, and either $p,q \in F$, or $p,q \notin F$. Assume $r \in \delta(p,a)$ and $s \in \delta(q,a)$.*

1. *If $r \sim_A s$, for all possible choices of r and s, then $p \sim_A q$.*
2. *The converse is false, i.e., we may have $r \not\sim_A s$, for some r and s, and $p \sim_A q$.*

Proof Assume $p \not\sim_A q$, and let $w \in \Sigma^{\leq l-\max\{level(p),level(q)\}} \cap \Sigma^+$. Because either $p,q \in F$, or $p,q \notin F$, we have that $\delta(p,aw) \cap F \neq \emptyset$ and $\delta(q,aw) \cap F = \emptyset$, or $\delta(p,aw) \cap F = \emptyset$, and $\delta(q,aw) \cap F \neq \emptyset$. If $\delta(p,aw) \cap F \neq \emptyset$ and $\delta(q,aw) \cap F = \emptyset$, it follows that we have two states $r \in \delta(p,a)$ and $s \in \delta(q,a)$ such that $\delta(r,w) \cap F \neq \emptyset$, and $\delta(s,w) \cap F = \emptyset$. This proves that the first implication is true. For the second implication, consider the automaton depicted in Figure 5 with $l = 14$, and the following states p,q,r,s: $p = q = 0$, $r = 1$, $s = 3$, and the letter b. We have that $p \sim q$, $1,3 \in \delta(p,b) = \delta(q,b) = \delta(0,b)$, but $r \not\sim s$, because $\delta(1,a) \cap F = \emptyset$ and $\delta(3,a) \cap F = \{4\} \neq \emptyset$.$\square$

This result contrasts with the one for the deterministic case for cover automata, and the main reason is the nondeterminism, not the fact that we work with cover languages.

Next, we would like to verify if similar states can be merged in case of NFCAs, also to check which type of merge works. In case we have two similar states, we can strongly merge them as shown below. In the case of DFCAs, if two states are similar, these can be merged. We must ensure that the same result is also true for NFCAs, and the next theorem shows it.

Theorem 3 *Let $A = (Q,\Sigma,\delta,q_0,F)$ be an NFCA for L, and $p,q \in Q$ such that $p \neq q$, and $p \sim q$. Then we have*

1. *if $level_A(p) \leq level_A(q)$, then $p \precsim q$.*
2. *It is possible that $p \not\precsim q$.*

Proof For the first part, let A' be the automaton obtained from A by strongly merging q in p. We need to show that A' is a cover NFCA for L. Let $w = w_1 \ldots w_n$ be a word in $\Sigma^{\leq l}$, $n \in \mathbb{N}$ and $w_i \in \Sigma$ for all i, $1 \leq i \leq n$. We now prove that $w \in L$ iff $\delta'(q_0,w) \cap F' \neq \emptyset$.

If we can find the states $\{q_0,q_1,\ldots,q_n\}$ such that $q_1 \in \delta(q_0,w_1)$, $q_2 \in \delta(q_1,w_2)$, $\ldots$, $q_n \in \delta(q_{n-1},w_n)$, $q_n \in F$ and $q \notin \{q_0,q_1,\ldots,q_n\}$, then $q_1 \in \delta'(q_0,w_1)$, $q_2 \in \delta'(q_1,w_2)$, $\ldots$, $q_n \in \delta'(q_{n-1},w_n)$, $q_n \in F'$, i.e., $\delta'(q_0,w) \cap F' \neq \emptyset$. Assume $q = q_j$, and j is the smallest with this property. If $j = n$, then $q \in F$, which implies $p \in F$, then $q_1 \in \delta'(q_0,w_1)$, $q_2 \in \delta'(q_1,w_2)$, $\ldots$, $q_n \in \delta'(p,w_n)$, which means $\delta'(q_0,w) \cap F' \neq \emptyset$.

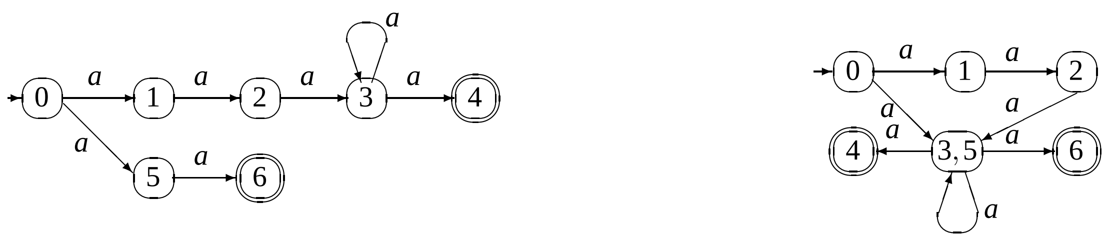


Figure 7: Example for weakly merging failure and similar states.

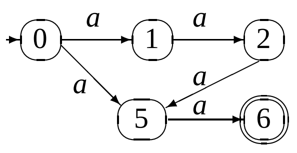


Figure 8: Example for strongly merging similar states for the example presented in Figure 7.

Assume the statements hold for $|w_j \dots w_n| < l'$ for $l' < l - |w|$ $(l - |w_1 ... w_j| \le l - level(q))$, and consider the case when $|w_{j-1} w_j \dots w_n| = l'$. If for every non-empty prefix of $w_{j+1} \dots w_n$, $w_{j-1} \dots w_h$, $q \notin \delta(p, w_{j-1} \dots w_h)$, then $\delta(p, w_{j+1} \dots w_n) \in F - \{q\}$ iff $\delta(q, w_{j+1} \dots w_n) \in F - \{q\}$, i.e., $\delta'(p, w_{j+1} \dots w_n) \cap F' \neq \emptyset$ iff $\delta(q, w_{j+1} \dots w_n) \cap F \neq \emptyset$.

Otherwise, let h be the smallest number such that $q \in \delta(q, w_{j+1} \dots w_h$. Then $|w_{h+1} \dots w_n| < l'$ (and $p \in \delta'(p, w_j \dots w_h)$). By induction hypothesis, $\delta'(p, w_{h+1} \dots w_n) \cap F' \neq \emptyset$ iff $\delta(q, w_{h+1} \dots w_n) \cap F \neq \emptyset$. Therefore, $\delta(p, w_{j+1} \dots w_h w_{h+1} \dots w_n) \cap F' \neq \emptyset$ iff $\delta(q, w_{j+1} \dots w_h w_{h+1} \dots w_n) \cap F \neq \emptyset$, proving the first part. For the second part, consider the automaton in Figure 7 as a NFCA for $L = \{a^2, a^4\}$. We have that $l = 4$ and $3 \sim 5$, because $level(3) = 3$, and $\delta(3, \varepsilon) \cap F = \delta(5, \varepsilon) \cap F = \emptyset$ $\delta(3, a) \cap F = \{4\}$, $\delta(5, a) \cap F = \{6\}$. We cannot weakly merge state 3 with state 5, as we would recognize $a^3 \notin L$. In Figure 8 we have the result for strongly merging state 3 in state 5.

We can observe that strongly merging states does not add words in the language, while weakly merging may add words. Because any DFCA is also a NFCA, then some smaller automata can be obtained from larger ones without using state merging technique, and the following lemma presents such a case. Also, the automaton in Figure 2 is obtained from automaton in Figure 1 by strongly merging states $0, \dots -m+1$ into state $-m$.

Lemma 6 *Let $A = (Q, \Sigma, \delta, q_0, F)$ be an NFCA for L, and consider the reduced sub-automaton generated by state p, $A = (Q_R, \Sigma, \delta_R, p, F)$, i.e., Q_R contains only reachable and useful states, and δ_R is the induced transition function. If $\delta(s, a) \cap Q_R = \emptyset$, for all $s \in (Q \setminus Q_R)$, we can find two regular languages L_1,L_2 such that*

- *$L_p = (L_1 \cup L_2) \cap \Sigma^{\le l - level(p)}$, and*
- *$nsc(L_1) + nsc(L_2) < \#Q_R + 1$,*

then A is not minimal.

Proof Let $A_i = (Q_i, \Sigma, \delta_i, q_{0,i}, F_i)$, $i = 1, 2$ be two NFAs for L_1 and L_2, and $L_p = (L_1 \cup L_2) \cap \Sigma^{\le l - level(p)}$. We define the automaton $B = ((Q \setminus Q_R) \cup \{p\} \cup Q_1 \cup Q_2, \Sigma, \delta_B, q_0, F_B)$ as follows: $F = (F \setminus Q_R) \cup F_1 \cup F_2$, in case $p \notin F$, and $F = (F \setminus Q_R) \cup F_1 \cup F_2 \cup \{p\}$ in case $p \in F$. For the transition function, we have $\delta_B(s,a) = \delta(s,a)$ if $s \in (Q \setminus Q_R)$, $\delta_B(s,a) = \delta_i(s,a)$ if $s \in Q_i$, $i = 1,2$, and $\delta_B(p,a) = \delta_1(q_{0,1},a) \cup \delta_2(q_{0,2},a) \cup \delta(p,a) \setminus Q_R$, if $p \notin \delta(p,a)$, and $\delta_B(p,a) = \delta_1(q_{0,1},a) \cup \delta_2(q_{0,2},a) \cup \delta(p,a) \setminus Q_R \cup \{p\}$, if $p \in \delta(p,a)$. Obviously, the automaton B recognizes the cover language for L, and its state complexity is lower.

This technique was used to produce the minimal NFCA for $L_{9,7}$ in Figure 5.

6 Conclusion

In this paper we showed that NFCAs are a more compact representation of finite languages than both NFAs and DFCAs, therefore it is a subject worth investigating. We presented a lower-bound technique for state complexity of NFCAs, and proved its limitations. We showed that minimizing NFCAs has at least the same level of difficulty as minimizing general NFAs, and that extra information about the maximum length of the words in the language does not help reducing the time complexity. We checked if some of the results involving reducing the size of automata for NFAs and DFCAs are still valid for NFCAs, and showed that most of them are no longer valid. However, the method of strong merging states still works in case of NFCAs, and we showed that there are also other methods that could be investigated.

As future research, below is a list of problems we consider worth investigating:

1. check if the bipartite graph lower-bound technique can be applied for NFCAs;
2. find bounds for nondeterministic cover state complexity;
3. investigate the problem of magic numbers for NFCAs. In this case, we can relate either to DFCAs, or DFAs.

References

[1] M. Agrawal, N. Kayal & N. Saxena (2004): *PRIMES is in P. Annals of mathematics*, pp. 781–793. Available at http://dx.doi.org/10.4007/annals.2004.160.781.

[2] J. Amilhastre, P. Janssen & M-C. Vilarem (2001): *FA Minimisation Heuristics for a Class of Finite Languages. Lecture Notes in Computer Science* 2214, pp. 1 – 12. Available at http://dx.doi.org/10.1007/3-540-45526-4_1.

[3] J.-C. Birget (1992): *Intersection and union of regular languages and state complexity. Information Processing Letters* 43, pp. 185–190. Available at http://dx.doi.org/10.1016/0020-0190(92)90198-5.

[4] C. Câmpeanu, A. P˘aun & S. Yu (2002): *An Efficient Algorithm for Constructing Minimal Cover Automata for Finite Languages. International Journal of Foundations of Computer Science* 13(1), pp. 83 – 97. Available at http://dx.doi.org/10.1142/S0129054102000960.

[5] C. Câmpeanu, K. Culik II, K. Salomaa & S. Yu (2001): *State complexity of basic operations on finite languages. Lecture Notes in Computer Science* 2214, pp. 60–70. Available at http://dx.doi.org/10.1007/3-540-45526-4_6.

[6] C. Câmpeanu, N. Santean & S. Yu (1986): *Minimal cover-automata for finite languages. Theoretical Computer Science* 267(1-2), pp. 3–16. Available at http://dx.doi.org/10.1016/S0304-3975(00)00292-9.

[7] M. Chrobak (1986): *Finite Automata and Unary Languages. Theoretical Computer Science* 47(2), pp. 149–158. Available at http://dx.doi.org/10.1016/0304-3975(86)90142-8.

[8] G.Gramlich (2003): *Probabilistic and Nondeterministic Unary Automata. Lecture Notes in Computer Science* 2747, pp. 460 – 469. Available at http://dx.doi.org/10.1007/978-3-540-45138-9_40.

[9] I. Glaister & J. Shallit (1996): *A lower bound technique for the size of nondeterministic finite automata*. *Information Processing Letters* 59, pp. 75 – 77. Available at http://dx.doi.org/10.1016/0020-0190(96)00095-6.

[10] H. Gruber & M. Holzer (2006): *Finding lower bounds for nondeterministic state complexity is hard*. *Lecture Notes in Computer Science* 4036, pp. 363–374. Available at http://dx.doi.org/10.1007/11779148_33.

[11] H. Gruber & M. Holzer (2007): *Computational Complexity of NFA Minimization for Finite and Unary Languages*. *LATA* 8, pp. 261–272. Available at http://www2.tcs.ifi.lmu.de/~gruberh/data/lata07-submission.pdf.

[12] M. Holzer & M. Kutrib (2003): *State complexity of basic operations on nondeterministic finite automata*. *Lecture Notes in Computer Science* 2608, pp. 148–157. Available at http://dx.doi.org/10.1007/3-540-44977-9_14.

[13] M. Holzer & M. Kutrib (2003): *Unary language operations and their nondeterministic state complexity*. *Lecture Notes in Computer Science* 2450, pp. 162–172. Available at http://dx.doi.org/10.1007/3-540-45005-X_14.

[14] M. Holzer & M. Kutrib (2009): *Descriptional and computational complexity of finite automata*. *Lecture Notes in Computer Science* 5457, pp. 23–42. Available at http://dx.doi.org/10.1007/978-3-642-00982-2_3.

[15] M. Holzer & M. Kutrib (2009): *Nondeterministic finite automata - recent results on the descriptional and computational complexity*. *Int. J. Found. Comput. Sci* 20(4), pp. 563–580. Available at http://dx.doi.org/10.1142/S0129054109006747.

[16] John Hopcroft (1971): *An $n \log n$ Algorithm for Minimizing States in a Finite Automaton*. In Z. Kohavi & A. Paz, editors: *Theory of Machines and Computations*, Academic Press, New York, pp. 189–196.

[17] John E. Hopcroft & Jeffrey D. Ullman (1979): *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley.

[18] L. Ilie, G. Navarro & S. Yu (2004): *On NFA reductions*. *Lecture Notes in Computer Science Volume: Theory Is Forever Essays Dedicated to Arto Salomaa on the Occasion of His 70th Birthday* 3113, pp. 112–124. Available at http://dx.doi.org/10.1007/978-3-540-27812-2_11.

[19] T. Jiang & B. Ravikumar (1993): *NFA minimization problems are hard*. *SIAM Journal on Computing* 22(1), pp. 117–141.

[20] N. Koblitz (1994): *A Course in Number Theory and Criptography*. Springer. Available at http://dx.doi.org/10.1007/978-1-4419-8592-7.

[21] H. Körner (2003): *A Time and Space Efficient Algorithm for Minimizing Cover Automata for Finite Languages*. *International Journal of Foundations of Computer Science* 14(6), pp. 1071–1086. Available at http://dx.doi.org/10.1142/S0129054103002187.

[22] A. N. Maslov (1970): *Estimates of the number of states of finite automata*. *Soviet Mathematics Doklady* 11, pp. 1373–1374.

[23] A. N. Maslov (1973): *Cyclic shift operation for languages*. *Probl. Inf. Transm* 9, pp. 333–338.

[24] F. Mera & G. Pighizzini (2005): *Complementing unary nondeterministic automata*. *Theoretical Computer Science* 330, pp. 349–360. Available at http://dx.doi.org/10.1016/j.tcs.2004.04.015.

[25] E. F. Moore (1956): *Gedanken-experiments on sequential machines*. *Automata studies, Annals of mathematics studies* 34, pp. 129–153.

[26] D. Revuz (1992): *Minimisation of acyclic deterministic automata in linear time*. *Theoretical Computer Science* 92(1), pp. 181 – 189. Available at http://dx.doi.org/10.1147/rd.32.0114.

[27] S. Yu, K. Salomaa & Q. Zhuang (1994): *The state complexities of some basic operations on regular languages*. *Theoretical Computer Science* 125(2), pp. 315–328. Available at http://dx.doi.org/10.1016/0304-3975(92)00011-F.

On the Expressiveness of TPTL and MTL over ω-Data Words

Claudia Carapelle[*], Shiguang Feng[*], Oliver Fernández Gil[*], Karin Quaas[†]
Institut für Informatik, Universität Leipzig,
D-04109 Leipzig, Germany
{carapelle, shiguang, fernandez, quaas}@informatik.uni-leipzig.de

Metric Temporal Logic (MTL) and Timed Propositional Temporal Logic (TPTL) are prominent extensions of Linear Temporal Logic to specify properties about data languages. In this paper, we consider the class of data languages of non-monotonic data words over the natural numbers. We prove that, in this setting, TPTL is strictly more expressive than MTL. To this end, we introduce Ehrenfeucht-Fraïssé (EF) games for MTL. Using EF games for MTL, we also prove that the MTL definability decision problem ("Given a TPTL-formula, is the language defined by this formula definable in MTL?") is undecidable. We also define EF games for TPTL, and we show the effect of various syntactic restrictions on the expressiveness of MTL and TPTL.

1 Introduction

Recently, verification and analysis of sets of *data words* have gained a lot of interest [18, 12, 10, 4, 5, 6, 7]. Here we consider ω-words, *i.e.*, infinite sequences over $\Sigma \times D$, where Σ is a finite set of labels, and D is a potentially infinite set of *data values*. One prominent example of data words are *timed words*, used in the analysis of real-time systems [1]. In this paper, we consider data words as behavioral models of one-counter machines. Therefore, in contrast to timed words, the sequence of data values within the word may be non-monotonic, and we choose the set of natural numbers as data domain. It is straightforward to adapt our results to the data domain of integers. In timed words, intuitively, the sequence of data values describes the timestamps at which the properties from the labels set Σ hold. Non-monotonic sequences of natural numbers, instead, can model the variation of an observed value during a time elapse: we can think of the heartbeat rate recorded by a cardiac monitor, atmospheric pressure, humidity or temperature measurements obtained from a meteorological station. For example, let $\mathsf{Weather} = \{\mathsf{sunny}, \mathsf{cloudy}, \mathsf{rainy}\}$ be a set of labels. A data word modeling the changing of the weather and highest temperature day after day could be:

$$(\mathsf{rainy}, 10)(\mathsf{cloudy}, 8)(\mathsf{sunny}, 12)(\mathsf{sunny}, 13)\ldots$$

For reasoning about data words, we consider extensions of *Linear Temporal Logic* (LTL, for short). One of these extensions is FreezeLTL, which extends LTL with a *freeze quantifier* that stores the current data value in a register variable. One can then check whether in a later position in the data word the data value equals the value stored in the register or not. Model checking one-counter machines with this logic is in general undecidable [12], and so is the satisfiability problem [10]. A good number of recent publications deal with decidable and undecidable fragments of FreezeLTL [10, 11, 12, 13].

Originally, the freeze quantifier was introduced in *Timed Propositional Temporal Logic* (TPTL, for short) [3]. Here, in contrast to FreezeLTL, a data value d can be compared to a register value x using

[*]The author is supported by *Deutsche Forschungsgemeinschaft* (DFG), GRK 1763 (QuantLA).
[†]The author is supported by *DFG*, project QU 316/1-1.

linear inequations of the form, *e.g.*, $d - x \leq 2$. Another widely used logic in the context of real-time systems is *Metric Temporal Logic* (MTL, for short) [16]. MTL extends LTL by constraining the temporal operators with intervals over the non-negative reals. It is well known that every MTL-formula can be effectively translated into an equivalent formula in TPTL. For the other direction, however, it turns out that the result depends on the data domain. For *monotonic data words* over the natural numbers, Alur and Henzinger [2] proved that MTL and TPTL are equally expressive. For timed words over the non-negative reals, instead, Bouyer et al. [8] showed that TPTL is strictly more expressive than MTL.

Both logics, however, have not gained much attention in the specification of non-monotonic data words. Recently we studied the decidability and complexity of MTL, TPTL and some of their fragments over non-monotonic data words [9], but still not much is known about their relative expressiveness, albeit they can express many interesting properties. To continue our example, using the MTL-formula $(\mathsf{sunny}\ \mathsf{U}_{[-3,-1]}\ \mathsf{cloudy})$ over the labels set $\mathsf{Weather}$, we can express the following property: it is sunny until it becomes cloudy and the highest temperature has decreased of 1 to 3 degrees. The following TPTL-formula expresses the fact that, at least three days from now, the highest temperature will be the same as today: $x.\mathsf{FFF}(x = 0)$. Over a data word, this formula expresses that there is a point whose data value is the same as that of the present one after at least three points. The main advantage of MTL with respect to TPTL is its concise syntax. It would be practical if we could show that, as in the case of monotonic data words over the natural numbers, MTL equals TPTL on data words. The goal of this paper is to investigate the relative expressiveness of TPTL and MTL when evaluated over data words.

In this paper, we show as a main result that for data words TPTL is strictly more expressive than MTL. More detailed, we use the formula $x.\mathsf{F}(b \wedge \mathsf{F}(c \wedge x \leq 2))$ to separate TPTL and MTL. This is the same formula used in the paper by Bouyer et al. [8] to separate these two logics over timed words. We also show that the simpler TPTL-formula $x.\mathsf{FFF}(x = 0)$ is not definable in MTL. Note that this formula is in the unary fragment of $\mathsf{FreezeLTL}$, which is very restrictive. The intuitive reason for the difference in expressiveness is that, using register variables, we can store data values at any position of a word to compare them with a later position, and it is possible to check that other properties are verified in between. This cannot be done using the constrained temporal operators in MTL. This does not result in a gap in expressiveness in the monotonic data words setting, because the monotonicity of the data sequence does not allow arbitrary values between two positions of a data word.

As a main tool for showing this result, we introduce *quantitative* versions of Ehrenfeucht-Fraïssé (EF) games for MTL and TPTL. In model theory, EF games are mainly used to prove inexpressibility results for first-order logic. Etessami and Wilke [14] introduced the EF game for LTL and used it to show that the Until Hierarchy for LTL is strict. Using our EF games for MTL and TPTL, we prove a number of results concerning the relation between the expressive power of TPTL and MTL, as well as between different fragments of both logics. We investigate the effects of restricting the syntactic resources. For instance, we show that TPTL that permits two register variables is strictly more expressive than TPTL restricted to one register variable. We also use EF games to show that the following problem is undecidable: given a TPTL-formula φ, is there an MTL-formula equivalent to φ?

We remark that quantitative EF games provide a very general and intuitive mean to prove results concerning the expressive power of quantitative logics. We would also like to point out that recently an extension of Etessami and Wilke's EF games has been defined [17] to investigate relative expressiveness of some fragments of the real-time version of MTL over *finite* timed words only. The proof of Theorem 1 in [17] relies on the fact that there is an integer bound on the timestamps of a finite timed word to deal with the potentially infinite number of equivalence classes of MTL formulas. It is not clear how this can be extended to *infinite* timed words. In contrast to this, the results in our paper using EF games can also be applied to *finite* data words.

2 Metric Temporal Logic and Timed Propositional Temporal Logic

In this section, we define two quantitative extensions of LTL: MTL and TPTL. The logics are evaluated over *data words*, defined in the following.

We use $\mathbb{Z}$ and $\mathbb{N}$ to denote the set of integers and the set of non-negative integers, respectively. Let P be a finite set of propositional variables. An *ω-data word*, or simply *data word*, w is an infinite sequence $(P_0, d_0)(P_1, d_1)\ldots$ of pairs in $2^{\mathsf{P}} \times \mathbb{N}$. Let $i \in \mathbb{N}$, we use $w[i]$ to denote the data word $(P_i, d_i)(P_{i+1}, d_{i+1})\ldots$ and use $(2^{\mathsf{P}} \times \mathbb{N})^{\omega}$ to denote the set of all data words.

2.1 Metric Temporal Logic

The set of formulas of MTL is built up from P by boolean connectives and a constraining version of the *until* operator:

$$\varphi ::= p \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \mathsf{U}_I \varphi_2$$

where $p \in \mathsf{P}$ and $I \subseteq \mathbb{Z}$ is a (half-)open or (half-)closed interval over the integers, possibly unbounded. We use pseudo-arithmetics expressions to denote intervals, *e.g.*, ≥ 1 to denote $[1, +\infty)$. If $I = \mathbb{Z}$, then we may omit the annotation I on U_I.

Formulas in MTL are interpreted over data words. Let $w = (P_0, d_0)(P_1, d_1)\ldots$ be a data word, and let $i \in \mathbb{N}$. We define the *satisfaction relation for* MTL, denoted by $\models_{\mathsf{MTL}}$, inductively as follows:

$$\begin{aligned}
&(w,i) \models_{\mathsf{MTL}} p \text{ iff } p \in P_i, \quad (w,i) \models_{\mathsf{MTL}} \neg\varphi \text{ iff } (w,i) \not\models_{\mathsf{MTL}} \varphi,\\
&(w,i) \models_{\mathsf{MTL}} \varphi_1 \wedge \varphi_2 \text{ iff } (w,i) \models_{\mathsf{MTL}} \varphi_1 \text{ and } (w,i) \models_{\mathsf{MTL}} \varphi_2,\\
&(w,i) \models_{\mathsf{MTL}} \varphi_1 \mathsf{U}_I \varphi_2 \text{ iff } \exists j > i \text{ such that } (w,j) \models_{\mathsf{MTL}} \varphi_2,\, d_j - d_i \in I,\\
&\qquad\qquad \text{and } \forall i < k < j, (w,k) \models_{\mathsf{MTL}} \varphi_1.
\end{aligned}$$

We say that a data word *satisfies* an MTL-formula φ, written $w \models_{\mathsf{MTL}} \varphi$, if $(w, 0) \models_{\mathsf{MTL}} \varphi$. We use the following syntactic abbreviations: $\mathtt{True} := p \vee \neg p$, $\mathtt{False} := \neg\mathtt{True}$, $\mathsf{X}_I\varphi := \mathtt{False}\mathsf{U}_I\varphi$, $\mathsf{F}_I\varphi := \mathtt{True}\mathsf{U}_I\varphi$. Note that the use of the *strict* semantics for the until operator is essential to define the next operator X_I.

Example. The following formula expresses the fact that the weather is sunny until it becomes cloudy and the temperature has decreased from one to three degrees. Furthermore in the future it will rain and the temperature will increase by at least one degree:

$$\mathsf{sunny}\ \mathsf{U}_{[-3,-1]}\ (\mathsf{cloudy} \wedge \mathsf{F}_{\geq 1}\ \mathsf{rainy}). \tag{1}$$

2.2 Timed Propositional Temporal Logic

Given an infinite countable set X of *register variables*, the set of formulas of TPTL is defined as follows:

$$\varphi ::= p \mid x \in I \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \mathsf{U} \varphi_2 \mid x.\varphi$$

where $p \in \mathsf{P}$, $x \in X$ and I is an interval over the integers, defined as for MTL. We will use pseudo-arithmetic expressions to denote intervals, *e.g.*, $x < 0$ denotes $x \in (0, -\infty)$. Intuitively, $x.\varphi$, means that we are *resetting* x to the current data value, and $x \in I$ means that, compared to the last time that we reset x, the data value has increased or decreased within the margins of the interval I.

Formulas in TPTL are interpreted over data words. A *register valuation* ν is a function from X to $\mathbb{N}$. Let $w = (P_0, d_0)(P_1, d_1)\ldots$ be a data word, let ν be a register valuation, and let $i \in \mathbb{N}$. The satisfaction relation for TPTL, denoted by $\models_{\mathsf{TPTL}}$, is inductively defined in a similar way as for MTL; we only give the definitions for the new formulas:

$$(w,i,\nu) \models_{\mathsf{TPTL}} x \in I \text{ iff } d_i - \nu(x) \in I,$$
$$(w,i,\nu) \models_{\mathsf{TPTL}} x.\varphi \text{ iff } (w,i,\nu[x \mapsto d_i]) \models_{\mathsf{TPTL}} \varphi,$$
$$(w,i,\nu) \models_{\mathsf{TPTL}} \varphi_1 \mathsf{U} \varphi_2 \text{ iff } \exists j > i, (w,j,\nu) \models_{\mathsf{TPTL}} \varphi_2, \forall i < k < j, (w,k,\nu) \models_{\mathsf{TPTL}} \varphi_2.$$

Here, $\nu[x \mapsto d_i]$ is the valuation that agrees with ν on all $y \in X \setminus \{x\}$, and maps x to d_i. We say that a data word w satisfies a TPTL-formula φ, written $w \models_{\mathsf{TPTL}} \varphi$, if $(w, 0, \overline{0}) \models_{\mathsf{TPTL}} \varphi$. Here, $\overline{0}$ denotes the valuation that maps each register variable to d_0. We use the same syntactic abbreviations as for MTL where the interval I for the temporal operators is ignored.

In the following, we define some fragments of TPTL. Given $n \geq 1$, we use TPTL^n to denote the set of TPTL-formulas that use at most n different register variables. The *unary fragment of* TPTL, denoted by $\mathsf{UnaTPTL}$, is defined by the following grammar:

$$\varphi ::= p \mid \neg\varphi \mid x \in I \mid \varphi_1 \wedge \varphi_2 \mid \mathsf{F}\varphi \mid \mathsf{X}\varphi \mid x.\varphi$$

We define $\mathsf{FreezeLTL}$ to be the subset of TPTL-formulas where the formula '$x \in I$' is restricted to be of the form '$x \in [0,0]$'. We denote combinations of these fragments in the expected manner; *e.g.*, $\mathsf{UnaFreezeLTL}^1$ denotes the unary fragment of TPTL in which only one register variable and equality checks of the form '$x \in [0,0]$' are allowed.

Example. The MTL-formula (1) in the above example is equivalent to the TPTL^1-formula

$$x.[\mathsf{sunny}\ \mathsf{U}\ (x \in [-3,-1] \wedge \mathsf{cloudy} \wedge x.\mathsf{F}\ (x \geq 1 \wedge \mathsf{rainy}))].$$

The formulas $x.((\mathsf{cloudy} \wedge x \leq 2) \mathsf{U}\ \mathsf{sunny})$ and $x.\mathsf{F}\ (\mathsf{cloudy} \wedge \mathsf{F}\ (\mathsf{sunny} \wedge x \leq 2))$, over the labels set $\mathsf{Weather}$ express the following properties:

1. The weather will eventually become sunny. Until then it is cloudy every day and the temperature is at most two degrees higher than the temperature at the present day.
2. It will be cloudy in the future, later it will become sunny, and the temperature will have increased by at most 2 degrees.

2.3 Relative Expressiveness

Let $\mathcal{L}$ and $\mathcal{L}'$ be two logics interpreted over elements in $(2^{\mathsf{P}} \times \mathbb{N})^{\omega}$, and $\varphi \in \mathcal{L}$ and $\varphi' \in \mathcal{L}'$ be two formulas. Define $L(\varphi) = \{w \in (2^{\mathsf{P}} \times \mathbb{N})^{\omega} \mid w \text{ satisfies } \varphi\}$. We say that φ is *equivalent* to φ' if $L(\varphi) = L(\varphi')$. Given a data language $\mathbf{L} \subseteq (2^{\mathsf{P}} \times \mathbb{N})^{\omega}$, we say that $\mathbf{L}$ is *definable in* $\mathcal{L}$ if there is a formula $\varphi \in \mathcal{L}$ such that $L(\varphi) = \mathbf{L}$. We say that a formula ψ is definable in $\mathcal{L}$ if $L(\psi)$ is definable in $\mathcal{L}$. We say that $\mathcal{L}'$ is *at least as expressive as* $\mathcal{L}$, written $\mathcal{L} \preccurlyeq \mathcal{L}'$, if each formula of $\mathcal{L}$ is definable in $\mathcal{L}'$. It is *strictly more expressive*, written $\mathcal{L} \prec \mathcal{L}'$ if, additionally, there is a formula in $\mathcal{L}'$ that is not definable in $\mathcal{L}$. Further, $\mathcal{L}$ and $\mathcal{L}'$ are *equally expressive*, written $\mathcal{L} \equiv \mathcal{L}'$, if $\mathcal{L} \preccurlyeq \mathcal{L}'$ and $\mathcal{L}' \preccurlyeq \mathcal{L}$. $\mathcal{L}$ and $\mathcal{L}'$ are *incomparable*, if neither $\mathcal{L} \preccurlyeq \mathcal{L}'$ nor $\mathcal{L}' \preccurlyeq \mathcal{L}$.

In this paper we are interested in the relative expressiveness of (fragments of) MTL and TPTL. It is straightforward to translate an MTL-formula into an equivalent TPTL^1-formula. So it can easily be seen that TPTL^1 is as least as expressive as MTL. However, we will show that there exist some TPTL^1-formulas that are not definable in MTL. For this we introduce the Ehrenfeucht-Fraïssé game for MTL. Before, we define the important notion of *until rank* of a formula.

2.4 Until Rank

The *until rank* of an MTL-formula φ, denoted by $\mathsf{Urk}(\varphi)$, is defined by induction on the structure of the formula:

- $\mathsf{Urk}(p) = 0$ for every $p \in \mathsf{P}$,
- $\mathsf{Urk}(\neg\varphi) = \mathsf{Urk}(\varphi)$, $\mathsf{Urk}(\varphi_1 \wedge \varphi_2) = \max\{\mathsf{Urk}(\varphi_1), \mathsf{Urk}(\varphi_2)\}$, and
- $\mathsf{Urk}(\varphi_1 \mathsf{U}_I \varphi_2) = \max\{\mathsf{Urk}(\varphi_1), \mathsf{Urk}(\varphi_2)\} + 1$.

We use $\mathsf{Cons}(\mathbb{Z})$ to denote the set $\{S \cup \{-\infty, +\infty\} \mid S \subseteq \mathbb{Z}\}$ and $\mathsf{FCons}(\mathbb{Z})$ for the subset of $\mathsf{Cons}(\mathbb{Z})$ which contains all *finite* sets in $\mathsf{Cons}(\mathbb{Z})$. Let $\mathcal{I} \in \mathsf{Cons}(\mathbb{Z})$, $k \in \mathbb{N}$. Define

$$\mathsf{MTL}^{\mathcal{I}} = \{\varphi \in \mathsf{MTL} \mid \text{the endpoints of } I \text{ in each operator } \mathsf{U}_I \text{ in } \varphi \text{ are in } \mathcal{I}\},$$
$$\mathsf{MTL}_k = \{\varphi \in \mathsf{MTL} \mid \mathsf{Urk}(\varphi) \leq k\}, \quad \mathsf{MTL}_k^{\mathcal{I}} = \mathsf{MTL}_k \cap \mathsf{MTL}^{\mathcal{I}}.$$

It is easy to check that $\mathsf{MTL} = \bigcup_{\mathcal{I} \in \mathsf{FCons}(\mathbb{Z})}^{k \in \mathbb{N}} \mathsf{MTL}_k^{\mathcal{I}}$, and $\mathsf{MTL}^{\mathcal{I}} = \bigcup_{\mathcal{I}' \in \mathsf{FCons}(\mathbb{Z})}^{\mathcal{I}' \subseteq \mathcal{I}, k \in \mathbb{N}} \mathsf{MTL}_k^{\mathcal{I}'}$ for each $\mathcal{I} \in \mathsf{Cons}(\mathbb{Z})$.

Lemma 1. *For each $\mathcal{I} \in \mathsf{FCons}(\mathbb{Z})$ and $k \in \mathbb{N}$, there are only finitely many formulas in $\mathsf{MTL}_k^{\mathcal{I}}$ up to equivalence.*

We define a family of equivalence relations over $(2^{\mathsf{P}} \times \mathbb{N})^{\omega} \times \mathbb{N}$. Let w_0, w_1 be two data words, $i_0, i_1 \geq 0$ be positions in w_0, w_1, respectively. Let $\mathcal{I} \in \mathsf{Cons}(\mathbb{Z})$, and let $k \in \mathbb{N}$. We say that (w_0, i_0) and (w_1, i_1) are $\mathsf{MTL}_k^{\mathcal{I}}$*-equivalent*, written $(w_0, i_0) \equiv_k^{\mathcal{I}} (w_1, i_1)$, if for each formula $\varphi \in \mathsf{MTL}_k^{\mathcal{I}}$, $(w_0, i_0) \models_{\mathsf{MTL}} \varphi$ if and only if $(w_1, i_1) \models_{\mathsf{MTL}} \varphi$.

3 The Ehrenfeucht–Fraïssé Game for MTL

Next we define the Ehrenfeucht–Fraïssé (EF) game for MTL. Let $\mathcal{I} \subseteq \mathsf{FCons}(\mathbb{Z})$, $k \in \mathbb{N}$, w_0, w_1 be two data words and i_0, i_1 be positions in w_0 and w_1, respectively. The k-round MTL EF game on (w_0, i_0) and (w_1, i_1) with respect to $\mathcal{I}$, denoted by $\mathrm{MG}_k^{\mathcal{I}}(w_0, i_0, w_1, i_1)$, is played by two players, called Spoiler and Duplicator, on the pair (w_0, w_1) of data words starting from the positions i_0 in w_0 and i_1 in w_1.

In each round of the game, Spoiler chooses a word and a position, and Duplicator tries to find a position in the respective other word satisfying conditions concerning the propositional variables and the data values in w_0 and w_1. We say that i_0 and i_1 *agree in the propositional variables* if $(w_0, i_0) \models_{\mathsf{MTL}} p$ iff $(w_1, i_1) \models_{\mathsf{MTL}} p$ for each $p \in \mathsf{P}$. We say that $m, n \in \mathbb{Z}$ *are in the same region* with respect to $\mathcal{I}$ if (a, b) or $[a, a]$ is the smallest interval I such that $a, b \in \mathcal{I}$ and $m \in I$, then $n \in I$. For example, let $\mathcal{I} = \{-\infty, 1, 3, 8, +\infty\}$, 1 and 2 are not in the same region with respect to $\mathcal{I}$, 4 and 5 are in the same region with respect to $\mathcal{I}$.

$\mathrm{MG}_k^{\mathcal{I}}(w_0, i_0, w_1, i_1)$ is defined inductively as follows. If $k = 0$, there are no rounds to be played, Spoiler wins if i_0 and i_1 do not agree in the propositional variables. Otherwise, Duplicator wins. If $k > 0$, in the first round,

1. Spoiler wins this round if i_0 and i_1 do not agree in the propositional variables. Otherwise, he chooses a word w_l ($l \in \{0, 1\}$), and a position $i'_l > i_l$ in w_l.
2. Then Duplicator tries to choose a position $i'_{(1-l)} > i_{(1-l)}$ in $w_{(1-l)}$ such that i'_0 and i'_1 agree in the propositional variables, and $d_{i'_0} - d_{i_0}$ and $d_{i'_1} - d_{i_1}$ are in the same region with respect to $\mathcal{I}$. If one of the conditions is violated, then Spoiler wins the round.

3. Then, Spoiler has two options: either he chooses to start a new game $\mathrm{MG}_{k-1}^{\mathcal{I}}(w_0, i_0', w_1, i_1')$; or
4. Spoiler chooses a position $i_{(1-l)} < i''_{(1-l)} < i'_{(1-l)}$ in $w_{(1-l)}$. In this case Duplicator tries to respond by choosing a position $i_l < i''_l < i'_l$ in w_l such that i''_0 and i''_1 agree in the propositional variables. If this condition is violated, Spoiler wins the round.
5. If Spoiler cannot win in Step 1, 2 or 4, then Duplicator wins this round. Then Spoiler chooses to start a new game $\mathrm{MG}_{k-1}^{\mathcal{I}}(w_0, i''_0, w_1, i''_1)$.

We say that Duplicator has a *winning strategy* for the game $\mathrm{MG}_k^{\mathcal{I}}(w_0, i_0, w_1, i_1)$ if she can win every round of the game regardless of the choices of Spoiler. We denote this by $(w_0, i_0) \sim_k^{\mathcal{I}} (w_1, i_1)$. Otherwise we say that Spoiler has a winning strategy. It follows easily that if $(w_0, i_0) \sim_k^{\mathcal{I}} (w_1, i_1)$, then for all $m < k$, $(w_0, i_0) \sim_m^{\mathcal{I}} (w_1, i_1)$.

Theorem 1. *For each $\mathcal{I} \in \mathsf{FCons}(\mathbb{Z})$ and $k \in \mathbb{N}$, $(w_0, i_0) \equiv_k^{\mathcal{I}} (w_1, i_1)$ if and only if $(w_0, i_0) \sim_k^{\mathcal{I}} (w_1, i_1)$.*

Theorem 2. *Let* **L** *be a data language. The following are equivalent:*

1. **L** *is not definable in* MTL.

2. For each $\mathcal{I} \in \mathsf{FCons}(\mathbb{Z})$ and $k \in \mathbb{N}$ there exist $w_0 \in \mathbf{L}$ and $w_1 \notin \mathbf{L}$ such that $(w_0, 0) \sim_k^{\mathcal{I}} (w_1, 0)$.

4 Application of the EF Game for MTL

4.1 Relative Expressiveness of TPTL and MTL

In this section, we present one of the main results in this paper: Over data words, TPTL is strictly more expressive than MTL. Before we come to this result, we show in the following lemma that in a data word the difference between data values is what matters, as opposed to the specific numerical value.

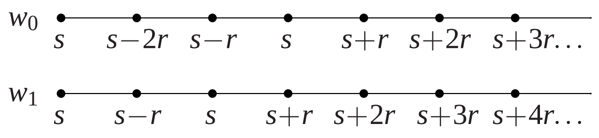
Lemma 2. *Let $w_0 = (P_0, d_0)(P_1, d_1)\ldots$ and $w_1 = (P_0, d_0 + c)(P_1, d_1 + c)\ldots$ for some $c \in \mathbb{N}$ be two data words. Then for every $k \in \mathbb{N}$ and $\mathcal{I} \in \mathsf{FCons}(\mathbb{Z})$, $(w_0, 0) \sim_k^{\mathcal{I}} (w_1, 0)$.*

Proof. The proof is straightforward. If Spoiler chooses a position in w_l $(l \in \{0,1\})$, then the duplicator can respond with the same position in $w_{(1-l)}$. $\square$

From now on, we use $(w_l : i, w_{(1-l)} : j)\,(l \in \{0,1\})$ to denote that Spoiler chooses a word w_l and a position i in w_l and Duplicator responds with a position j in $w_{(1-l)}$.

Proposition 1. *The* $\mathsf{UnaFreezeLTL}^1$*-formula* $x.\mathsf{FFF}(x = 0)$ *and the* TPTL*-formula* $x.\mathsf{F}(b \wedge \mathsf{F}(c \wedge x \leq 2))$ *are not definable in* MTL.

Proof. To show that the formula $\varphi = x.\mathsf{FFF}(x = 0)$ is not definable in MTL, for each $\mathcal{I} \in \mathsf{FCons}(\mathbb{Z})$ and $k \in \mathbb{N}$, we will define two data words w_0 and w_1 such that $w_0 \models \varphi$ and $w_1 \not\models \varphi$, and $(w_0, 0) \sim_k^{\mathcal{I}} (w_1, 0)$. Then, by Theorem 2, φ is not definable in MTL. So let $r, s \in \mathbb{N}$ be such that all numbers in $\mathcal{I}$ are contained in $(-r, +r)$ and $s \geq 2r$. Intuitively, we choose r in such a way that a jump of magnitude $\pm r$ in data value cannot be detected by $\mathsf{MTL}^{\mathcal{I}}$, as all constants in $\mathcal{I}$ are smaller than r. Define w_0 and w_1 as follows:

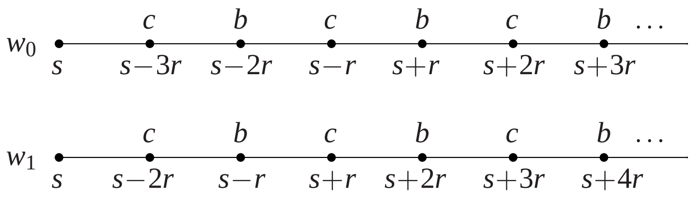


There are no propositional variables in w_0, w_1. We show that Duplicator has a winning strategy for the game $\mathrm{MG}_k^{\mathcal{I}}(w_0, 0, w_1, 0)$. The case $k = 0$ is trivial. Suppose $k > 0$. Note that after the first round, they start a new $(k-1)$-round game $\mathrm{MG}_{k-1}^{\mathcal{I}}(w_0, i_0, w_1, i_1)$, where $i_0, i_1 \geq 1$. By Lemma 2, Duplicator has a winning strategy for this game. So it is sufficient to show that Duplicator can win the first round. In the following we give the winning strategy for Duplicator in the first round.

Case / Move	1	2	3	4
1st	$(w_l : 1, w_{(1-l)} : 1)$, $(l \in \{0,1\})$	$(w_0 : 2, w_1 : 1)$	$(w_0 : i, w_1 : i-1)$, $(i > 2)$	$(w_1 : i, w_0 : i+1)$, $(i \geq 2)$
2nd	-	-	$(w_1 : j, w_0 : j+1)$, $(0 < j < i-1)$	$(w_0 : 1, w_1 : 1)$, or $(w_0 : j, w_1 : j-1)$, $(2 \leq j < i+1)$

By the choice of number r, $d_1^{w_0} - d_0^{w_0} (= -2r)$ is in the same region as $d_1^{w_1} - d_0^{w_1} (= -r)$. It is easy to check that Duplicator's responses satisfy the winning condition about the data value. Hence $(w_0, 0) \sim_k^{\mathcal{I}} (w_1, 0)$.

The proof for the formula $x.\mathsf{F}(b \wedge \mathsf{F}(c \wedge x \leq 2))$ is similar, we define $\mathcal{I}$, k, r and $s \geq 3r$ as above. We leave it to the reader to verify that Duplicator has a winning strategy for the game $\mathrm{MG}_k^{\mathcal{I}}(w_0, 0, w_1, 0)$ on the following two data words.


w_0
c b c b c b ...
s $s-3r$ $s-2r$ $s-r$ $s+r$ $s+2r$ $s+3r$
w_1
c b c b c b ...
s $s-2r$ $s-r$ $s+r$ $s+2r$ $s+3r$ $s+4r$

□

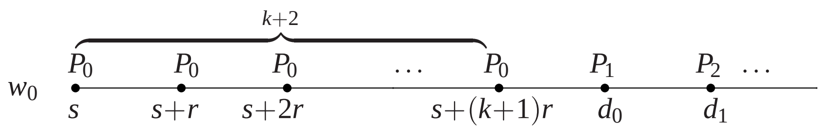
As a corollary, together with the fact that every MTL-formula is equivalent to a TPTL^1-formula we obtain the following.

Corollary 1. TPTL^1 *is strictly more expressive than* MTL.

4.2 The MTL Definability Decision Problem

For many logics whose expressiveness has been shown to be in a strict inclusion relation, the definability decision problem has been considered. For example, it is well known that Monadic second-order logic (MSO) defines exactly regular languages. Its first-order fragment (FO) defines the star-free languages which is a proper subset of regular languages. The problem of whether a MSO formula is equivalent to an FO formula over words is decidable. In our case the problem is stated as follows: Given a TPTL-formula φ, is φ definable in MTL? We show in the following, using the EF game method, that this problem is undecidable. First, we prove a Lemma.

Lemma 3. *Given an arbitrary $\mathcal{I} \in \mathsf{FCons}(\mathbb{Z})$, let $r, s \in \mathbb{N}$ be such that all numbers in $\mathcal{I}$ are contained in $(-r, +r)$. For each $k \in \mathbb{N}$, if the data word w_0 is of the following form:*



where $P_i \subseteq \mathsf{P}, d_i \geq s+(k+2)r, (i \geq 0)$, *and* w_1 *is defined by* $w_1 = w_0[1]$, *then Duplicator has a winning strategy on the game* $\mathrm{MG}_k^{\mathcal{I}}(w_0,0,w_1,0)$.

Proof. The proof is by induction on k. It is trivial when $k=0$. Suppose the statement holds for k, we must show that it also holds for $k+1$, *i.e.*, Duplicator has a winning strategy for the game $\mathrm{MG}_{k+1}^{\mathcal{I}}(w_0,0,w_1,0)$. We give the winning strategy for Duplicator as follows:

- $(w_l : 1, w_{(1-l)} : 1), (l \in \{0,1\})$. Then, by induction hypothesis, Duplicator has a winning strategy for the game $\mathrm{MG}_k^{\mathcal{I}}(w_0,1,w_1,1)$.
- $(w_0 : i, w_1 : i-1), (i \geq 2)$. Then by Lemma 2, Duplicator has a winning strategy for the game $\mathrm{MG}_k^{\mathcal{I}}(w_0,i,w_1,i-1)$. Moreover, for the second move of Spoiler in this round, if $(w_1 : j, w_0 : j+1)$, $(0 < j < i-1)$, by Lemma 2, Duplicator has a winning strategy for the game $\mathrm{MG}_k^{\mathcal{I}}(w_0,j+1,w_1,j)$.
- $(w_1 : i, w_0 : i+1), (i \geq 2)$. Then by Lemma 2, Duplicator has a winning strategy for the game $\mathrm{MG}_k^{\mathcal{I}}(w_0,i+1,w_1,i)$. Moreover, for the second move, if $(w_0 : 1, w_1 : 1)$, by induction hypothesis, Duplicator has a winning strategy for the game $\mathrm{MG}_k^{\mathcal{I}}(w_0,1,w_1,1)$. Otherwise, if $(w_0 : j, w_1 : j-1), (1 < j < i+1)$, by Lemma 2, Duplicator has a winning strategy for the game $\mathrm{MG}_k^{\mathcal{I}}(w_0,j,w_1,j-1)$.

This completes the proof.

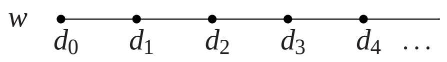
□

Theorem 3. *The problem, whether a given* TPTL*-formula is definable in* MTL*, is undecidable.*

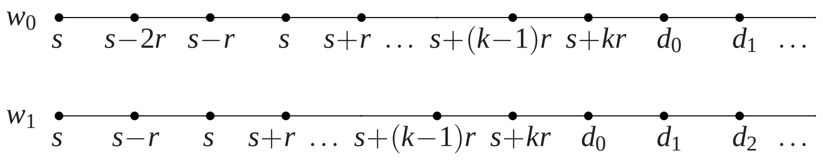
Proof. The recurrent state problem for two-counter machines is defined as follows: given a two-counter machine M, does there exist a computation of M that visits the initial instruction infinitely often? Alur and Henzinger showed that this problem is Σ_1^1-hard [3]. We reduce the recurrent state problem to the MTL definability decision problem in the following way: For each two-counter machine M, we construct a TPTL-formula ψ_M such that ψ_M is definable in MTL iff M is a negative instance of the recurrent state problem.

We use the fact that for each two-counter machine M there is a TPTL-formula φ_M which is satisfiable iff M is a positive instance of the recurrent state problem [3]. Define $\psi_M = (x.\mathsf{FFF}(x=0)) \wedge \mathsf{F}\varphi_M$. If φ_M is unsatisfiable, then ψ_M is definable by the MTL-formula `False`. Otherwise, if φ_M is satisfiable, we will prove that ψ_M is not definable in MTL. We show that for each $\mathcal{I} \in \mathsf{FCons}(\mathbb{Z})$ and $k \in \mathbb{N}$, there is no formula in $\mathsf{MTL}_k^{\mathcal{I}}$ that is equivalent to ψ_M.

For an arbitrary $\mathcal{I} \in \mathsf{FCons}(\mathbb{Z})$, let $r, s \in \mathbb{N}$ be such that all numbers in $\mathcal{I}$ are contained in $(-r,+r)$ and $s \geq 2r$. Suppose $k \geq 1$. By an exploration of the proof in [3] we can find that there is no propositional variable occurring in φ_M, and by Lemma 2, if a data word satisfies φ_M, then the new data word obtained by adding the same arbitrary value to every data value in the original word still satisfies φ_M. Hence we can assume that the data word w satisfying φ_M is of the form:



where $d_i \geq s+(k+1)r$ for each $i \geq 0$. We define the following two data words w_0 and w_1:



Clearly, $w_0 \models_{\mathsf{TPTL}} \psi_M$ and $w_1 \not\models_{\mathsf{TPTL}} \psi_M$. To show that there is no formula in $\mathsf{MTL}_k^{\mathcal{I}}$ that is equivalent to ψ_M, we prove that Duplicator has a winning strategy for the game $\mathrm{MG}_k^{\mathcal{I}}(w_0,0,w_1,0)$. The winning strategy for Duplicator in the first round is the same as the one that we give in the proof of Lemma 3. By Lemma 2 and 3, Duplicator can win the remaining rounds.

Since $\mathsf{MTL} = \bigcup_{\mathcal{I}\in\mathsf{FCons}(\mathbb{Z})}^{k\in\mathbb{N}} \mathsf{MTL}_k^{\mathcal{I}}$, we know by the argument given above that there is no formula in MTL that is equivalent to ψ_M if φ_M is satisfiable.

□

4.3 Effects on the Expressiveness of MTL by Restriction of syntactic Resources

We use the EF game for MTL to show the effects of restricting syntactic resources of MTL-formulas. We start with restrictions on the class of constraints occurring in an MTL-formula. For each $n \in \mathbb{Z}$, define $\varphi^n = \mathsf{F}_{[n,n]}\mathtt{True}$.

Lemma 4. *Let $\mathcal{I}_1, \mathcal{I}_2 \in \mathsf{Cons}(\mathbb{Z})$, for each $n \in \mathbb{Z}$, if $n \in \mathcal{I}_1$ and $n-1, n$ or $n, n+1$ are not in $\mathcal{I}_2$, then φ^n is definable in $\mathsf{MTL}^{\mathcal{I}_1}$ but not in $\mathsf{MTL}^{\mathcal{I}_2}$.*

Let $\mathcal{I}[n] = \{m \in \mathbb{Z} \mid m \le n\} \cup \{-\infty, +\infty\}$. The expressive power relation $\preccurlyeq$ defines a linear order on the set $\{\mathsf{MTL}^{\mathcal{I}[n]} \mid n \in \mathbb{Z}\}$ such that if $n_1 \le n_2$, then $\mathsf{MTL}^{\mathcal{I}[n_1]} \preccurlyeq \mathsf{MTL}^{\mathcal{I}[n_2]}$. We have $\mathsf{MTL} = \bigcup\{\mathsf{MTL}^{\mathcal{I}[n]} \mid n \in \mathbb{Z}\}$.

Proposition 2. (Linear Constraint Hierarchy of MTL)
For each $n_1, n_2 \in \mathbb{Z}$, if $n_1 < n_2$, then $\mathsf{MTL}^{\mathcal{I}[n_1]} \prec \mathsf{MTL}^{\mathcal{I}[n_2]}$.

In Proposition 2 we show that $\mathsf{MTL}^{\mathcal{I}[n+1]}$ is strictly more expressive than $\mathsf{MTL}^{\mathcal{I}[n]}$. Intuitively, if $\mathcal{I}_2$ is a proper subset of $\mathcal{I}_1$, one should expect that $\mathsf{MTL}^{\mathcal{I}_1}$ is more powerful than $\mathsf{MTL}^{\mathcal{I}_2}$. But in general this is not true. For example, $\mathsf{MTL}^{\mathcal{I}_1}$ with $\mathcal{I}_1 = \{-\infty, 0, 1, 2, +\infty\}$ has the same expressive power as $\mathsf{MTL}^{\mathcal{I}_2}$ where $\mathcal{I}_2 = \mathcal{I}_1 \setminus \{1\}$, since we can use 0 and 2 to express constraints that use the constant 1. It is natural to ask, for $\mathcal{I} \in \mathsf{Cons}(\mathbb{Z})$, what is the minimal subset $\mathcal{I}'$ of $\mathcal{I}$ such that $\mathsf{MTL}^{\mathcal{I}'} \equiv \mathsf{MTL}^{\mathcal{I}}$. In the following we give another constraint hierarchy.

Let EVEN be the subset of $\mathsf{Cons}(\mathbb{Z})$ where only even numbers are in consideration. Let $\mathbf{even} \in \mathsf{EVEN}$ be the set that contains all even numbers. It is easily seen that $\mathsf{MTL}^{\mathbf{even}} \equiv \mathsf{MTL}$. Given $\mathcal{I}_1, \mathcal{I}_2 \in \mathsf{EVEN}$, if $\mathcal{I}_1 \subsetneq \mathcal{I}_2$, by Lemma 4, we have $\mathsf{MTL}^{\mathcal{I}_1} \prec \mathsf{MTL}^{\mathcal{I}_2}$. The expressive power relation $\preccurlyeq$ defines a partial order on the set $\{\mathsf{MTL}^{\mathcal{I}} \mid \mathcal{I} \in \mathsf{EVEN}\}$.

Proposition 3. (Lattice Constraint Hierarchy of MTL)
$\langle \{\mathsf{MTL}^{\mathcal{I}} \mid \mathcal{I} \in \mathsf{EVEN}\}, \preccurlyeq \rangle$ *constitutes a complete lattice in which*

(i) *the greatest element is* $\mathsf{MTL}^{\mathbf{even}}$,

(ii) *the least element is* $\mathsf{MTL}^{\{-\infty,+\infty\}}$,

and for each nonempty subset $S \subseteq \mathsf{EVEN}$,

(iii) $\bigwedge_{\mathcal{I}\in S} \mathsf{MTL}^{\mathcal{I}} = \mathsf{MTL}^{\bigcap_{\mathcal{I}\in S}\mathcal{I}}$,

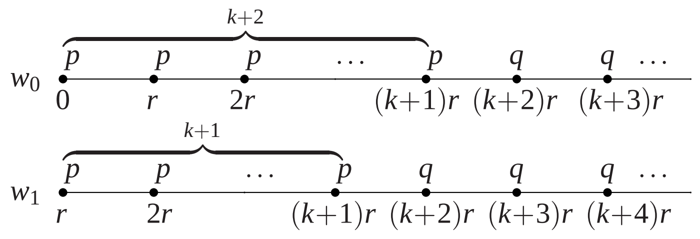
(iv) $\bigvee_{\mathcal{I}\in S} \mathsf{MTL}^{\mathcal{I}} = \mathsf{MTL}^{\bigcup_{\mathcal{I}\in S}\mathcal{I}}$.

Note that $\langle \{\mathsf{MTL}^{\mathcal{I}} \mid \mathcal{I} \in \mathsf{EVEN}\}, \preccurlyeq \rangle$ is isomorphic to the complete lattice $\langle \mathcal{P}(X), \subseteq \rangle$, where X is a countable infinite set, $\mathcal{P}(X)$ is the powerset of X and $\subseteq$ is the containment relation.

Next we show that, as for LTL [14], there is a strict until hierarchy for MTL.

Proposition 4. *For all $k \in \mathbb{N}$, MTL_{k+1} is strictly more expressive than MTL_k.*

Proof. Define $\varphi[1] = (p \wedge \mathsf{X}p)$ and $\varphi[k+1] = (p \wedge \mathsf{X}\varphi[k])$ for every $k \geq 1$. Note that for each $k \geq 1$, $\varphi[k] \in \mathsf{MTL}_k$. We show that for each $\mathcal{I} \in \mathsf{FCons}(\mathbb{Z}), k \geq 0$, $\varphi[k+1]$ is not definable in $\mathsf{MTL}_k^{\mathcal{I}}$. Let $r \in \mathbb{N}$ be such that all numbers in $\mathcal{I}$ are contained in $(-r, +r)$. Define two data words w_0 and w_1 as follows:



We see that $w_0 \models_{\mathsf{MTL}} \varphi[k+1]$ and $w_1 \not\models_{\mathsf{MTL}} \varphi[k+1]$. By Lemma 3 and Theorem 2, there is no formula in $\mathsf{MTL}_k^{\mathcal{I}}$ that is equivalent to $\varphi[k+1]$. Since $\mathsf{MTL}_k = \bigcup_{\mathcal{I} \in \mathsf{FCons}(\mathbb{Z})} \mathsf{MTL}_k^{\mathcal{I}}$, $\varphi[k+1]$ is not definable in MTL_k.

□

As for the MTL definability decision problem, we can show that the MTL_k definability decision problem which asks whether the data language defined by an MTL_{k+1}-formula is definable in MTL_k is undecidable. As a corollary, we know that whether an MTL-formula is equivalent to an MTL_k-formula is undecidable.

Proposition 5. *There exists $m \in \mathbb{N}$ such that for every $k \geq m$, the problem whether a formula $\varphi \in \mathsf{MTL}_{k+1}$ is definable in MTL_k is undecidable.*

5 The Ehrenfeucht-Fraïssé Game for TPTL

In Proposition 1 we have proved that there is an $\mathsf{UnaFreezeLTL}^1$-formula that is not definable in MTL, and we concluded that TPTL^1 is strictly more expressive than MTL. A natural question is to ask for the relation between MTL, $\mathsf{UnaTPTL}$ and $\mathsf{FreezeLTL}$. For this, we define the EF game for TPTL.

The *until rank* of a TPTL-formula φ, denoted by $\mathsf{Urk}(\varphi)$, is defined analogously to that of MTL-formulas in Sect. 2.4; we additionally define $\mathsf{Urk}(x \in I) = 0$ and $\mathsf{Urk}(x.\varphi) = \mathsf{Urk}(\varphi)$. Let $\mathcal{I} \in \mathsf{Cons}(\mathbb{Z}), k \geq 0, n \geq 1$, we define

$$\mathsf{TPTL}^{\mathcal{I}} = \{\varphi \in \mathsf{TPTL} \mid \text{ for each subformula } x \in I \text{ of } \varphi, \text{ the endpoints of } I \text{ belong to } \mathcal{I}\},$$
$$\mathsf{TPTL}^{n} = \{\varphi \in \mathsf{TPTL} \mid \text{the register variables in } \varphi \text{ are from } \{x_1, \ldots, x_n\}\},$$
$$\mathsf{TPTL}_k = \{\varphi \in \mathsf{TPTL} \mid \mathsf{Urk}(\varphi) \leq k\}, \quad \mathsf{TPTL}_k^{n,\mathcal{I}} = \mathsf{TPTL}^n \cap \mathsf{TPTL}^{\mathcal{I}} \cap \mathsf{TPTL}_k.$$

Lemma 5. *For each $\mathcal{I} \in \mathsf{FCons}(\mathbb{Z})$, $n \geq 1$ and $k \geq 0$, there are only finitely many formulas in $\mathsf{TPTL}_k^{n,\mathcal{I}}$ up to equivalence.*

Let w_0, w_1 be two data words, and $i_0, i_1 \geq 0$ be positions in w_0, w_1, respectively, and ν_0, ν_1 be two register valuations. We say that (w_0, i_0, ν_0) and (w_1, i_1, ν_1) are $\mathsf{TPTL}_k^{n,\mathcal{I}}$*-equivalent*, written $(w_0, i_0, \nu_0) \equiv_k^{n,\mathcal{I}} (w_1, i_1, \nu_1)$, if for each formula $\varphi \in \mathsf{TPTL}_k^{n,\mathcal{I}}$, $(w_0, i_0, \nu_0) \models_{\mathsf{TPTL}} \varphi$ iff $(w_1, i_1, \nu_1) \models_{\mathsf{TPTL}} \varphi$.

The k-round TPTL EF game on (w_0, i_0, ν_0) and (w_1, i_1, ν_1) with respect to n and $\mathcal{I}$, denoted by $\mathrm{TG}_k^{n,\mathcal{I}}(w_0, i_0, \nu_0, w_1, i_1, \nu_1)$, is played by Spoiler and Duplicator on w_0 and w_1 starting from i_0 in w_0 with valuation ν_0 and i_1 in w_1 with valuation ν_1.

We say that (i_0, ν_0) and (i_1, ν_1) *agree in the atomic formulas in* $\mathsf{TPTL}^{n,\mathcal{I}}$, if $(w_0, i_0, \nu_0) \models_{\mathsf{TPTL}} p$ iff $(w_1, i_1, \nu_1) \models_{\mathsf{TPTL}} p$ for for each $p \in \mathsf{P}$, and $(w_0, i_0, \nu_0) \models_{\mathsf{TPTL}} x \in I$ iff $(w_1, i_1, \nu_1) \models_{\mathsf{TPTL}} x \in I$ for each formula $x \in I$ in $\mathsf{TPTL}^{n,\mathcal{I}}$.

Analogously to the EF game for MTL, $\mathrm{TG}_k^{n,\mathcal{I}}(w_0, i_0, \nu_0, w_1, i_1, \nu_1)$ is defined inductively. If $k = 0$, then Spoiler wins if (i_0, ν_0) and (i_1, ν_1) do not agree in the atomic formulas in $\mathsf{TPTL}^{n,\mathcal{I}}$. Otherwise, Duplicator wins. Suppose $k > 0$, in the first round,

1. Spoiler wins this round if (i_0, ν_0) and (i_1, ν_1) do not agree in the atomic formulas in $\mathsf{TPTL}^{n,\mathcal{I}}$. Otherwise, Spoiler chooses a subset Y (maybe empty) of $\{x_1, \ldots, x_n\}$ and sets $\nu_l' = \nu_l[x := d_{i_l}(x \in Y)]$ for all $l \in \{0,1\}$. Then Spoiler chooses a word w_l for some $l \in \{0,1\}$ and a position $i_l' > i_l$ in w_l.
2. Then Duplicator tries to choose a position $i'_{(1-l)} > i_{(1-l)}$ in $w_{(1-l)}$ such that (i_0', ν_0') and (i_1', ν_1') agree in the atomic formulas in $\mathsf{TPTL}^{n,\mathcal{I}}$. If Duplicator fails, then Spoiler wins this round.
3. Then, Spoiler has two options: either he chooses to start a new game $\mathrm{TG}_{k-1}^{n,\mathcal{I}}(w_0, i_0', \nu_0', w_1, i_1', \nu_1')$; or
4. Spoiler chooses a position $i_{(1-l)} < i''_{(1-l)} < i'_{(1-l)}$ in $w_{(1-l)}$. Then Duplicator tries to respond by choosing a position $i_l < i_l'' < i_l'$ in w_l such that (i_0'', ν_0') and (i_1'', ν_1') agree in the atomic formulas in $\mathsf{TPTL}^{n,\mathcal{I}}$. If Duplicator fails to do so, Spoiler wins this round.
5. If Spoiler cannot win in Step 1, 2 or 4, then Duplicator wins this round. Then Spoiler chooses to start a new game $\mathrm{TG}_{k-1}^{n,\mathcal{I}}(w_0, i_0'', \nu_0', w_1, i_1'', \nu_1')$.

If Duplicator has a winning strategy for the game $\mathrm{TG}_k^{n,\mathcal{I}}(w_0, i_0, \nu_0, w_1, i_1, \nu_1)$, then we denote it by $(w_0, i_0, \nu_0) \sim_k^{n,\mathcal{I}} (w_1, i_1, \nu_1)$.

Theorem 4. *For each* $\mathcal{I} \in \mathsf{FCons}(\mathbb{Z})$, $n \geq 1, k \geq 0$, $(w_0, i_0, \nu_0) \equiv_k^{n,\mathcal{I}} (w_1, i_1, \nu_1)$ *if and only if* $(w_0, i_0, \nu_0) \sim_k^{n,\mathcal{I}} (w_1, i_1, \nu_1)$.

Theorem 5. *Let* $\mathbf{L}$ *be a data language. For each* $\mathcal{I} \in \mathsf{FCons}(\mathbb{Z})$, $n \geq 1$ *and* $k \geq 0$, *the following are equivalent:*

1. $\mathbf{L}$ *is not definable in* $\mathsf{TPTL}_k^{n,\mathcal{I}}$.
2. *There exist* $w_0 \in \mathbf{L}$ *and* $w_1 \notin \mathbf{L}$ *such that* $(w_0, 0, \overline{0}) \sim_k^{n,\mathcal{I}} (w_1, 0, \overline{0})$.

5.1 More on the Relative Expressiveness of MTL and TPTL

We are going to compare MTL with two fragments of TPTL, namely the unary fragment UnaTPTL and the fragment FreezeLTL. Using the EF game for TPTL we can prove the following results:

Proposition 6. *The* MTL-*formula* $\mathsf{F}_{=1}\mathtt{True}$ *is not definable in* FreezeLTL.

Proposition 7. *The* MTL-*formula* $(\neg a)\mathsf{U}b$ *is not definable in* UnaTPTL.

We remark that for these results, we have to slightly change the definition of the games to suit to the fragments $\mathsf{FreezeLTL}$ and $\mathsf{UnaTPTL}$ such that an analogous version of Theorem 4 holds. The preceding propositions yield another interesting result for MTL and these two fragments of TPTL.

Corollary 2. *1.* MTL *and* $\mathsf{FreezeLTL}$ *are incomparable.*

2. MTL *and* $\mathsf{UnaTPTL}$ *are incomparable.*

3. $\mathsf{UnaTPTL}$ *and* $\mathsf{FreezeLTL}$ *are incomparable.*

Analogously to Theorem 3, we can prove that the $\mathsf{FreezeLTL}$ (resp., $\mathsf{UnaTPTL}$) definability problem is undecidable.

Proposition 8. *The problem, whether a given* TPTL*-formula is definable in* $\mathsf{FreezeLTL}$ *(resp.,* $\mathsf{UnaTPTL}$*), is undecidable.*

5.2 Restricting Resources in TPTL

In the following we prove results on the effects of restricting syntactic resources of TPTL-formulas similar to those for MTL. For each $n \in \mathbb{Z}$, we redefine $\varphi^n = x.\mathsf{F}(x = n)$.

Lemma 6. *Let* $\mathcal{I}_1, \mathcal{I}_2 \in \mathsf{Cons}(\mathbb{Z})$*, for each* $n \in \mathbb{Z}$*, if* $n \in \mathcal{I}_1$ *and* $n-1, n$ *or* $n, n+1$ *are not in* $\mathcal{I}_2$*, then* φ^n *is definable in* $\mathsf{TPTL}^{\mathcal{I}_1}$ *but not in* $\mathsf{TPTL}^{\mathcal{I}_2}$*.*

Using this lemma we can prove the following two propositions.

Proposition 9. (Linear Constraint Hierarchy of TPTL)
The expressive power relation $\preccurlyeq$ *defines a linear order on the set* $\{\mathsf{TPTL}^{\mathcal{I}[n]} \mid n \in \mathbb{Z}\}$ *such that if* $n_1 \leq n_2$*, then* $\mathsf{TPTL}^{\mathcal{I}[n_1]} \preccurlyeq \mathsf{TPTL}^{\mathcal{I}[n_2]}$ *. Moreover, if* $n_1 < n_2$*, then* $\mathsf{TPTL}^{\mathcal{I}[n_1]} \prec \mathsf{TPTL}^{\mathcal{I}[n_2]}$*.*

Proposition 10. (Lattice Constraint Hierarchy of TPTL)
$\langle\{\mathsf{TPTL}^{\mathcal{I}} \mid \mathcal{I} \in \mathsf{EVEN}\}, \preccurlyeq\rangle$ *constitutes a complete lattice in which*

(i) the greatest element is $\mathsf{TPTL}^{\mathbf{even}} (\equiv \mathsf{TPTL})$*,*

(ii) the least element is $\mathsf{TPTL}^{\{-\infty,+\infty\}} (\equiv \mathsf{LTL})$*,*

and for each nonempty subset $S \subseteq \mathsf{EVEN}$*,*

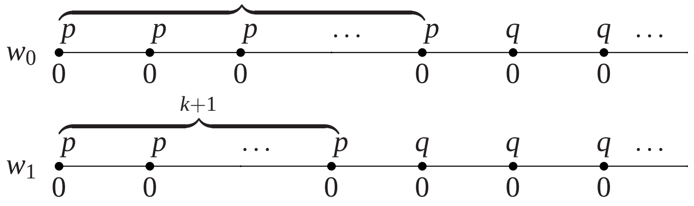
(iii) $\bigwedge_{\mathcal{I} \in S} \mathsf{TPTL}^{\mathcal{I}} = \mathsf{TPTL}^{\bigcap_{\mathcal{I} \in S} \mathcal{I}}$*,*

(iv) $\bigvee_{\mathcal{I} \in S} \mathsf{TPTL}^{\mathcal{I}} = \mathsf{TPTL}^{\bigcup_{\mathcal{I} \in S} \mathcal{I}}$*.*

In the next proposition we show that the until hierarchy for TPTL is strict.

Proposition 11. TPTL_{k+1} *is strictly more expressive than* TPTL_k*.*

Proof. Let $\varphi[k]\,(k \geq 1)$ be as defined in Proposition 4. $\varphi[k]$ is a formula in TPTL_k. For every $k \geq 0$. We can show that $(w_0, 0, \overline{0}) \sim_k^{n,\mathcal{I}} (w_1, 0, \overline{0})$ on the following two data words $w_0 \models_{\mathsf{TPTL}} \varphi[k+1]$ and $w_1 \not\models_{\mathsf{TPTL}} \varphi[k+1]$.



□

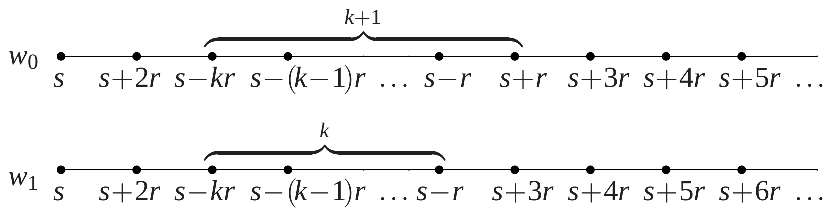
Corollary 3. MTL_{k+1} *and* TPTL_k *are incomparable in expressive power.*

Proposition 12. *There exists $m \in \mathbb{N}$ such that for every $k \geq m$, the problem whether a formula $\varphi \in \mathsf{TPTL}_{k+1}$ is definable in TPTL_k is undecidable.*

We have seen in the previous chapters that TPTL is strictly more expressive than MTL. The register variables play a crucial role in reaching this greater expressiveness. In the following we want to explore more deeply whether the number of register variables allowed in a TPTL formula has an impact on the expressive power of the logic. We are able to show that there is a strict increase in expressiveness when allowing two register variables instead of just one. The following results concern the number of register variables allowed in a TPTL-formula.

Proposition 13. *For the* $\mathsf{UnaTPTL}^2$*-formula* $\varphi = x_1.\mathsf{F}(x_1 > 0 \wedge x_2.\mathsf{F}(x_1 > 0 \wedge x_2 < 0))$ *there is no equivalent formula in* TPTL^1.

Proof. Let $\mathcal{I} \in \mathsf{FCons}(\mathbb{Z})$ and $k \geq 1$. Let $s, r \in \mathbb{N}$ be such that all elements in $\mathcal{I}$ are contained in $(-r, +r)$ and $s - kr \geq 0$. One can show that $(w_0, 0, \overline{0}) \sim_k^{1,\mathcal{I}} (w_1, 0, \overline{0})$ on the following two data words $w_0 \models_{\mathsf{TPTL}} \varphi$ and $w_1 \not\models_{\mathsf{TPTL}} \varphi$.



□

Corollary 4. TPTL^2 *is strictly more expressive than* TPTL^1.

It remains open whether we can generalize this result to TPTL^{n+1} and TPTL^n, where $n \geq 2$, to get a complete hierarchy for the number of register variables. We have the following conjecture.

Conjecture 1. *For each $n \geq 1$,* TPTL^{n+1} *is strictly more expressive than* TPTL^n.

6 Conclusion and Future Work

In this paper, we consider the expressive power of MTL and TPTL on non-monotonic ω-data words and introduce EF games for these two logics. We show that TPTL is strictly more expressive than MTL and some other expressiveness results of various syntactic restrictions. For TPTL, we examine the effects of allowing only a bounded number of register variables: We prove that TPTL^2 is strictly more expressive than TPTL^1, but it is still open if TPTL^{n+1} is strictly more expressive than TPTL^n for all $n \geq 1$ (Conjecture 1). In future work we want to figure out whether there is a decidable characterization for the set of data domains for which TPTL and MTL are equally expressive.

References

[1] Rajeev Alur & David L. Dill (1994): *A Theory of Timed Automata*. *Theor. Comput. Sci.* 126(2), pp. 183–235. Available at `http://dx.doi.org/10.1016/0304-3975(94)90010-8`.

[2] Rajeev Alur & Thomas A. Henzinger (1993): *Real-Time Logics: Complexity and Expressiveness*. *Inf. Comput.* 104(1), pp. 35–77. Available at `http://dx.doi.org/10.1006/inco.1993.1025`.

[3] Rajeev Alur & Thomas A. Henzinger (1994): *A Really Temporal Logic*. *J. ACM* 41(1), pp. 181–204. Available at `http://doi.acm.org/10.1145/174644.174651`.

[4] Mikolaj Bojanczyk, Claire David, Anca Muscholl, Thomas Schwentick & Luc Segoufin (2011): *Two-variable logic on data words*. *ACM Trans. Comput. Log.* 12(4), p. 27. Available at `http://doi.acm.org/10.1145/1970398.1970403`.

[5] Benedikt Bollig (2011): *An Automaton over Data Words That Captures EMSO Logic*. In Katoen & König [15], pp. 171–186. Available at `http://dx.doi.org/10.1007/978-3-642-23217-6_12`.

[6] Benedikt Bollig, Aiswarya Cyriac, Paul Gastin & K. Narayan Kumar (2012): *Model Checking Languages of Data Words*. In Lars Birkedal, editor: *FoSSaCS*, *LNCS* 7213, Springer, pp. 391–405. Available at `http://dx.doi.org/10.1007/978-3-642-28729-9_26`.

[7] Patricia Bouyer (2002): *A logical characterization of data languages*. *Inf. Process. Lett.* 84(2), pp. 75–85. Available at `http://dx.doi.org/10.1016/S0020-0190(02)00229-6`.

[8] Patricia Bouyer, Fabrice Chevalier & Nicolas Markey (2010): *On the expressiveness of TPTL and MTL*. *Inf. Comput.* 208(2), pp. 97–116. Available at `http://dx.doi.org/10.1016/j.ic.2009.10.004`.

[9] Claudia Carapelle, Shiguang Feng, Oliver Fernandez Gil & Karin Quaas (2014): *Satisfiability for MTL and TPTL over Non-monotonic Data Words*. In: *LATA*, pp. 248–259. Available at `http://dx.doi.org/10.1007/978-3-319-04921-2_20`.

[10] Stéphane Demri & Ranko Lazic (2009): *LTL with the freeze quantifier and register automata*. *ACM Trans. Comput. Log.* 10(3). Available at `http://doi.acm.org/10.1145/1507244.1507246`.

[11] Stéphane Demri, Ranko Lazic & David Nowak (2007): *On the freeze quantifier in Constraint LTL: Decidability and complexity*. *Inf. Comput.* 205(1), pp. 2–24. Available at `http://dx.doi.org/10.1016/j.ic.2006.08.003`.

[12] Stéphane Demri, Ranko Lazic & Arnaud Sangnier (2008): *Model Checking Freeze LTL over One-Counter Automata*. In Roberto M. Amadio, editor: *FoSSaCS*, *LNCS* 4962, Springer, pp. 490–504. Available at `http://dx.doi.org/10.1007/978-3-540-78499-9_34`.

[13] Stéphane Demri & Arnaud Sangnier (2010): *When Model-Checking Freeze LTL over Counter Machines Becomes Decidable*. In C.-H. Luke Ong, editor: *FOSSACS*, *LNCS* 6014, Springer, pp. 176–190. Available at `http://dx.doi.org/10.1007/978-3-642-12032-9_13`.

[14] Kousha Etessami & Thomas Wilke (1996): *An Until Hierarchy for Temporal Logic*. In: *LICS*, IEEE Computer Society, pp. 108–117. Available at `http://doi.ieeecomputersociety.org/10.1109/LICS.1996.561310`.

[15] Joost-Pieter Katoen & Barbara König, editors (2011): *CONCUR 2011 - Concurrency Theory - 22nd International Conference, CONCUR 2011, Aachen, Germany, September 6-9, 2011. Proceedings*. *LNCS* 6901, Springer. Available at `http://dx.doi.org/10.1007/978-3-642-23217-6`.

[16] Ron Koymans (1990): *Specifying Real-Time Properties with Metric Temporal Logic*. *Real-Time Systems* 2(4), pp. 255–299. Available at `http://dx.doi.org/10.1007/BF01995674`.

[17] Paritosh K. Pandya & Simoni S. Shah (2011): *On Expressive Powers of Timed Logics: Comparing Boundedness, Non-punctuality, and Deterministic Freezing*. In Katoen & König [15], pp. 60–75. Available at `http://dx.doi.org/10.1007/978-3-642-23217-6_5`.

[18] Luc Segoufin (2006): *Automata and Logics for Words and Trees over an Infinite Alphabet*. In Zoltán Ésik, editor: *CSL*, *LNCS* 4207, Springer, pp. 41–57. Available at `http://dx.doi.org/10.1007/11874683_3`.

On Determinism and Unambiguity of Weighted Two-way Automata

Vincent Carnino
LIGM - Laboratoire d'informatique Gaspard-Monge
Université Paris-Est Marne-la-Vallée, France
Vincent.Carnino@univ-mlv.fr

Sylvain Lombardy
LaBRI - Laboratoire Bordelais de Recherche en Informatique
Institut Polytechnique de Bordeaux, France
Sylvain.Lombardy@labri.fr

In this paper, we first study the conversion of weighted two-way automata to one-way automata. We show that this conversion preserves the unambiguity but does not preserve the determinism. Yet, we prove that the conversion of an unambiguous weighted one-way automaton into a two-way automaton leads to a deterministic two-way automaton. As a consequence, we prove that unambiguous weighted two-way automata are equivalent to deterministic weighted two-way automata in commutative semirings.

1 Introduction

A classical question in automata theory concerns the expressive power of a device and especially the difference between one-way devices and two-way devices. It is well known that two-way automata may be reduced to one-way automata and therefore recognize the same language family [14, 12].

In this paper, we deal with the weighted versions of these two devices. We describe the conversion of a two-way automaton over a commutative sering into a one-way automaton. Such an algorithm has already be stated in [1]; our construction is close, but we are mainly interested here in proving that this conversion preserves the unambiguity of automata; it does not preserve the determinism.

We then present a construction for the conversion of any unambiguous one-way automaton into a deterministic two-way automaton; this part does not require that the semiring is commutative.

A consequence of these two procedure is that, on commutative semirings, opposite to the case of one-way automata, unambiguous two-way automata are not more powerful than deterministic ones.

2 Weighted Two-way Automata

2.1 Automata and runs

A semiring $\mathbb{K}$ is a set endowed with two binary associative operations, $\oplus$ and $\otimes$, such that $\oplus$ is commutative and $\otimes$ distributes over $\oplus$. The set $\mathbb{K}$ contains two particular elements, $0_{\mathbb{K}}$ and $1_{\mathbb{K}}$ that are respectively neutral for $\oplus$ and $\otimes$; moreover, $0_{\mathbb{K}}$ is an annihilator for $\otimes$.

For every alphabet A, we assume that there exist two fresh symbols $\vdash$ and $\dashv$ that are marks at the beginning and the end of the tapes of automata. We denote $A_{\vdash\dashv}$ the alphabet $A \cup \{\vdash, \dashv\}$. For every word w in A, $w_{\vdash\dashv}$ is the word in $A_{\vdash\dashv}$ equal to $\vdash w \dashv$.

One-way and two-way $\mathbb{K}$-automata share a part of their definition. A $\mathbb{K}$-automaton is a tuple $\mathscr{A} = (Q, A, E, I, T)$ where Q is a finite set of states, A is a finite alphabet, and I and T are partial functions from Q to $\mathbb{K}$. The support of I, $\underline{I}$, is the set of initial states of $\mathscr{A}$, and the support of T, $\underline{T}$, is the set of final states of $\mathscr{A}$.

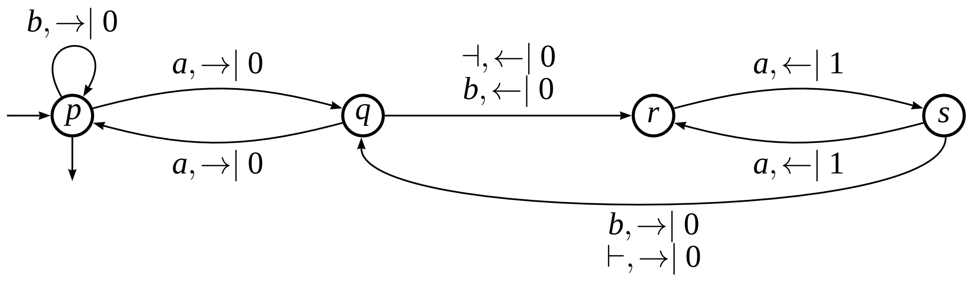


Figure 1: The two-way $\mathscr{N}$-automaton $\mathscr{A}_1$.

The definition of transitions differ. In a two-way $\mathbb{K}$-automaton, E is a partial function from $Q \times (A_{\vdash\dashv} \times \{-1, +1\}) \times Q$ into $\mathbb{K}$ and the support of E, $\underline{E}$, is the set of transitions of $\mathscr{A}$. Moreover, the intersection of $\underline{E}$ and $Q \times (\{\vdash\} \times \{-1\} \cup \{\dashv\} \times \{1\}) \times Q$ must be empty.
Let t be a transition in $\underline{E}$; if $t = (p, a, d, q)$, we denote $\sigma(t) = p$, $\tau(t) = q$, $\lambda(t) = a$, $\delta(t) = d$. On figures, the value of δ is represented by a left (-1) or right (+1) arrow. For instance, if $t = (p, a, -1, q)$ and $E_t = k$, we draw $p \xrightarrow{a, \leftarrow | k} q$.

In a one-way $\mathbb{K}$-automaton, E is a partial function from $Q \times A \times Q$ into $\mathbb{K}$, and the support of E, $\underline{E}$, is the set of transitions of $\mathscr{A}$.
Let t be a transition in $\underline{E}$; if $t = (p, a, q)$, we denote $\sigma(t) = p$, $\tau(t) = q$, $\lambda(t) = a$.

Example 1. *Let $\mathscr{A}_1$ be the two-way $\mathscr{N}$-automaton of Figure 1, where $\mathscr{N} = (\mathbb{N} \cup \{\infty\}, \min, +)$ is the tropical semiring; since the multiplication law in this semiring is the usual sum, the weight of a path in this automaton is the sum of the weights of its transitions. This automaton is deterministic (cf. Definition 9) and thus there is only one computation for each accepted word. The behaviour of this automaton is quite easy. For each block of $'a'$ it checks through a left-right reading, whether the length of the block is odd; if it is, a right-left reading computes the length of the block; otherwise the automaton goes to the next block of $'a'$.*

Definition 1. *Let $w = w_1 \ldots w_n$ be a word of A^*, we set $w_0 = \vdash$ and $w_{n+1} = \dashv$. A* configuration *of $\mathscr{A}$ on w is a pair (p, i) where i is in $[0; n+1]$ and p is a state of $\mathscr{A}$. A computation (or run) ρ of $\mathscr{A}$ on w is a finite sequence of configurations $((p_0, i_0), \ldots, (p_k, i_k))$ such that :*

- *$i_0 = 1$, $i_k = n+1$, p_0 is in $\underline{I}$ and p_k is in $\underline{T}$;*
- *for every j in $[0; k-1]$, there exists t_j, such that*
$$\sigma(t_j) = p_j,\ \tau(t_j) = p_{j+1},\ \lambda(t_j) = a_{i_j},\ \text{and } i_{j+1} = i_j + \delta(t_j).$$

The weight of such a computation, denoted by $|\rho|$, is $I(p_0) \otimes \bigotimes_{j=0}^{k-1} E(t_j) \otimes T(p_k)$. The weight of w in $\mathscr{A}$, denoted by $\langle |\mathscr{A}|, w \rangle$, is the addition of the weights of all the runs with label w in $\mathscr{A}$. Notice that there may be an infinite number of computations with the same label w. The definition of the behaviour of $\mathscr{A}$ in this case requires to study the definition of infinite sums. This can be done, like for one-way $\mathbb{K}$-automata with ε-transitions, for instance with complete semirings or topological semirings [11]. This is not the purpose of this paper, since we mainly deal with two-way automata where the number of computations is finite for every word.

Example 2. *A run of the $\mathscr{N}$-automaton $\mathscr{A}_1$ over the word abaaba is represented on Figure 2. The weight of this run is equal to* 2.

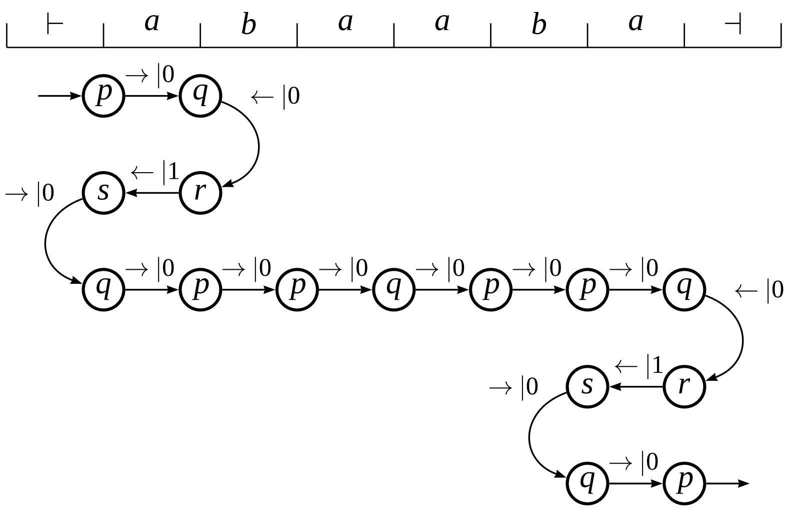


Figure 2: A run of $\mathscr{A}_1$ over the word *abaaba*.

Definition 2. *Let $\rho = ((p_0,i_0),\ldots,(p_k,i_k))$ be a run over w. If there exists m,n in $[1,k]$, with $m < n$ such that $(p_m,i_m) = (p_n,i_n)$, then we say that $((p_m,i_m),\ldots,(p_n,i_n))$ is an* unmoving circuit *of ρ. If ρ does not contain any unmoving circuit, it is* reduced.

Lemma 1. *If a two-way $\mathbb{K}$-automaton admits a run ρ which is not reduced, it admits a reduced run with the same label.*

Proof. We consider a shortest non reduced run $\rho = ((p_0,i_0),\ldots,(p_m,i_m),\ldots,(p_n,i_n),\ldots,(t_k,i_k))$, with $(p_m,i_m) = (p_n,i_n)$.
Then $((p_0,i_0),\ldots,(p_{m-1},i_{m-1}),(p_n,i_n),\ldots,(p_k,i_k))$ is a run; by minimality of ρ, this run is reduced. ☐

Definition 3. *A one-way or two-way automaton $\mathscr{A}$ is* unambiguous *if every word labels at most one computation.*

Unambiguous automata have obviously only reduced computations.

2.2 Coverings

We extend here the notion of covering (*cf.* [13]) to two-way automata.

Definition 4. *Let $\mathscr{A} = (Q,A,E,I,T)$ and $\mathscr{B} = (R,A,F,J,U)$ be two weighted two-way automata. A mapping φ from Q into R is a* morphism *if,*
i) $\forall p \in \underline{I}$, $J(\varphi(p)) = I(p)$;
ii) $\forall p \in \underline{T}$, $U(\varphi(p)) = T(p)$;
iii) $\forall t = (p,a,\delta,q) \in \underline{E}$, $\tilde{\varphi}(t) = (\varphi(p),a,\delta,\varphi(q)) \in \underline{F}$ and $F(\tilde{\varphi}(t')) = E(t)$.
The morphism is surjective if $\varphi(Q) = R$, $\varphi(\underline{I}) = \underline{J}$, $\varphi(\underline{T}) = \underline{U}$, and $\tilde{\varphi}(\underline{E}) = \underline{F}$.

Definition 5. *Let $\mathscr{A} = (Q,A,E,I,T)$ and $\mathscr{B} = (R,A,F,J,U)$ be two weighted two-way automata. $\mathscr{A}$ is a* covering *of $\mathscr{B}$ if there exists a surjective morphism φ from $\mathscr{A}$ onto $\mathscr{B}$ such that*

$$i)\ \forall r \in \underline{U},\ \varphi^{-1}(r) \subseteq \underline{T} \qquad ii)\ \forall r \in \underline{J},\ \exists! p \in \varphi^{-1}(r) \cap \underline{I}$$
$$iii)\ \forall t \in \underline{F}, \forall p \in \varphi^{-1}(\sigma(t)), \exists! t' \in \tilde{\varphi}^{-1}(t), \sigma(t') = p.$$

$\mathscr{A}$ is an in-covering *of $\mathscr{B}$ is there exists a surjective morphism φ from $\mathscr{A}$ onto $\mathscr{B}$ such that*

$$i)\ \forall r \in \underline{J},\ \varphi^{-1}(r) \subseteq \underline{I} \qquad ii)\ \forall r \in \underline{U},\ \exists! p \in \varphi^{-1}(r) \cap \underline{T}$$
$$iii)\ \forall t \in \underline{F}, \forall q \in \varphi^{-1}(\tau(t)), \exists! t' \in \tilde{\varphi}^{-1}(t), \tau(t') = q.$$

Proposition 1. *Let $\mathscr{A}$ and $\mathscr{B}$ be two weighted two-way automata. If $\mathscr{A}$ is a covering (resp. an in-covering) of $\mathscr{B}$, the corresponding morphism φ induces a bijection between computations of $\mathscr{A}$ and $\mathscr{B}$ such that every computation of $\mathscr{A}$ and its image in $\mathscr{B}$ have the same label and the same weight.*

Proof. Assume that $\mathscr{A}$ is a covering of $\mathscr{B}$. Let w be a word and let $((p_0,i_0),\ldots,(p_k,i_k))$ be a computation on w in $\mathscr{A}$. For every j in $[0;k]$, we set $r_j = \varphi(p_j)$; by definition of a morphism $((r_0,i_0),\ldots,(r_k,i_k))$ is a computation on w in $\mathscr{B}$ with the same weight. Conversely, let $((r_0,i_0),\ldots,(r_k,i_k))$ be a computation in $\mathscr{B}$. Let p_0 be the unique initial state in $\varphi^{-1}(r_0)$.For every j in $[0;k-1]$, let $\delta_j = i_{j+1} - i_j$; the configuration (r_{j+1}, r_{j+1}) is reached from configuration (r_j, i_j) through the transition $(r_j, w_j, \delta_j, r_{j+1})$; inductively, we define p_{j+1} as the unique state in $\varphi^{-1}(r_{j+1})$ such that $(p_j, w_j, \delta_j, p_{j+1})$ is a transition of $\mathscr{A}$. Then, $((p_0,i_0),\ldots,(p_k,i_k))$ is a computation on w in $\mathscr{A}$. Hence, every computation ρ of $\mathscr{B}$ is lift up in a unique way into a computation of $\mathscr{A}$ whose image by φ is ρ.

The proof is similar for in-coverings. □

This proposition implies that a two-way automaton and its covering (*resp.* in-covering) are equivalent; moreover, if a two-way automaton is unambiguous, so is every of its (in-)coverings.

2.3 δ-Locality

Definition 6. *Let $\mathscr{A}$ be a two-way $\mathbb{K}$-automaton. If, for each state p of $\mathscr{A}$, every transition outgoing from p has the same direction, then $\mathscr{A}$ is δ-local.*

If Q is the set of states of a two-way $\mathbb{K}$-automaton, we denote Q_+ (*resp.* Q_-) the set of states p such that, for every transition t outgoing from p, $\delta(t) = +1$ (*resp.* $\delta(t) = -1$); by convention, if p has no outgoing transition, p is in Q_+. For every state p of Q_+ (*resp.* Q_-), we set $\delta(p) = 1$ (*resp.* $\delta(p) = -1$).

If $\mathscr{A}$ is a δ-local automaton, $\{Q_+, Q_-\}$ is a partition of Q.

Proposition 2. *Every two-way $\mathbb{K}$-automaton admits a δ-local in-covering.*

Proof. In this proof, we denote $\pm = \{-1,+1\}$. Let $\mathscr{A} = (R,A,F,J,U)$ be a two-way $\mathbb{K}$-automaton and let $P = R \setminus (R_+ \cup R_-)$ be the set of states in $\mathscr{A}$ such that there are at least two transitions with different direction outgoing from each state. Let P_+ and P_- be two copies of P and let $Q = R_+ \cup R_- \cup P_+ \cup P_-$. Let φ be the canonical mapping from Q onto R: it maps every element of P_+ or P_- onto the corresponding element of P. Let $\tilde{\varphi}$ be the mapping from $Q \times A_{\vdash\dashv} \times \pm \times Q$ into $R \times A_{\vdash\dashv} \times \pm \times R$ defined by $\tilde{\varphi}(p,a,d,q) = (\varphi(p), a, d, \varphi(q))$.

Let $\mathscr{A}' = (Q,A,E,I,T)$ be the automaton defined by:

$$\underline{I} = \varphi^{-1}(\underline{J}); \quad \underline{T} = \varphi^{-1}(\underline{U}) \setminus P_-;$$
$$\underline{E} = \{(p,a,d,q) \in \tilde{\varphi}^{-1}(\underline{F}) \mid (p,d) \in (P_+ \cup R_+) \times \{+1\} \cup (P_- \cup R_-) \times \{-1\}\};$$
$$\forall p \in \underline{I},\ I(p) = J(\varphi(p)), \quad \forall p \in \underline{T},\ T(p) = U(\varphi(p)), \quad \forall t \in \underline{E},\ E(t) = F(\tilde{\varphi}(p)).$$

The automaton $\mathscr{A}'$ is δ-local and it is an in-covering of $\mathscr{A}$. □

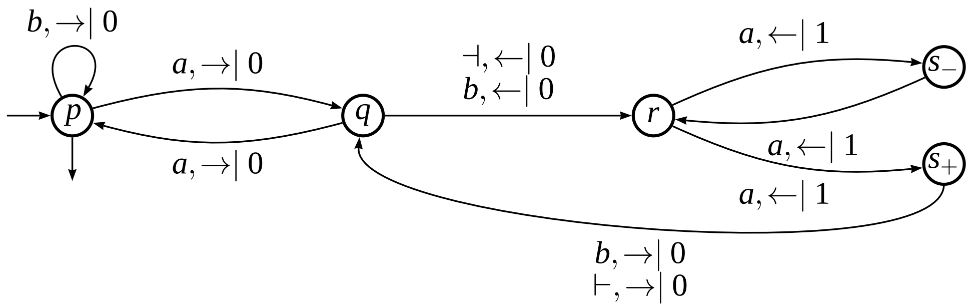


Figure 3: The δ-local two-way distance automaton $\mathscr{A}_1'$.

Example 3. *The automaton $\mathscr{A}_1$ of Figure 1 is not δ-local; from state q (resp. s), there are transitions leaving with $\delta = 1$ and other ones with $\delta = -1$. The automaton $\mathscr{A}_1'$ of Figure 3 is a δ-local in-covering of $\mathscr{A}_1$. Notice that an in-covering of a deterministic automaton is not necessarily deterministic.*

Actually, on $\mathscr{A}_1'$, transitions $s_+ \xrightarrow{b,\rightarrow|0} q_-$ and $s_+ \xrightarrow{\vdash,\rightarrow|0} q_-$ do not belong to any computation, since the label of any trnasition that would follow one of these boths transitions should be the same as the label of the transition arriving at s_- (a), and there is no transition outgoing from q_- with label a.

3 Slices

In this section, we describe the conversion of two-way automata over commutative semirings into one-way automata. We give sufficient conditions to get finite one-way automata.

3.1 The Slice Automaton

Definition 7. *Let $\mathscr{A} = (Q,A,E,I,T)$ be a two-way $\mathbb{K}$-automaton and let $w = w_1 \dots w_k$ be a word. $\rho = ((p_0,i_0),\dots(p_n,i_n))$ be a run over w, and j in $[1;k+1]$. Let h be the subsequence of all pairs (p_r,i_r) such that $(i_r,i_{r+1}) = (j,j+1)$ or $(i_{r-1},i_r) = (j,j-1)$. The j-th* slice *of ρ is the vector $s^{(j)}$ of states obtained by the projection of the first component of each pair of h.*
The signature *$S(\rho)$ of ρ is the sequence of its slices.*

The slices we define here are not exactly the *crossing sequences* defined in [14].

Example 4. *The vector $\begin{bmatrix} q \\ r \\ p \end{bmatrix}$ is the second (and the seventh) slice of the run of Figure 2. The signature of this run is:*

$$\left(\begin{matrix} p \\ s \\ q \end{matrix}, \begin{matrix} q \\ r \\ p \end{matrix}, p, q, p, \begin{matrix} p \\ s \\ q \end{matrix}, \begin{matrix} q \\ r \\ p \end{matrix} \right). \tag{1}$$

The signature of the (unique) run on the word abaaba in the automaton $\mathscr{A}_1'$ is

$$\left(\begin{matrix} p \\ s_+ \\ q_+ \end{matrix}, \begin{matrix} q_- \\ r \\ p \end{matrix}, p, q_+, p, \begin{matrix} p \\ s_+ \\ q_+ \end{matrix}, \begin{matrix} q_- \\ r \\ p \end{matrix} \right). \tag{2}$$

Let $\mathscr{A} = (Q, A, E, I, T)$ be a δ-local two-way $\mathbb{K}$-automaton. To define a one-way $\mathbb{K}$-automaton from slices we consider the set X of subvectors of slices, that are vectors v in Q^* with an odd length; let Y be the vectors v in Q^* with an even length.

We define inductively two partial functions $\theta : X \times A \times X \to \mathbb{K}$ and $\eta : Y \times A \times Y \to \mathbb{K}$ by:

$$\begin{aligned} & & \eta(\varepsilon, a, \varepsilon) &= 0_{\mathbb{K}}, \\ \forall p, q \in Q, \quad & \delta(p) = 1 \Longrightarrow \forall u, v \in Y, & \theta(pu, a, qv) &= E(p, a, 1, q) + \eta(u, a, v), \\ & & \eta(u, a, pqv) &= E(p, a, 1, q) + \eta(u, a, v), \\ & \delta(p) = -1 \Longrightarrow \forall u, v \in X, & \theta(pqu, a, v) &= E(p, a, -1, q) + \theta(u, a, v), \\ & & \eta(qu, a, pv) &= E(p, a, -1, q) + \theta(u, a, yv). \end{aligned} \tag{3}$$

Since $\mathscr{A}$ is δ-local, for every triple (u, a, v) in $X \times A \times X$, if $\theta(u, a, v)$ is defined, it is uniquely defined.

For every vector pu in X, pu is initial if p is in $\underline{I}$ and $(\varepsilon, \vdash, u)$ is in $\underline{\eta}$; in this case, we set $\mathscr{I}(pu) = I(p) + \eta(\varepsilon, \vdash, u)$. Likewise, every vector up in X is final if p is in $\underline{T}$ and $\overline{(u, \dashv, \varepsilon)}$ is in $\underline{\eta}$; in this case, we set $\mathscr{T}(up) = \eta(u, \dashv, \varepsilon) + T(p)$.

Example 5. *For instance, with slices from automaton $\mathscr{A}_1$,*

$$\begin{aligned} \theta\left(\begin{matrix} p \\ s_+ \\ q_+ \end{matrix}, a, \begin{matrix} q_- \\ r \\ p \end{matrix}\right) &= E(p, a, 1, q_-) + \eta\left(\begin{matrix} s_+ \\ q_+ \end{matrix}, a, \begin{matrix} r \\ p \end{matrix}\right) \\ &= E(p, a, 1, q_-) + E(r, a, -1, s_+) + \theta(q_+, a, p) \\ &= E(p, a, 1, q_-) + E(r, a, -1, s_+) + E(q_+, a, 1, p). \end{aligned} \tag{4}$$

The vector $\begin{bmatrix} p \\ s_+ \\ q_+ \end{bmatrix}$ *is initial and*

$$\mathscr{I}\begin{pmatrix} p \\ s_+ \\ q_+ \end{pmatrix} = I(p) + E(s_+, \vdash, 1, q_+). \tag{5}$$

Definition 8. *Let $\mathscr{A} = (Q, A, E, I, T)$ be a two-way $\mathbb{K}$-automaton. With the above notations, the* slice automaton *of $\mathscr{A}$ is the infinite one-way $\mathbb{K}$-automaton $\mathscr{C} = (X, A, \theta, \mathscr{I}, \mathscr{T})$.*

Proposition 3. *Let $\mathbb{K}$ be a* commutative *semiring and let $\mathscr{A}$ be a δ-local two-way $\mathbb{K}$-automaton. There is a bijection φ between the computations of $\mathscr{A}$ and the computations of the slice automaton of $\mathscr{A}$ such that, for every computation ρ of $\mathscr{A}$,*
– ρ and $\varphi(\rho)$ have the same label and the same weight;
– the signature of ρ is the sequence of states of $\varphi(\rho)$.

Proof. Let $\mathscr{C}$ be the slice automaton of $\mathscr{A}$. Let π be a run in $\mathscr{C}$ with label w. Let $\pi^{(k)}$ be the prefix of length k of π and let $(v^{(0)}, \dots, v^{(k)})$ be the sequence of states of $\pi^{(k)}$. We show by induction on k that from $\pi^{(k)}$, there is a unique way to retrieve the restriction of a run of $\mathscr{A}$ on w to the k first letters. Moreover, the weight of $\pi^{(k)}$ (including initial weight) is equal to the weight of this restriction. If $k = 0$, $\pi^{(k)}$ is reduced to an initial slice. By Equation 3, the restriction of the path in the two-way automaton is uniquely defined: $v_1^{(0)}$ is initial with weight $I(v_1^{(0)})$, and for every r in $[1; (v^{(0)} - 1)/2]$, there is a transition $v_{2r}^{(0)} \xrightarrow{\vdash, \to |h_r} v_{2r+1}^{(0)}$; the weight of this restriction is actually the initial weight of $v^{(0)}$ in $\mathscr{C}$. If $k > 0$, we consider the restriction built for $k-1$; this restriction corresponds to a disjoint union of parts of the

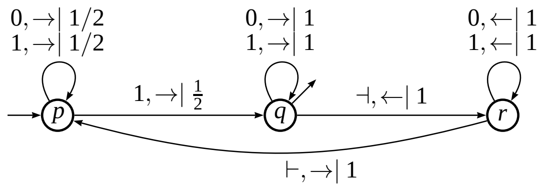


Figure 4: The two-way $\mathbb{Q}$-automaton $\mathscr{A}_2$.

computations and there is only one way to connect them to the states of the slice $v^{(k)}$ (since $\mathscr{A}$ is δ-local). The weight of the transition between $v^{(k-1)}$ and $v^{(k)}$ is exacltly the sum of the weights of the new transitions involved in the restriction.

Finally, from the restriction of length $|w|$, if we consider $v^{(|w|)}$ as a final state of $\mathscr{C}$, by an argument similar to the initial state, we obtain that there is one and only one run in $\mathscr{A}$ that corresponds to a given run in $\mathscr{C}$. □

3.2 Reduced computations and one-way automata

In unweighted (or Boolean) automata, two-way automata describe exactly the same languages as one-way automata [14, 12]. It is not always the case with weighted automata. For instance, let $\mathbb{K}$ be the semiring of languages of the alphabet $\{x,y\}$. It is not difficult to design a deterministic two-way $\mathbb{K}$-automaton over the alphabet $\{a\}$ such that the image of a^n is x^ny^n (a first left-right traversal outputs an x for each a, then the automaton comes back to the beginning of the word and a second left-right traversal outputs a y for each a). This function is obviously not rational and can not be realized by a one-way $\mathbb{K}$-automaton.

Proposition 4. *Let $\mathbb{K}$ be a* commutative *semiring and let $\mathscr{A}$ be a δ-local two-way $\mathbb{K}$-automaton. There exists a (finite) one-way $\mathbb{K}$-automaton $\mathscr{B}$ such that there is a bijection φ between the reduced computations of $\mathscr{A}$ and the computations of $\mathscr{B}$ such that, for every reduced computation ρ of $\mathscr{A}$,*
– ρ and $\varphi(\rho)$ have the same label and the same weight;
– the signature of ρ is the sequence of states of $\varphi(\rho)$.

Proof. Let $\mathscr{A} = (Q,A,E,I,T)$ be a two-way $\mathbb{K}$-automaton. We consider vectors of elements of Q such that no state of Q appears twice at positions with the same parity. For all k in $\mathbb{N}$, we set

$$\begin{aligned} V_k =& \{v \in Q^{2k+1} \mid v_i = v_j \Rightarrow i \neq j \mod 2\} \\ =& \{v \in Q^{2k+1} \mid \forall p \in Q, \forall s \in [0;1], |\{i \mid v_i = p \text{ and } i = s \mod 2\}| \leqslant 1\} \end{aligned} \tag{6}$$

For every k larger than $|Q|-1$, V_k is empty. Let $V = \bigcup_k V_k$; we define the one-way $\mathbb{K}$-automaton with set of states V. It is straightforward that a run is reduced if and only if every slice of this run is in V.

By Proposition 3, the restriction of the slice automaton to V gives a finite automaton that fulfils the proposition. □

Actually, the sufficient condition for the finiteness of the trim part of the slice automaton can be weaken. If the number of slices of a two-way automaton is finite, it is equivalent to a one-way automaton. Unfortunately, this condition is not easy to check and is not a necessary condition.

Example 6. *The two-way $\mathbb{Q}$-automaton $\mathscr{A}_2$ computes for each word w over the alphabet $\{0,1\}$ the value $\frac{x}{1-x}$, where $x = \sum_{i\in[1;|w|]} \frac{w_i}{2^i}$. Although $\mathbb{Q}$ is commutative, this two-way automaton is not equivalent to any one-way $\mathbb{Q}$-automaton; $s_2 = |\mathscr{A}_2|$ is not a rational series.*

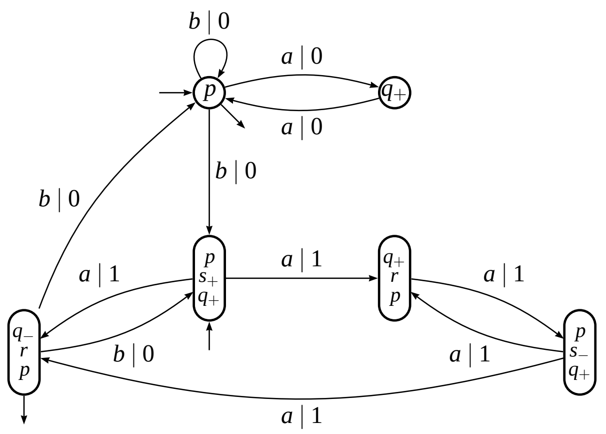


Figure 5: The unambiguous one-way distance automata $\mathscr{B}_1$.

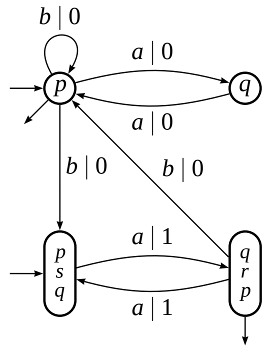


Figure 6: The unambiguous one-way distance automata $\mathscr{B}_1$.

Example 7. *Let $\mathscr{B}_1'$ be the trim part of the slice automaton of $\mathscr{A}_1'$ (Figure 5). In this particular case, although $\mathscr{A}_1$ is not δ-local, the slice automaton $\mathscr{B}_1$ of $\mathscr{A}_1$ (Figure 6) is also unambiguous. It has been shown in [8] that there is no deterministic one-way distance automaton equivalent to these automata.*

4 Unambiguity and Determinism

Since every computation in an unambiguous two-way automaton is reduced, Proposition 4 implies the following statement.

Proposition 5. *Let $\mathbb{K}$ be a commutative semiring. Every unambiguous two-way $\mathbb{K}$-automaton is equivalent to an unambiguous one-way $\mathbb{K}$-automaton.*

A unambiguous one-way automaton can obviously be seen as a unambiguous two-way automaton. In this part, we show that an unambiguous one-way automaton can actually be converted into a determinstic two-way automaton.

4.1 From Unambiguous one-way to Deterministic two-way Automata

Definition 9. *A two-way automaton is deterministic if*
i) it has at most one initial state;

ii) for every state p and every letter a, there is at most one transition outgoing from p with label a;
iii) for every final state p, there is no transition outgoing from p with label $\dashv$.

The last condition means that if a final state is reached at the end of the word, there is no nondeterministic choice between ending the computation and reading the right mark to continue.

Theorem 1. *Let $\mathbb{K}$ be a semiring. Every unambiguous one-way $\mathbb{K}$-automaton is equivalent to a deterministic two-way $\mathbb{K}$-automaton.*

This result is an extension of [7], where it is proved that an unambiguous one-way automaton can be simulated by a deterministic two-way automaton. Our proof is inspired by [3], where it is proven that any rational function can be realized by a sequential two-way transducer. Other works on the conversion of two-way transducers to one-way transducers can be found in [5] or in [4].

Proof. Let $\mathscr{A} = (I, E, T)$ an unambiguous one-way $\mathbb{K}$-automaton with set of states Q.

We consider the mapping μ from A into the $Q \times Q$ Boolean matrices defined by:

$$\forall a \in A,\ \forall p, q \in Q,\ \mu(a)_{p,q} = 1 \Longleftrightarrow (p, a, q) \in \underline{E}. \tag{7}$$

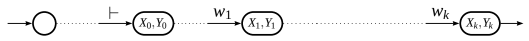
The monoid generated by $\{\mu(a) \mid a \in A\}$ is the *transition* monoid M of $\mathscr{A}$. The mapping μ is naturally extended to a morphism of the monoid A^* onto M. Every subset of Q can be interpreted as a vector in $\mathbb{B}^Q$; for every word w, $\underline{I}\mu(w)$ is the set of states accessible from an initial state by a path with label w and conversely, $\mu(w)\underline{T}$ is the set of states from which a terminal state can be reached by a path with label w.

Since $\mathscr{A}$ is unambiguous, for every pair of words (u, v), $\underline{I}\mu(u) \cap \mu(v)\underline{T}$ has at most one element (otherwise there would exist several computations accepting uv); likewise, for every letter, there exists at most one transition (p, a, q) in $\mathscr{A}$ with p in $\underline{I}\mu(u)$ and q in $\mu(v)\underline{T}$ (otherwise there would exist several computations accepting uav).

For every word $w = w_1 \dots w_k$, for every i in $[0; k]$, we set

$$X_i(w) = \underline{I}\mu(w_1 \dots w_i) \qquad \text{and} \qquad Y_i(w) = \mu(w_{i+1} \dots w_k)\underline{T}.$$

We build a deterministic two-way $\mathbb{K}$-automaton $\mathscr{B}$ equivalent to $\mathscr{A}$. $\mathscr{B}$ has the following property. If w is accepted by $\mathscr{B}$, for every i in $[1; k]$, the state reached after the last reading of w_i contains the information $(X_i(w), Y_i(w))$:



From $(X_{i-1}(w), Y_{i-1}(w))$ and $(X_i(w), Y_i(w))$, the transition labeled by w_i in the run with label w can be deduced: it is the only transition (p, w_i, q) with p in $X_{i-1}(w)$ and q in $Y_i(w)$. Likewise (X_0, Y_0) determines the initial weight and (X_k, Y_k) determines the final weight.

The set X_i can easily be deduced from X_{i-1} : $X_i = X_{i-1}\mu(w_i)$. the computation of Y_i from Y_{i-1} is more subtle.

Let x and y be two elements of M. If there exists z in M such that $x = zy$, we say that $x \leqslant_L y$; this relation is a preorder. If there also exists t such that $tx = y$, we say that x and y are L-equivalent.

Let u be a factor of w that starts in w_{i+1}. It obviously holds $\mu(w_i u) \leqslant_L \mu(u)$. If $\mu(w_i u)$ and $\mu(u)$ are L-equivalent, there exists y in M such that $y\mu(w_i u) = \mu(u)$. In this case, it also holds $y\mu(w_i \dots w_k) = \mu(w_{i+1} \dots w_k)$ and therefore, $Y_i = yY_{i-1}$. The two-way automaton can perform these computations, since they lie in the transition monoid, which is finite. The automaton incrementally computes for each j in $[i; k]$ the value of $\mu(w_{i+1} \dots w_j)$ until $\mu(w_i \dots w_j) <_L \mu(w_{i+1} \dots w_j)$ and $\mu(w_i \dots w_{j+1}) \equiv_L \mu(w_{i+1} \dots w_{j+1})$. If it reaches $j = k$, then $Y_i = \mu(w_{i+1}\dots w_j)T$, otherwise, $Y_i = yY_{i-1}$ where y is such that $y\mu(w_i\dots w_{j+1}) = \mu(w_{i+1}\dots w_{j+1})$.

Once Y_i is computed, the automaton must come back to position i. The automaton is in some position j such that $\mu(w_i\dots w_j) <_L \mu(w_{i+1}\dots w_j)$; *a fortiori*, for every r in $[i+1;j]$, $\mu(w_i\dots w_j) <_L \mu(w_r\dots w_j)$. The automaton therefore spans every position smaller than j until it arrives to some point s such that $\mu(w_i\dots w_j) = \mu(w_s\dots w_j)$. It then holds $s = i$.

Let $\mathscr{P}$ be the powerset of Q. The set of states of $\mathscr{B}$ is the union of five kinds of states:
– $Q_0 = \{i\}$ is the initial state; in this state, the automaton read the input from left to right until it reached the right mark $\dashv$. It then goes to the state $\underline{T}$ in Q_1.
– $Q_1 \subseteq \mathscr{P}$; in this state, the automaton read the input w from right to left; after reading the suffix v, the state corresponds to $\mu(v)\underline{T}$. When the left mark $\vdash$ is reached, the automaton goes to the state $(\underline{I}, \mu(w)\underline{T})$ in Q_2.
– $Q_2 \subseteq \mathscr{P}^2$; these states corresponds to the pairs (X_i, Y_i); the incoming transitions on these states correspond to the transition of the one-way automaton; they are weighted by the corresponding weight. Likewise, a state in Q_2 may be terminal if it belongs to $\mathscr{P} \times \{T\}$. When the automaton is in one of these states, either it stops, or it starts to deal with a new letter; this letter is read and stored in the next state which belongs to Q_3.
– $Q_3 \subseteq A \times M \times \mathscr{P}^2$; the automaton stays in states Q_3 as long as it needs to compute Y_i from Y_{i-1}. It stores the current letter a as well as the image in the transition monoid of the factor u that follows a and ends at the current position. If the state stores $\mu(u)$ that is L-larger than $\mu(au)$ and the read letter b is such that $\mu(aub)$ and $\mu(ub)$ are L-equivalent, there exists y such that $y\mu(aub) = \mu(ub)$; then $Y_i = yY_{i-1}$, the automaton stores $\mu(au)$ and jump to a state in Q_4.
– $Q_4 \subseteq M^2 \times \mathscr{P}^2$; the automaton stays in a state of Q_4 while it reads from right to left the word u; it stores the image of the suffix v of u which is read; it holds $\mu(v) >_L \mu(au)$ until $v = u$; at this point, the automaton read the letter a and checks that $\mu(av) = \mu(au)$; at this point, it knows both X_i and Y_{i+1}, therefore, it can output the weigth of the unique transition compatible with a, X_i and Y_{i+1}, and jump to the state $(X_{i+1} = X_i\mu(a), Y_{i+1})$.

$$\begin{aligned}
F = &\{i \xrightarrow{a,\rightarrow} i \in Q_0 \mid a \in A\}\\
&\cup \{i \xrightarrow{\dashv,\leftarrow} \underline{T} \in Q_1\}\\
&\cup \{Y \xrightarrow{a,\leftarrow} \mu(a)Y \in Q_1 \mid Y \in Q_1, a \in A\}\\
&\cup \{Y \xrightarrow{\vdash,\rightarrow|I_k} (I,Y) \in Q_2 \mid Y \in Q_1, k \in I \cap Y\}\\
&\cup \{(X,Y) \xrightarrow{a,\rightarrow} (a, 1_{\mathbb{K}}, X, Y) \in Q_3 \mid (X,Y) \in Q_2, a \in A\}\\
&\cup \{(a,x,X,Y) \xrightarrow{b,\rightarrow} (a, x\mu(b), X, Y) \in Q_3 \mid (a,x,X,Y) \in Q_3, b \in A, \mu(a)x\mu(b) <_L x\mu(b)\}\\
&\cup \{(a,x,X,Y) \xrightarrow{b,\leftarrow} (\mu(a)x, 1, X, yY) \in Q_4 \mid (a,x,X,Y) \in Q_3, b \in A, y \in M, y\mu(a)x\mu(b) = x\mu(b)\}\\
&\cup \{(a,x,X,Y) \xrightarrow{\dashv,\leftarrow} (\mu(a)x, 1, X, T) \in Q_4 \mid (a,x,X,Y) \in Q_3\}\\
&\cup \{(x,y,X,Y) \xrightarrow{a,\leftarrow} (x, \mu(a)y, X, Y) \in Q_4 \mid (x,y,X,Y) \in Q_4, a \in A, x <_L \mu(a)y\}\\
&\cup \{(x,y,X,Y) \xrightarrow{a,\rightarrow|k} (X\mu(a), Y) \in Q_2 \mid (x,y,X,Y) \in Q_4, a \in A, x = \mu(a)y,\\
&\qquad\qquad \exists (p,q) \in X \times Y, \exists p \xrightarrow{a|k} q \in \mathscr{A}\}.
\end{aligned}$$

□

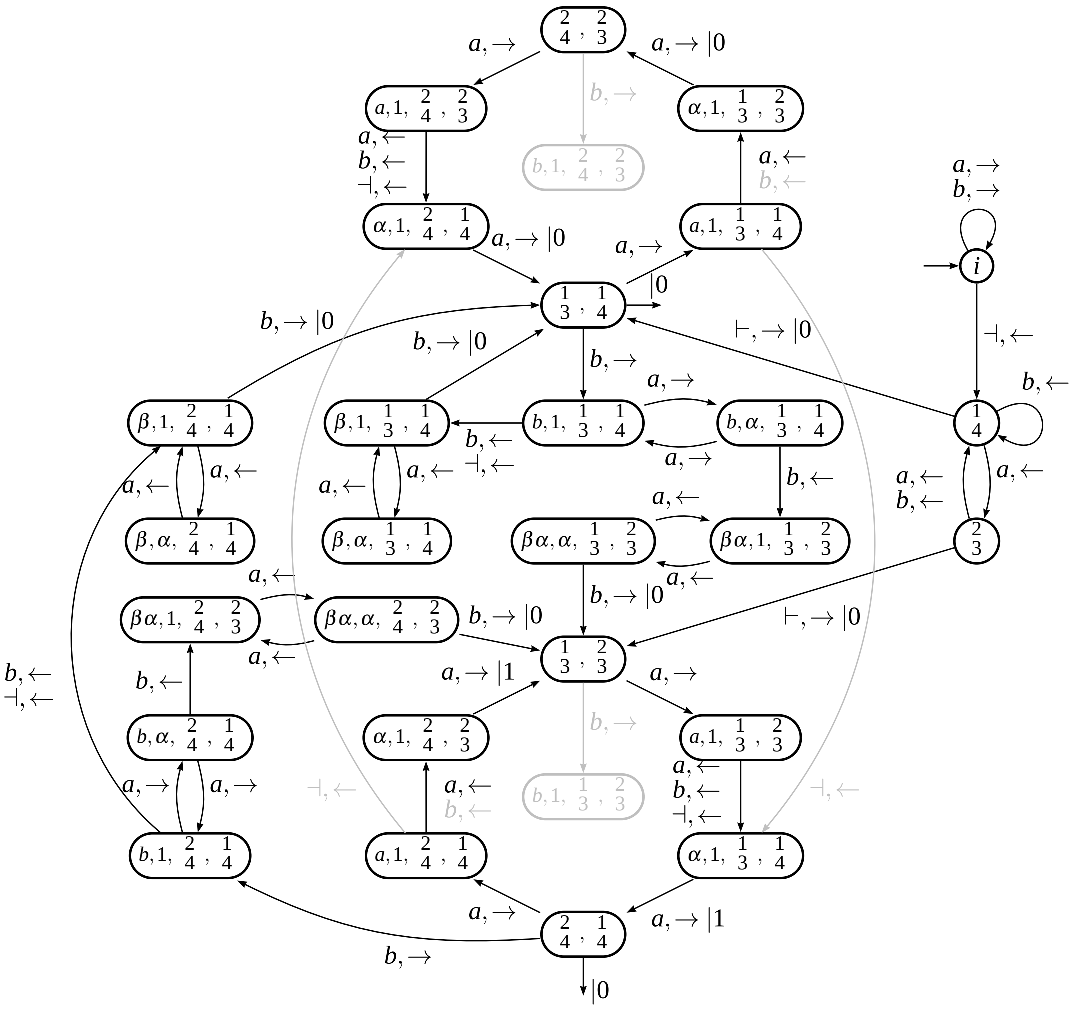


Figure 7: The deterministic two-way distance automaton $\mathscr{D}_1$. The transitions or states in gray are not accessible. For sake of clearness, transitions outgoing from non accessible states are not drawn. Every column of numbers is the set of non-zero components of a Boolean vector of size 4. The weights are only written on transitions where it comes from the weight of a transition (or from an initial/final weight) of $\mathscr{B}_1$.

For every X in $\mathscr{P}$, if the state $(X,\underline{T})$ belongs to Q_2, $(X,\underline{T})$ is final with weight T_p, where p is the unique state in $X \cap \underline{T}$.

Example 8. *Let $\mathscr{B}_1$ be the unambiguous one-way automaton of Figure 6. We number the states of this automaton: $[p]=1$, $[q]=2$, $[p,s,q]=3$ and $[q,r,p]=4$. The transition monoid is generated by the following matrices:*

$$\alpha = \mu(a) = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}, \qquad \beta = \mu(b) = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix}. \tag{8}$$

The following identities hold : $\alpha^2 = 1$, $\beta^2 = \beta$, $\beta\alpha\beta = \beta$. It then holds $1 \equiv_L \alpha$, $\alpha\beta \equiv_L \beta$ and $\alpha\beta\alpha \equiv_L$

$\beta\alpha$, while $\beta <_L 1$ and $\beta\alpha <_L 1$. Notice that β and $\beta\alpha$ are uncomparable. We can apply the proof of Theorem 1 to compute the equivalent deterministic two-way automaton $\mathscr{D}_1$ of Figure 7.

Corollary 1. *Let $\mathbb{K}$ be a commutative semiring. Every unambiguous two-way $\mathbb{K}$-automaton is equivalent to a deterministic one.*

Remark 1. *This conversion can lead to a combinatorial blow-up. For instance, the deterministic two-way automaton built from the unambiguous one-way automato $\mathscr{B}_1$ (Figure 6 (right)) has 27 states in its trim part.*

A lower bound on the number of states can be computed. Let n be the number of states of the unambiguous one-way automaton.

- *Q_0 has one state;*
- *Q_1 has at most $2^n - 1$ states;*
- *Q_2 is made of pairs of subset of Q which share exactly one element, hence Q_2 has at most $n3^{n-1}$ states;*
- *Q_3 is made of a pair of Q_2 endowed with a letter and an element of the transition monoid (that may have 2^{n^2} elements); hence Q_3 has at most $|A|n3^{n-1}2^{n^2}$ states;*
- *Q_4 is made of two (non empty) subsets of Q and two elements of the transition monoid; its size is bounded by 2^{2n+2n^2}.*

References

[1] Marcella Anselmo (1990): *Two-way Automata with Multiplicity*. In: *ICALP'90, Lect. Notes in Comput. Sci.* 443, pp. 88–102, doi:10.1007/BFb0032024.

[2] Marie-Pierre Béal, Olivier Carton, Christophe Prieur & Jacques Sakarovitch (2003): *Squaring transducers: an efficient procedure for deciding functionality and sequentiality*. *Theor. Comput. Sci.* 292(1), pp. 45–63, doi:10.1016/S0304-3975(01)00214-6.

[3] Olivier Carton (2012): *Two-Way Transducers with a Two-Way Output Tape*. In: *DLT'12, Lect. Notes in Comput. Sci.* 7410, pp. 263–272, doi:10.1007/978-3-642-31653-1_24.

[4] Rodrigo De Souza (2013): *Uniformisation of Two-Way Transducers*. In: *LATA'13, Lect. Notes in Comput. Sci.* 7810, pp. 547–558, doi:10.1007/978-3-642-37064-9_48.

[5] Joost Engelfriet & Hendrik Jan Hoogeboom (2007): *Finitary Compositions of Two-way Finite-State Transductions*. *Fundam. Inform.* 80(1-3), pp. 111–123. Available at http://iospress.metapress.com/content/143422w0253h8644/.

[6] Zoltán Ésik & Werner Kuich (2009): *Handbook of Weighted Automata*, chapter Finite Automata, pp. 69–104. Springer, doi:10.1007/978-3-642-01492-5.

[7] John E. Hopcroft & Jeffrey D. Ullman (1967): *An Approach to a Unified Theory of Automata*. In: *SWAT (FOCS)*, IEEE Computer Society, pp. 140–147, doi:10.1109/FOCS.1967.4.

[8] Ines Klimann, Sylvain Lombardy, Jean Mairesse & Christophe Prieur (2004): *Deciding unambiguity and sequentiality from a finitely ambiguous max-plus automaton*. *Theor. Comput. Sci.* 327(3), pp. 349–373, doi:10.1016/j.tcs.2004.02.049.

[9] Daniel Krob (1994): *The equality problem for rational series with multiplicities in the tropical semiring is undecidable*. *Internat. J. Algebra Comput.* 4(3), pp. 405–425, doi:10.1142/S0218196794000063.

[10] Sylvain Lombardy & Jean Mairesse (2006): *Series which are both max-plus and min-plus rational are unambiguous*. *RAIRO - Theor. Inf. and Appl.* 40(1), pp. 1–14, doi:10.1051/ita:2005042.

[11] Sylvain Lombardy & Jacques Sakarovitch (2013): *The validity of weighted automata*. *Internat. J. Algebra Comput.* 23, pp. 863–913, doi:10.1142/S0218196713400146.

[12] M. O. Rabin & D. Scott (1959): *Finite automata and their decision problems*. *IBM J. Res. Dev.* 3(2), pp. 114–125, doi:10.1147/rd.32.0114.

[13] Jacques Sakarovitch (2009): *Elements of Automata Theory*. Cambridge University Press, doi:10.1017/CBO9781139195218.

[14] J. C. Shepherdson (1959): *The reduction of two-way automata to one-way automata*. *IBM J. Res. Dev.* 3(2), pp. 198–200, doi:10.1147/rd.32.0198.

Operations on Automata with All States Final

Kristína Čevorová
Mathematical Institute,
Slovak Academy of Sciences,
Bratislava, Slovakia
cevorova@mat.savba.sk

Galina Jirásková[*]
jiraskov@saske.sk

Peter Mlynárčik
Mathematical Institute,
Slovak Academy of Sciences,
Košice, Slovakia
mlynarcik1972@gmail.com

Matúš Palmovský
matp93@gmail.com

Juraj Šebej[†]
Institute of Computer Science,
Šafárik University,
Košice, Slovakia
juraj.sebej@gmail.com

We study the complexity of basic regular operations on languages represented by incomplete deterministic or nondeterministic automata, in which all states are final. Such languages are known to be prefix-closed. We get tight bounds on both incomplete and nondeterministic state complexity of complement, intersection, union, concatenation, star, and reversal on prefix-closed languages.

1 Introduction

A language L is prefix-closed if $w \in L$ implies that every prefix of w is in L. It is known that a regular language is prefix-closed if and only if it is accepted by a nondeterministic finite automaton (NFA) with all states final [18]. In the minimal incomplete deterministic finite automaton (DFA) for a prefix-closed language, all the states are final as well.

The authors of [18] examined several questions concerning NFAs with all states final. They proved that the inequivalence problem for NFAs with all states final is PSPACE-complete in the binary case, but polynomially solvable in the unary case. Next, they showed that minimizing a binary NFA with all states final is PSPACE-hard, and that deciding whether a given NFA accepts a language that is not prefix-closed is PSPACE-complete, while the same problem for DFAs can be solved in polynomial time. The NFA-to-DFA conversion and complementation of NFAs with all states final have been also considered in [18], and the tight bound 2^n for the first problem, and the lower bound 2^{n-1} for the second one have been obtained.

The quotient complexity of prefix-closed languages has been studied in [5]. The quotient of a language L by the string w is the set $L_w = \{x \mid wx \in L\}$. The quotient complexity of a language L, $\kappa(L)$, is the number of distinct quotients of L. Quotient complexity is defined for any language, and it is finite if and only if the language is regular. The quotient automaton of a regular language L is the DFA $(\{L_w \mid w \in \Sigma^*\}, \Sigma, \cdot, L_\varepsilon, F)$, where $L_w \cdot a = L_{wa}$, and a quotient L_w is final if it contains the empty string. The quotient automaton of L is a minimal complete DFA for L, so quotient complexity is the same as the state complexity of L which is defined as the number of states in the minimal DFA for L. In [5], the tight bounds on the quotient complexity of basic regular operation have been obtained, and to prove upper bounds, the properties of quotients have been used rather than automata constructions.

[*]Research supported by grant APVV-0035-10.
[†]Research supported by grant VEGA 1/0479/12.

Automata with all states final represent systems, for example, production lines, and their intersection or parallel composition represents the composition of these systems [21]. A question that arises here is, whether the complexity of intersection of automata with all states final is the same as in the general case of arbitrary DFAs or NFAs. At the first glance, it seems that this complexity could be smaller. Our first result shows that this is not the case. We show that both incomplete and nondeterministic state complexity of intersection on prefix-closed languages is given by the function mn, which is the same as in the general case of regular languages.

In the deterministic case, to have all the states final, we have to consider incomplete deterministic automata because otherwise, the complete automaton with all states final would accept the language consisting of all the strings over an input alphabet. Notice that the model of incomplete deterministic automata has been considered already by Maslov [20]. The same model has been used in the study of the complexity of the shuffle operation [6]; here, the complexity on complete DFAs is not known yet.

We next study the complexity of complement, union, concatenation, square, star, and reversal on languages represented by incomplete DFAs or NFAs with all states final. We get tight bounds in both nondeterministic and incomplete deterministic cases. In the nondeterministic case, all the bounds are the same as in the general case of regular languages, except for the bound for star that is n instead of $n+1$. However, to prove the tightness of these bounds, we usually use larger alphabets than in the general case of regular languages where all the upper bounds can be met by binary languages [10, 12].

To get lower bounds, we use a fooling-set lower-bound method [1, 2, 3, 8, 11]. In the case of union and reversal, the method does not work since it provides a lower bound on the size of NFAs with multiple initial states. Since the nondeterministic state complexity of a regular language is defined using a model of NFAs with a single initial state [10], we have to use a modified fooling-set technique to get the tight bounds $m+n+1$ and $n+1$ for union and reversal, respectively.

In the case of incomplete deterministic finite automata, the tight bounds for complement, union, concatenation, star, and reversal are $n+1, mn+m+n, m\cdot 2^{n-1}+2^n-1$, 2^{n-1}, and 2^n-1, respectively. To define worst-case examples, we use a binary alphabet for union, star, and reversal, and a ternary alphabet for concatenation.

The paper is organized as follows. In the next section, we give some basic definitions and preliminary results. In Sections 3 and 4, we study boolean operations. Concatenation is discussed in Section 5, and star and reversal in Section 6. The last section contains some concluding remarks.

2 Preliminaries

In this section, we recall some basic definitions and preliminary results. For details and all unexplained notions, the reader may refer to [24].

A *nondeterministic finite automaton* (NFA) is a quintuple $A=(Q,\Sigma,\delta,I,F)$, where Q is a finite set of states, Σ is a finite alphabet, $\delta\colon Q\times\Sigma\to 2^Q$ is the transition function which is extended to the domain $2^Q\times\Sigma^*$ in the natural way, $I\subseteq Q$ is the set of initial states, and $F\subseteq Q$ is the set of final states. The language accepted by A is the set $L(A)=\{w\in\Sigma^*\mid \delta(I,w)\cap F\neq\emptyset\}$.

The *nondeterministic state complexity* of a regular language L, $\text{nsc}(L)$, is the smallest number of states in any NFA with a *single initial state* recognizing L.

An NFA A is *incomplete deterministic* (DFA) if $|I|=1$ and $|\delta(q,a)|\leq 1$ for each q in Q and each a in Σ. In such a case, we write $\delta(q,a)=q'$ instead of $\delta(q,a)=\{q'\}$. A non-final state q of a DFA is called a *dead* state if $\delta(q,a)=q$ for each symbol a in Σ.

The *incomplete state complexity* of a regular language L, $\mathrm{isc}(L)$, is the smallest number of states in any incomplete DFA recognizing L. An incomplete DFA is minimal (with respect to the number of states) if it does not have any dead state, all its states are reachable, and no two distinct states are equivalent.

Every NFA $A = (Q,\Sigma,\delta,I,F)$ can be converted to an equivalent DFA $A' = (2^Q,\Sigma,\cdot,I,F')$, where $R \cdot a = \delta(R,a)$ and $F' = \{R \in 2^Q \mid R \cap F \neq \emptyset\}$. The DFA A' is called the *subset automaton* of the NFA A. The subset automaton need not be minimal since some of its states may be unreachable or equivalent. However, if for each state q of an NFA A, there exists a string w_q that is accepted by A only from the state q, then the subset automaton of the NFA A does not have equivalent states since if two subsets of the subset automaton differ in a state q, then they are distinguishable by w_q.

To prove the minimality of NFAs, we use a fooling set lower-bound technique, see [1, 2, 3, 8, 11].

Definition A set of pairs of strings $\{(x_1,y_1),(x_2,y_2),\ldots,(x_n,y_n)\}$ is called a *fooling set* for a language L if for all i,j in $\{1,2,\ldots,n\}$, the following two conditions hold:

(F1) $x_iy_i \in L$, and

(F2) if $i \neq j$, then $x_iy_j \notin L$ or $x_jy_i \notin L$.

It is well known that the size of a fooling set for a regular language provides a lower bound on the number of states in any NFA (with multiple initial states) for the language. The argument is simple. Fix the accepting computations of any NFA on strings x_iy_i and x_jy_j. Then, the states on these computations reached after reading x_i and x_j must be distinct, otherwise the NFA accepts both x_iy_j and x_jy_i for two distinct pairs. Hence we get the following observation.

Lemma 1 ([3, 8, 11])**.** *Let $\mathscr{F}$ be a fooling set for a language L. Then every NFA (with multiple initial states) for the language L has at least $|\mathscr{F}|$ states.* □

The next lemma shows that sometimes, if we insist on having a single initial state in an NFA, one more state is necessary. It can be used in the case of union, reversal, cyclic shift [15], and AFA-to-NFA conversion [13]. In each of these cases, NFAs with a single initial state require one more state than NFAs with multiple initial states. For the sake of completeness, we recall the proof of the lemma here.

Lemma 2 ([14])**.** *Let $\mathscr{A}$ and $\mathscr{B}$ be sets of pairs of strings and let u and v be two strings such that $\mathscr{A} \cup \mathscr{B}$, $\mathscr{A} \cup \{(\varepsilon,u)\}$, and $\mathscr{B} \cup \{(\varepsilon,v)\}$ are fooling sets for a language L. Then every NFA with a single initial state for the language L has at least $|\mathscr{A}|+|\mathscr{B}|+1$ states.*

Proof. Consider an NFA for a language L, and let $\mathscr{A} = \{(x_i,y_i) \mid i = 1,2,\ldots,m\}$ and $\mathscr{B} = \{(x_{m+j},y_{m+j}) \mid j = 1,2,\ldots,n\}$. Since the strings x_ky_k are in L, we fix an accepting computation of the NFA on each string x_ky_k. Let p_k be the state on this computation that is reached after reading x_k. Since $\mathscr{A} \cup \mathscr{B}$ is a fooling set for L, the states $p_1, p_2, \ldots, p_{m+n}$ are pairwise distinct. Since $\mathscr{A} \cup \{(\varepsilon,u)\}$ is a fooling set, the initial state is distinct from all the states $p_1, p_2, \ldots, p_m$. Since $\mathscr{B} \cup \{(\varepsilon,v)\}$ is a fooling set, the (single) initial state is also distinct from all the states $p_{m+1}, p_{m+2}, \ldots, p_{m+n}$. Thus the NFA has at least $m+n+1$ states. □

Example Let $K = (a^3)^*$ and $L = (b^3)^*$. Then $\mathrm{nsc}(K) = 3$ and $\mathrm{nsc}(L) = 3$, and the language $K \cup L$ is accepted by a 6-state NFA with two initial states. Therefore, we cannot expect that we will be able to find a fooling set for $K \cup L$ of size 7. However, every NFA with a *single* initial state for the language $K \cup L$ requires at least 7 states since Lemma 2 is satisfied for the language $K \cup L$ with

$$\begin{aligned}
\mathscr{A} &= \{(a,a^2),(a^2,a),(a^3,a^3)\},\\
\mathscr{B} &= \{(b,b^2),(b^2,b),(b^3,b^3)\},\\
u &= b^3, \text{ and}\\
v &= a^3.
\end{aligned}$$

If $w = uv$ for strings u and v, then u is a *prefix* of w. A language L is *prefix-closed* if $w \in L$ implies that every prefix of w is in L. The following observations are easy to prove.

Proposition 3 ([18])**.** *A regular language is prefix-closed if and only if it is accepted by some NFA with all states final.* ☐

Proposition 4. *Let A be a minimal incomplete DFA for a language L. Then the language L is prefix-closed if and only if all the states of the DFA A are final.* ☐

3 Complementation

If L is a language over an alphabet Σ, then the complement of L is the language $L^c = \Sigma^* \setminus L$. If L is accepted by a minimal complete DFA A, then we can get a minimal DFA for L^c from the DFA A by interchanging the final and non-final states. In the case of incomplete DFAs, we first have to add a dead state, that is, a non-final state which goes to itself on each input, and let all the undefined transitions go to the dead state. After that, we can interchange the final and non-final states to get a (complete) DFA for the complement. This gives the following result.

Theorem 5. *Let $n \geq 1$. Let L be a prefix-closed regular language over an alphabet Σ with* $\operatorname{isc}(L) = n$*. Then* $\operatorname{isc}(L^c) \leq n+1$*, and the bound is tight if* $|\Sigma| \geq 1$*.*

Proof. For tightness, we can consider the unary prefix-closed language $\{a^i \mid 0 \leq i \leq n-1\}$. ☐

If a language L is represented by an n-state NFA, then we first construct the corresponding subset automaton, and then interchange the final and non-final states to get a DFA for the language L^c of at most 2^n states. This upper bound on the nondeterministic state complexity of complement on regular languages is know to be tight in the binary case [12].

For prefix-closed languages, we get the same bound, however, to prove tightness, we use a ternary alphabet. Whether or not the bound 2^n can be met by a binary language remains open.

Theorem 6. *Let $n \geq 2$. Let L be a prefix-closed regular language over an alphabet Σ with* $\operatorname{nsc}(L) = n$*. Then* $\operatorname{nsc}(L^c) \leq 2^n$*, and the bound is tight if* $|\Sigma| \geq 3$*.*

Proof. The upper bound is the same as in the general case of regular languages [10]. To prove tightness, consider the language L accepted by the NFA N shown in Figure 1, in which state n goes to the empty set on both a and b, and to $\{1\}$ on c. Each other state i goes to $\{i+1\}$ on both a and c, and to $\{1, i+1\}$ on b. Our aim is to describe a fooling set $\mathscr{F} = \{(x_S, y_S) \mid S \subseteq \{1,2,\ldots,n\}\}$ of size 2^n for L^c.

First, let us show that each subset of $\{1,2,\ldots,n\}$ is reachable in the subset automaton of the NFA N. The initial state is $\{1\}$, and each singleton set $\{i\}$ is reached from $\{1\}$ by a^{i-1}. The empty set is reached from $\{n\}$ by a. The set $\{i_1, i_2, \ldots, i_k\}$ of size k, where $2 \leq k \leq n$ and $1 \leq i_1 < i_2 < \cdots < i_k \leq n$, is reached from the set $\{i_2 - i_1, \ldots, i_k - i_1\}$ of size $k-1$ by the string $ba^{i_1 - 1}$. This proves reachability by induction. Now, define x_S as the string, by which the initial state 1 of the NFA N goes to the set S.

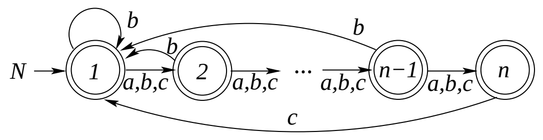


Figure 1: The NFA N of a prefix-closed language L with $\operatorname{nsc}(L^c) = 2^n$.

Next, for a subset S of $\{1,2,\ldots,n\}$, define the string y_S as the string $y_S = y_0 y_1 \cdots y_{n-1}$ of length n, where

$$y_i = \begin{cases} a, & \text{if } n-i \in S, \\ c, & \text{if } n-i \notin S. \end{cases}$$

We claim that the string y_S is rejected by the NFA N from each state in S and accepted from each state that is not in S. Indeed, if i is a state in S, then $y_{n-i} = a$ and $y_S = uav$ with $u = y_0 y_1 \cdots y_{n-i-1}$ and $v = y_{n-i+1} y_{n-i+2} \cdots y_{n-1}$. Hence $|u| = n-i$, which means that the state i goes to $\{n\}$ by u since both a and c move each state q to state $q+1$. However, in state n the NFA N cannot read a, and therefore the string $y_S = uav$ is rejected from i. On the other hand, if $i \notin S$, then $y_{n-i} = c$, and the string $y_S = ucv$ with $|u| = n-i$ and $|v| = i-1$ is accepted from i through the computation $i \xrightarrow{u} n \xrightarrow{c} 1 \xrightarrow{v} i$.

Now, we are ready to prove that the set of pairs of strings $\mathscr{F} = \{(x_S, y_S) \mid S \subseteq \{1,2,\ldots,n\}\}$ is a fooling set for the language L^c.

(F1) By x_S, the initial state 1 goes to the set S. The string y_S is rejected by N from each state in S. It follows that the NFA N rejects the string $x_S y_S$. Thus the string $x_S y_S$ is in L^c.

(F2) Let $S \neq T$. Then without loss of generality, there is a state i such that $i \in S$ and $i \notin T$. By x_S, the initial state 1 goes to S, so it also goes to the state i. Since $i \notin T$, the string x_T is accepted by N from i. Therefore, the NFA N accepts the string $x_S y_T$, and so this string is not in L^c.

Hence $\mathscr{F}$ is a fooling set for L^c of size 2^n. By Lemma 1, we have $\operatorname{nsc}(L^c) \geq 2^n$. □

4 Intersection and Union

In this section, we study the incomplete and nondeterministic state complexity of intersection and union of prefix-closed languages. If regular languages K and L are accepted by m-state and n-state NFAs, respectively, then the language $K \cap L$ is accepted by an NFA of at most mn states, and this bound is known to be tight in the binary case [10]. Our first result shows that the bound mn can be met by binary prefix-closed languages. Then, using this result, we get the same bound on the incomplete state complexity of intersection on prefix-closed languages.

Theorem 7. *Let K and L be prefix-closed languages over an alphabet Σ with $\operatorname{nsc}(K) = m$ and $\operatorname{nsc}(L) = n$. Then $\operatorname{nsc}(K \cap L) \leq mn$, and the bound is tight if $|\Sigma| \geq 2$.*

Proof. The upper bound is the same as for regular languages [10]. For tightness, consider prefix-closed binary languages $K = \{w \in \{a,b\}^* \mid \#_a(w) \leq m-1\}$ and $L = \{w \in \{a,b\}^* \mid \#_b(w) \leq n-1\}$ that are accepted by an m-state and an n-state incomplete DFAs A and B, respectively, shown in Figure 2.

Consider the set of pairs of strings $\mathscr{F} = \{(a^i b^j, a^{m-1-i} b^{n-1-j}) \mid 0 \leq i \leq m-1, 0 \leq j \leq n-1\}$ of size mn. Let us show that $\mathscr{F}$ is a fooling set for the language $K \cap L$.

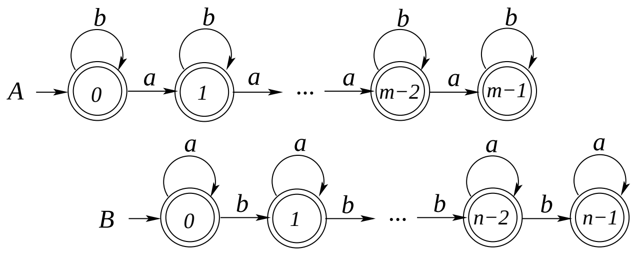


Figure 2: The incomplete DFAs A and B of prefix-closed languages K and L with $\operatorname{nsc}(K \cap L) = mn$.

(F1) The string $a^ib^j \cdot a^{m-1-i}b^{n-1-j}$ has exactly $m-1$ a's and $n-1$ b's. It follows that it is in $K \cap L$.

(F2) Let $(i,j) \neq (k,\ell)$. If $i < k$, then the string $a^kb^\ell \cdot a^{m-1-i}b^{n-1-j}$ contains $m-1+(k-i)$ a's, and therefore it is not in K. The case of $j < \ell$ is symmetric.

Hence $\mathscr{F}$ is a fooling set for $K \cap L$, and the theorem follows. □

Theorem 8. *Let K and L be prefix-closed languages over an alphabet Σ with* $\operatorname{isc}(K) = m$ *and* $\operatorname{isc}(L) = n$. *Then* $\operatorname{isc}(K \cap L) \leq mn$*, and the bound is tight if* $|\Sigma| \geq 2$.

Proof. Let $A = (Q_A, \Sigma, \delta_A, s_A, Q_A)$ and $B = (Q_B, \Sigma, \delta_B, s_B, Q_B)$ be incomplete DFAs for K and L, respectively. Define an incomplete product automaton $M = (Q_A \times Q_B, \Sigma, \delta, (s_A, s_B), Q_A \times Q_B)$, where

$$\delta((p,q),a) = \begin{cases} (\delta_A(p,a), \delta_B(q,a)), & \text{if both } \delta_A(p,a) \text{ and } \delta_B(q,a) \text{ are defined,} \\ \text{undefined}, & \text{otherwise}. \end{cases}$$

The DFA M accepts the language $K \cap L$. This gives the upper bound mn. For tightness, consider the same languages K and L as in the proof of the previous theorem. Notice that K and L are accepted by m-state and n-state incomplete DFAs, respectively. We have shown that nondeterministic state complexity of their intersection is mn. It follows that the incomplete state complexity is also at least mn. □

Our next result on the incomplete state complexity of union on prefix-closed languages can be derived from the result on the quotient complexity of union in [5]. For the sake of completeness, we restate it in terms of incomplete complexities, and recall the proof.

Theorem 9. *Let K and L be prefix-closed languages over an alphabet Σ with* $\operatorname{isc}(K) = m$ *and* $\operatorname{isc}(L) = n$. *Then* $\operatorname{isc}(K \cup L) \leq mn + m + n$*, and the bound is tight if* $|\Sigma| \geq 2$.

Proof. Let $A = (\{0,1,\ldots,m-1\}, \Sigma, \delta_A, 0, F_A)$ and $B = (\{0,1,\ldots,n-1\}, \Sigma, \delta_B, 0, F_B)$ be incomplete DFAs for the languages K and L, respectively. To construct a DFA for the language $K \cup L$, we first add the dead states m and n to the DFAs A and B, and let go all the undefined transitions to the dead states. Now we construct the classic product-automaton from the resulting complete DFAs with the state set $\{0,1,\ldots,m\} \times \{0,1,\ldots,n\}$. All its states are final, except for the state (m,n) that is dead, and we do not count it. Hence we get the upper bound $mn + m + n$ on the incomplete state complexity of union.

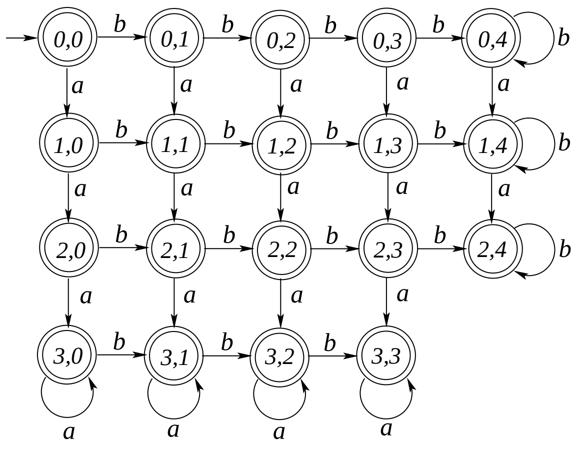


Figure 3: The product automaton for incomplete DFAs A and B from Figure 2; $m = 3$ and $n = 4$.

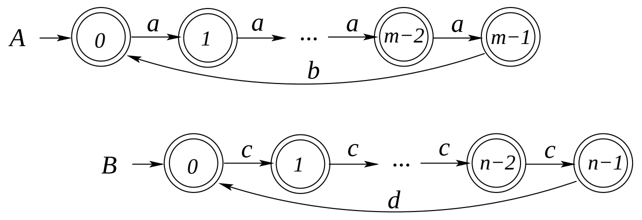


Figure 4: The NFAs A and B of prefix-closed languages K and L with $\operatorname{nsc}(K \cup L) = m+n+1$.

For tightness, we again consider the languages described in the proof of Theorem 7. We add the dead states m and n and construct the product automaton. The product automaton in the case of $m = 3$ and $n = 4$ is shown in Figure 3.

Each state (i, j) of the product automaton is reached from the initial state $(0,0)$ by the string $a^i b^j$. Let (i, j) and (k, ℓ) be two distinct states of the product automaton. If $i < k$, then the string $a^{m-k}b^n$ is rejected from (k, ℓ) and accepted from (i, j). If $j < \ell$, then the string $b^{n-\ell}a^m$ is rejected from (k, ℓ) and accepted from (i, j). Thus all the states in the product-automaton are reachable and pairwise distinguishable, and the lower bound $mn+m+n$ follows. □

In the nondeterministic case, the upper bound for union on regular language is $m+n+1$, and it is tight in the binary case [10]. We get the same bound for union on prefix-closed languages, however, to define witness languages, we use a four-letter alphabet.

Theorem 10. *Let K and L be prefix-closed languages over an alphabet Σ with $\operatorname{nsc}(K) = m$ and $\operatorname{nsc}(L) = n$. Then $\operatorname{nsc}(K \cup L) \le m+n+1$, and the bound is tight if $|\Sigma| \ge 4$.*

Proof. The upper bound is the same as for regular languages [10]. To prove tightness, let K and L be the prefix-closed languages accepted by the NFAs A and B, respectively, shown in Figure 4. Let

$$\mathscr{A} = \{(a^i, a^{m-1-i}b) \mid i = 1,2,\ldots,m-1\} \cup \{(a^{m-1}b, a)\},$$
$$\mathscr{B} = \{(c^j, c^{n-1-j}d) \mid j = 1,2,\ldots,n-1\} \cup \{(c^{n-1}d, c)\}.$$

Let us show that $\mathscr{A} \cup \mathscr{B}$ is a fooling set for the language $K \cup L$.

(F1) We have $a^i \cdot a^{m-1-i}b = a^{m-1}b$ and $c^j \cdot c^{n-1-j}d = c^{n-1}d$. Both these strings are in $K \cup L$. The strings $a^{m-1}b \cdot a$ and $c^{n-1}d \cdot c$ are in $K \cup L$ as well.

(F2) If $1 \le i < i' \le m-1$, then the string $a^i \cdot a^{m-1-i'}b$ is not in K since $m-1-(i'-i) < m-1$. Next, if $1 \le i \le m-1$, then $a^{m-1}b \cdot a^{m-1-i}b$ is not in K. The argumentation for two pairs from $\mathscr{B}$ is similar. If we concatenate the first part of a pair in $\mathscr{A}$ with the second part of a pair in $\mathscr{B}$, then we get a string that either contains all three symbols a, c, d, or contains both symbols a and d. No such string is in $K \cup L$.

Thus $\mathscr{A} \cup \mathscr{B}$ is a fooling set for the language $K \cup L$. Moreover, the sets $\mathscr{A} \cup \{(\varepsilon, c)\}$ and $\mathscr{B} \cup \{(\varepsilon, a)\}$ are fooling sets for $K \cup L$ as well. By Lemma 2, we have $\operatorname{nsc}(K \cup L) \ge m+n+1$. □

5 Concatenation

In this section, we deal with the concatenation operation on prefix-closed languages. We start with incomplete state complexity. We use a slightly different ternary witness language than in [5], and prove the upper bound using automata constructions.

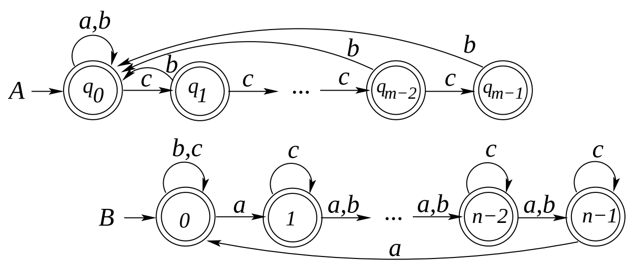


Figure 5: The incomplete DFAs A and B of languages K and L with $\text{isc}(KL) = m \cdot 2^{n-1} + 2^n - 1$.

Theorem 11. *Let $m,n \geq 3$. Let K and L be prefix-closed languages over an alphabet Σ with $\text{isc}(K) = m$ and $\text{isc}(L) = n$. Then $\text{isc}(KL) \leq m \cdot 2^{n-1} + 2^n - 1$, and the bound is tight if $|\Sigma| \geq 3$.*

Proof. Let $A = (Q_A, \Sigma, \delta_A, s_A, Q_A)$ and $B = (Q_B, \Sigma, \delta_B, s_B, Q_B)$ be incomplete DFAs with all states final accepting the languages K and L, respectively. Construct an NFA N for the language KL from the DFAs A and B by adding the transition on a symbol a from a state q in Q_A to the initial state s_B of B whenever the transition on a in state q is defined in A. The initial states of the NFA N are s_A and s_B, and the set of final states is Q_B. Each reachable subset of the subset automaton of the NFA N contains at most one state of Q_A, and several states of Q_B. Moreover, if a state of Q_A is in a reachable subset S, then S must contain the state s_B. This gives the upper bound $m \cdot 2^{n-1} + 2^n - 1$ on $\text{isc}(KL)$ since the empty set is not counted.

For tightness, consider the prefix-closed languages K and L accepted by incomplete DFAs A and B, respectively, shown in Figure 5, in which the transitions are as follows:

on a, state q_0 goes to itself, and each state j goes to $(j+1) \bmod n$;

on b, each state q_i goes to state q_0, state 0 goes to itself, and state j with $1 \leq j \leq n-2$ goes to $j+1$;

on c, each state q_i with $0 \leq i \leq m-2$ goes to q_{i+1}, and each state j goes to itself;

and all the remaining transitions are undefined.

Construct an NFA N for the language KL as described above. Let us show that the subset automaton of the NFA N has $m \cdot 2^{n-1} + 2^n - 1$ reachable and pairwise distinguishable non-empty subsets.

(1) First, let us show that each set $\{q_0\} \cup S$ is reachable, where $S \subseteq \{0, 1, \ldots, n-1\}$ and $0 \in S$. The proof is by induction on the size of subsets. The set $\{q_0, 0\}$ is the initial subset. The set $\{q_0, 0, j_1, j_2, \ldots, j_k\}$ with $1 \leq j_1 < j_2 < \cdots < j_k \leq n-1$ is reached from the set $\{q_0, 0, j_2 - j_1, \ldots, j_k - j_1\}$ by the string $ab^{j_1 - 1}$, and the latter set is reachable by induction.

(2) Now, let us show that each set $\{q_i\} \cup S$, is reachable, where $1 \leq i \leq m-1$, $S \subseteq \{0, 1, \ldots, n-1\}$ and $0 \in S$. The set $\{q_i\} \cup S$ is reached from $\{q_0\} \cup S$ by c^i, and the latter set is reachable as shown in (1).

(3) Next, we show that each set S with $S \subseteq \{0, 1, \ldots, n-1\}$ and $0 \in S$ is reachable. The set S is reached from $\{q_{m-1}\} \cup S$ by c, and the latter set is reachable as shown in case (2).

(4) Finally, we show that each non-empty set S with $S \subseteq \{0, 1, \ldots, n-1\}$ and $0 \notin S$ is reachable. If $S = \{j_1, j_2, \ldots, j_k\}$ with $j_1 \geq 1$, then S is reached from the set $\{0, j_2 - j_1, \ldots, j_k - j_1\}$ by a^{j_1}, and the latter set is reachable as shown in case (3).

This proves the reachability of $m \cdot 2^{n-1} + 2^n - 1$ non-empty subsets.

To prove distinguishability, notice that the string b^n is accepted by the DFA B only from the state 0, and the string $a^{n-1-i}ab^n$ is accepted only from the state i ($1 \leq i \leq n-1$). If S and T are two distinct subsets of $\{0, 1, \ldots, n-1\}$, then S and T differ in a state i. If $i = 0$, then b^n distinguishes S and T, and if $i \geq 1$, then $a^{n-i}b^n$ distinguishes S and T.

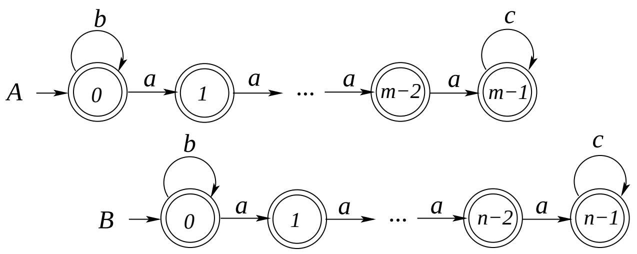


Figure 6: The incomplete DFAs of prefix-closed languages K and L with $\operatorname{nsc}(KL) = m+n$.

Next, the sets $\{q_i\} \cup S$ and $\{q_i\} \cup T$, where S and T are distinct subsets of $\{0,1,\ldots,n-1\}$, go to S and T, respectively, by c^m. Since S and T are distinguishable, the sets $\{q_i\} \cup S$ and $\{q_i\} \cup T$ are distinguishable as well.

Finally, notice that the string $b^n ab^n$ is accepted by the NFA N from each state q_i, but rejected from each state i in $\{0,1,\ldots,n-1\}$. Hence the sets $\{q_i\} \cup S$ and T, where S and T are subsets of $\{0,\ldots,n-1\}$, are distinguishable. Now let $0 \le i < j \le m-1$. Then $\{q_i\} \cup S$ and $\{q_j\} \cup T$ go to $\{q_{i+m-j}\} \cup S$ and T, respectively, by c^{m-j}. Since $\{q_{i+m-j}\} \cup S$ and T are distinguishable, the sets $\{q_i\} \cup S$ and $\{q_j\} \cup T$ are distinguishable as well. This proves the distinguishability of all the reachable subsets, and completes the proof. □

In the next theorem, we consider the nondeterministic case. For regular languages, the upper bound on the nondeterministic state complexity of concatenation is $m+n$, and it is tight in the binary case [10]. For prefix-closed languages, we get the same bound for concatenation. However, we define witness languages over a ternary alphabet.

Theorem 12. *Let $m,n \ge 3$. Let K and L be prefix-closed languages over an alphabet Σ with $\operatorname{nsc}(K) = m$ and $\operatorname{nsc}(L) = n$. Then $\operatorname{nsc}(KL) \le m+n$, and the bound is tight if $|\Sigma| \ge 3$.*

Proof. The upper bound is the same as for regular languages [10]. For tightness, consider the ternary prefix-closed languages K and L accepted by incomplete DFAs A and B, respectively, shown in Figure 6. Notice that if a string w is in KL, then w is in the language $b^*a^*c^*b^*a^*c^*$, and the number of a's in w is at most $(n+m-2)$.

For $i = 0,1,\ldots,m+n-1$, define the pair (x_i, y_i) as follows:

$$(x_i, y_i) = (a^i, a^{m-1-i}cba^{n-1}), \quad \text{for } i = 0,1,\ldots m-1,$$
$$(x_{m+j}, y_{m+j}) = (a^{m-1}cba^j, a^{n-1-j}), \quad \text{for } j = 0,1,\ldots n-1.$$

Let us show that the set of pairs $\mathscr{F} = \{(x_i, y_i) \mid i = 0,1,\ldots,m+n-1\}$ is a fooling set for the language KL.

(F1) For each i, we have $x_i y_i = a^{m-1}cba^{n-1}$. Thus $x_i y_i$ is in KL since $a^{m-1}c$ is in K and ba^{n-1} is in L.

(F2) Let $i < j$ and $(i,j) \ne (m-1,m)$. Then the number of a's in the string $x_j y_i$ is greater than $m+n-2$, and therefore the string $x_j y_i$ is not in KL. If $(i,j) = (m-1,m)$, then $x_m y_{m-1} = a^{m-1}cbcba^{n-1}$. Thus $x_m y_{m-1}$ is not in $b^*a^*c^*b^*a^*c^*$, and therefore it is not in KL.

Hence the set $\mathscr{F}$ is a fooling set for the language KL, so $\operatorname{nsc}(KL) \ge m+n$. □

6 Star and Reversal

We conclude our paper with the star and reversal operation on prefix-closed languages. The star of a language L is the language $L^* = \bigcup_{i \ge 0} L^i$, where $L^0 = \{\varepsilon\}$ and $L^{i+1} = L^i \cdot L$.

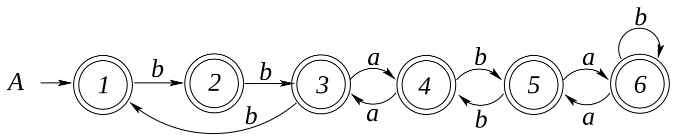


Figure 7: The incomplete DFA A of a prefix-closed language L with $\operatorname{isc}(L^*) = 2^{n-1}$; $n = 6$.

If a regular language L is accepted by a complete n-state DFA, then the language L^* is accepted by a DFA of at most $3/4 \cdot 2^n$ states, and the bound is tight in the binary case [20, 25].

For prefix-closed languages, the upper bound on the quotient complexity for star is $2^{n-2} + 1$, and it has been shown to be tight in the ternary case [5]. In the case of incomplete state complexity, we get the bound 2^{n-1}. For the sake of completeness, we give a simple proof of the upper bound using automata constructions. Moreover, we are able to define a witness language over a binary alphabet.

Theorem 13. *Let $n \geq 4$. Let L be a prefix-closed regular language over an alphabet Σ with $\operatorname{isc}(L) = n$. Then $\operatorname{isc}(L^*) \leq 2^{n-1}$, and the bound is tight if $|\Sigma| \geq 2$.*

Proof. Let $A = (Q, \Sigma, \cdot, s, Q)$ be an incomplete DFA for L. Construct an NFA A^* for L^* from the DFA A by adding the transition on a symbol a from a state q to the initial state s whenever the transition $q \cdot a$ is defined. In the subset automaton of the NFA A^*, each reachable set is either empty, or it contains the initial state s. It follows that $\operatorname{isc}(L^*) \leq 2^{n-1}$.

For tightness, consider the binary incomplete DFA with the state set $\{1, 2, \dots, n\}$, the initial state 1 and with all states final. The transitions are as follows. By a, the transitions in states 1 and 2 are undefined, each odd state i with $3 \leq i \leq n-1$ goes to $i+1$, and each even state i with $3 \leq i \leq n-1$ goes to $i-1$. By b, there is a cycle $(1, 2, 3)$, each odd state i with $4 \leq i \leq n-1$ goes to $i-1$, and each even state i with $4 \leq i \leq n-1$ goes to $i+1$. If n is odd, then n goes to itself by a, otherwise it goes to itself by b. The DFA for $n = 6$ is shown in Figure 7.

Notice that each state i with $3 \leq i \leq n$ has exactly one in-transition on a and on b. Denote by $a^{-1}(i)$ the state that goes to i on a, and by $b^{-1}(i)$ the state that goes to i on b.

Construct an NFA A^* as described above. Let us show that in the subset automaton of the NFA A^*, all subsets of $\{1, 2, \dots, n\}$ containing state 1 are reachable and pairwise distinguishable.

We prove reachability by induction on the size of subsets. The basis is $|S| = 1$, and the set $\{1\}$ is reachable since it is the initial state of the subset automaton. Assume that every set S containing 1 with $|S| = k$, where $1 \leqslant k \leqslant n-1$, is reachable. Let $S = \{1, i_1, i_2, i_3, \dots, i_k\}$, where $2 \leqslant i_1 < i_2 < \cdots < i_k \leqslant n$, be a set of size $k+1$. Consider three cases:

(i) $i_1 = 2$. Take $S' = \{1, b^{-1}(i_2), b^{-1}(i_3), \dots, b^{-1}(i_k)\}$. Then $|S'| = k$, and therefore S' is reachable by the induction hypothesis. Since we have $S' \xrightarrow{b} \{1, 2, i_2, \dots, i_k\} = S$, the set S is reachable.

(ii) $i_1 = 3$. Take $S' = \{1, 2, b^{-1}(i_2), b^{-1}(i_3), \dots, b^{-1}(i_k)\}$. Then $|S'| = k+1$ and S' contains states 1 and 2. Therefore, the set S' is reachable as shown in case (i). Since we have $S' \xrightarrow{b} \{1, 2, 3, i_2, i_3, \dots, i_k\} \xrightarrow{aa} \{1, 3, i_2, i_3, \dots, i_k\} = S$, the set S is reachable.

(iii) Let $i_1 = j \geq 3$, and assume that each set $\{1, j, i_2, \dots, i_k\}$ is reachable. Let us show that then also each set $\{1, j+1, i_2, \dots, i_k\}$ is reachable. If j is odd, then the set $\{1, j+1, i_2, \dots, i_k\}$ is reached from the set $\{1, j, a^{-1}(i_2), a^{-1}(i_3), \dots, a^{-1}(i_k)\}$ by a. If j is even, then the set $\{1, j+1, i_2, \dots, i_k\}$ is reached from the set $\{1, j, b^{-1}(i_2), b^{-1}(i_3), \dots, b^{-1}(i_k)\}$ by baa.

This proves reachability. To prove distinguishability, notice that the string $(ab)^{n-2}$ is accepted by the NFA A^* from state 3 since state 3 goes to the initial state 1 by $(ab)^{n-2}$ through the computation

$$3 \xrightarrow{ab} 5 \xrightarrow{ab} 7 \xrightarrow{ab} \cdots \xrightarrow{ab} n \xrightarrow{a} n \xrightarrow{b} n-1 \xrightarrow{ab} n-3 \xrightarrow{ab} \cdots \xrightarrow{ab} 4 \xrightarrow{ab} 1$$

if n is odd, and through a similar computation if n is even. On the other hand, the string $(ab)^{n-2}$ cannot be read from any other state $2i$ with $2 \le i \le n/2$ since we have

$$2i \xrightarrow{ab} \{2i-2,1,2\} \xrightarrow{ab} \{2i-4,1,2\} \xrightarrow{ab} \cdots \xrightarrow{ab} \{4,1,2\} \xrightarrow{a} \{3,1\} \xrightarrow{b} \{1,2\} \xrightarrow{ab} \emptyset,$$

thus $2i$ goes to the empty set by $(ab)^i$, so also by $(ab)^{n-2}$. If n is odd, then we have

$$2i+1 \xrightarrow{ab} \{2i+3,1,2\} \xrightarrow{ab} \{2i+5,1,2\} \xrightarrow{ab} \cdots \xrightarrow{ab} \{n,1,2\} \xrightarrow{a} \{n,1\} \xrightarrow{b} \{n-1,1,2\} \xrightarrow{ab}$$

$$\{n-3,1,2\} \xrightarrow{ab} \cdots \xrightarrow{ab} \{2i,1,2\} \xrightarrow{(ab)^i} \emptyset,$$

thus $2i+1$ goes to the empty set by $(ab)^{n-i}$, $i \ge 2$, and so also by $(ab)^{n-2}$. For n even, the argument is similar. The string $(ab)^{n-2}$ is not accepted from states 1 and 2. Hence the NFA A^* accepts the string $(ab)^{n-2}$ only from the state 3. Since there is exactly one in-transition on b in state 3, and it goes from state 2, the string $b(ab)^{n-2}$ is accepted by A^* only from state 2. Similarly, the string $bb(ab)^{n-2}$ is accepted by A^* only from state 1. Next, for similar reasons, the string $a(ab)^{n-2}$ is accepted only from 4, the string $ba(ab)^{n-2}$ is accepted only from 5, and in the general case, the string $(ab)^i a(ab)^{n-2}$ is accepted only from $4+2i$ ($i \ge 0$), and the string $(ba)^i(ab)^{n-2}$ is accepted only from $3+2i$ ($i \ge 1$). Hence for each state q of the NFA A^*, there exists a string w_q that is accepted by A^* only from the state q. It follows that all the subsets of the subset automaton of the NFA A^* are pairwise distinguishable since two distinct subsets differ in a state q, and the string w_q distinguishes the two subsets. This completes the proof. $\square$

We did some computations in the binary case. Having the files of n-state minimal binary pairwise non-isomorphic complete DFAs with a dead state and all the remaining states final, we computed the state complexity of the star of languages accepted by DFAs on the lists; here the state complexity of a regular language L, $\mathrm{sc}(L)$, is defined as the smallest number of states in any *complete* DFA for the language L. We computed the frequencies of the resulting complexities, and the average complexity of star. Our results are summarized in Table 2. Notice that for $n = 3,4,5$, there is just one language with $\mathrm{sc}(L) = n$ and $\mathrm{sc}(L^*) = 2$. Let us show that this holds for every n with $n \ge 3$.

$n\backslash\mathrm{sc}(L^*)$	1	2	3	4	5	6	7	8	9	average
2	-	2	-	-	-	-	-	-	-	2
3	8	1	6	-	-	-	-	-	-	1.866
4	161	1	48	30	6	-	-	-	-	1.857
5	4177	1	771	275	350	84	84	-	26	1.849

Table 1: The frequencies of the complexities and the average complexity of star on prefix-closed languages in the binary case; $n = 2,3,4,5$.

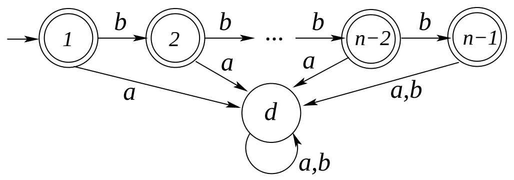


Figure 8: The only binary n-state complete DFA of a prefix-closed language L with $\mathrm{sc}(L^*) = 2$.

Proposition 14. *Let $n \geq 3$. There exists exactly one (up to renaming of alphabet symbols) binary prefix-closed regular language L with $\mathrm{sc}(L) = n$ and $\mathrm{sc}(L^*) = 2$.*

Proof. Let $A = (\{0,1\}, \{a,b\}, \delta, 0, F)$ be a minimal two-state DFA for the language L^*. Since L is prefix-closed, the language L^* is prefix-closed as well. It follows that state 0 is final, and state 1 is dead, thus $F = \{0\}$ and $\delta(1,a) = \delta(1,b) = 1$.

Without loss of generality, state 1 is reached from the initial state 0 by a, thus $\delta(0,a) = 1$.

Since $n \geq 3$, the language L contains a non-empty string. This means that the language L^* contains a non-empty string as well. Therefore, we must have $\delta(0,b) = 0$, and so $L^* = b^*$.

Now let B be the minimal n-state DFA for L. Then all the states of B are final, except for the dead state. Since $L^* = b^*$, no a may occur in any string of L. Hence each non-dead state of B must go to the dead state on a. Since all states must be reachable, we must have a path labeled by b^{n-2} and going through all the final states. The last final state must go to the dead state on b because otherwise all final states would be equivalent. The resulting n-state DFA B is shown in Figure 8. □

The reverse w^R of a string w is defined by $\varepsilon^R = \varepsilon$, and $(wa)^R = aw^R$ for a in Σ and w in Σ^*. The reverse of a language L is the language $L^R = \{w^R \mid w \in L\}$. If a regular language L is accepted by a complete n-state DFA, then the language L^R is accepted by a complete DFA of at most 2^n states [22, 25], and the bound is tight in the binary case [16, 19].

For prefix-closed languages, the quotient complexity of reversal is 2^{n-1} [5], and it follows from the results on ideal languages [4] since reversal commutes with complementation, and the complement of a prefix-closed language is a right ideal; here a language L is a right ideal if $L = L \cdot \Sigma^*$.

We restate the result for reversal in terms of incomplete state complexity, and prove tightness using a slightly different witness language.

Theorem 15. *Let $n \geq 2$. Let L be a prefix-closed regular language over an alphabet Σ with $\mathrm{isc}(L) = n$. Then $\mathrm{isc}(L^R) \leq 2^n - 1$, and the bound is tight if $|\Sigma| \geq 2$.*

Proof. Let A be an incomplete DFA for L. Construct an NFA A^R for the language L^R from the DFA A by swapping the role of the initial and final states, and by reversing all the transitions. The subset automaton of the NFA A^R has at most $2^n - 1$ non-empty reachable states, and the upper bound follows.

For tightness, consider the incomplete DFA A with all states final, shown in Figure 9. Construct an NFA A^R as described above. In the subset automaton of the NFA A^R, the initial state is $\{1,2,\ldots,n\}$. If S is a subset and if $i \in S$, then the subset $S \setminus \{i\}$ is reached from S by $a^i b a^{n-i}$. This proves the reachability of all non-empty subset by odd induction. Since the states of the subset automaton of any reversed DFA are pairwise distinguishable [7, 16, 22], the theorem follows.

□

Now, let us turn to the nondeterministic case. For regular languages, the tight bound for both star and reversal is $n+1$. It is met by a unary language for star [10], and by a binary language for reversal [12].

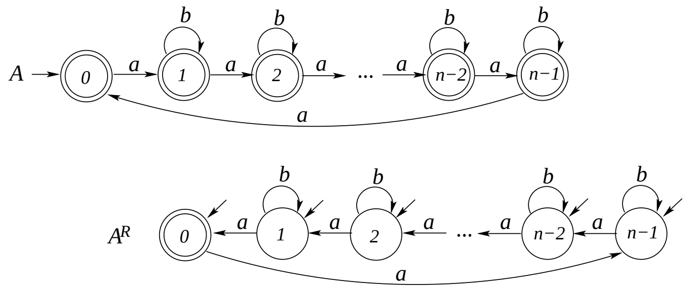


Figure 9: The incomplete DFA A of a language L with $\mathrm{isc}(L^R) = 2^n - 1$, and the NFA A^R.

For prefix-closed languages, we get the same bound for reversal. However, for star, the upper bound is n since every prefix-closed language contains the empty string, and there is no need to add a new initial state in the construction of an NFA for star. In the following theorem, we show that both these bounds are tight in the binary case.

Theorem 16. *Let $n \geq 2$. Let L be a prefix-closed language over an alphabet Σ with $\mathrm{nsc}(L) = n$. Then*
(1) $\mathrm{nsc}(L^*) \leq n$,
(2) $\mathrm{nsc}(L^R) \leq n+1$,
and both bounds are tight if $|\Sigma| \geq 2$.

Proof. (1) Let $N = (Q,\Sigma,\delta,s,F)$ be an n-state NFA for L. Since L is prefix-closed, the empty string is in L. Therefore, we can get an n-state NFA for the language L^* from the NFA N as follows: for each state q and each symbol a such that $\delta(q,a) \cap F \neq \emptyset$, we add a transition on a from q to the initial state s. Thus the upper bound is n.

For tightness, consider the prefix-closed language L accepted by the NFA A shown in Figure 10. Consider the set of pair of strings $\mathscr{F} = \{(a^i, a^{n-1-i}b) \mid i = 0,1,\dots,n-1\}$ of size n. Let us show that $\mathscr{F}$ is a fooling set for the language L^*.

(F1) We have $a^i a^{n-1-i} b = a^{n-1}b$. Since the string $a^{n-1}b$ is in L, it also is in L^*.

(F2) Let $i < j$. Then $a^i a^{n-1-j} b = a^{n-1-(j-i)}b$. Since no string $a^\ell b$ with $\ell < n-1$ is in L, the string $a^{n-1-(j-i)}b$ is not in L^*.

Hence the set $\mathscr{F}$ is a fooling set for the language L^*, and the lower bound follows.

(2) The upper bound is the same as for regular languages [10]. It is shown in [12, Theorem 2] that this bound is met by the binary prefix-closed language L accepted by the NFA shown in Figure 10. The proof in [12] is by a counting argument. Notice that Lemma 2 is satisfied for the language L^R with $\mathscr{A} = \{(ba^i, a^{n-1-i}) \mid i = 0,1,\dots,n-2\}, \mathscr{B} = \{(ba^{n-1}, ba^{n-1})\}, u = ba^{n-1}$, and $v = a$. This gives $\mathrm{nsc}(L^R) \geq n+1$ immediately. □

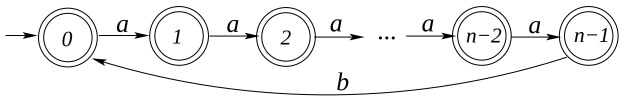


Figure 10: The NFA of a prefix-closed language L with $\mathrm{nsc}(L^*) = n$ and $\mathrm{nsc}(L^R) = n+1$.

		complement	$\|\Sigma\|$	intersection	$\|\Sigma\|$	union	$\|\Sigma\|$
isc	on prefix-closed	$n+1$	1	mn	2	$mn+m+n$	2
sc	on prefix-closed [5]	n	1	$mn-m-n+2$	2	mn	2
sc	on regular [20, 25]	n	1	mn	2	mn	2
nsc	on prefix-closed	2^n	3	mn	2	$m+n+1$	4
nsc	on regular [10, 12]	2^n	2	mn	2	$m+n+1$	2

Table 2: The complexity of boolean operations on prefix-closed and regular languages.

		concatenation	$\|\Sigma\|$	star	$\|\Sigma\|$	reversal	$\|\Sigma\|$
isc	on prefix-closed	$m2^{n-1}+2^n-1$	3	2^{n-1}	2	2^n-1	2
sc	on prefix-closed [5]	$(m+1)2^{n-2}$	3	$2^{n-2}+1$	3	2^{n-1}	2
sc	on regular [20, 25]	$m2^n-2^{n-1}$	2	$2^{n-1}+2^{n-2}$	2	2^n	2
nsc	on prefix-closed	$m+n$	3	n	2	$n+1$	2
nsc	on regular [10, 12]	$m+n$	2	$n+1$	2	$n+1$	2

Table 3: The complexity of concatenation, star, and reversal on prefix-closed and regular languages.

7 Conclusions

In this paper we considered operations on languages recognized by incomplete deterministic or nondeterministic finite automata with all states final. Our results are summarized in Tables 2 and 3. The results on quotient (state) complexity on prefix-closed languages are from [5], and the results for regular languages are from [10, 12, 20, 25]. Notice that in the nondeterministic case, our results are the same as in the general case of regular languages, except for the star operation. However, to prove tightness, we usually used larger alphabets than in the general case. Whether or not these bounds are tight also for smaller alphabets remains open.

References

[1] Alfred V. Aho, Jeffrey D. Ullman & Mihalis Yannakakis (1983): *On notions of information transfer in VLSI circuits*. In Johnson et al. [17], pp. 133–139. Available at `http://doi.acm.org/10.1145/800061.808742`.

[2] Jean-Camille Birget (1992): *Intersection and union of regular languages and state complexity*. *Inform. Process. Lett.* 43(4), pp. 185–190. Available at `http://dx.doi.org/10.1016/0020-0190(92)90198-5`.

[3] Jean-Camille Birget (1993): *Partial orders on words, minimal elements of regular languages and state complexity*. *Theoret. Comput. Sci.* 119(2), pp. 267–291. Available at `http://dx.doi.org/10.1016/0304-3975(93)90160-U`.

[4] Janusz A. Brzozowski, Galina Jirásková & Baiyu Li (2013): *Quotient complexity of ideal languages*. *Theoret. Comput. Sci.* 470, pp. 36–52. Available at `http://dx.doi.org/10.1016/j.tcs.2012.10.055`.

[5] Janusz A. Brzozowski, Galina Jirásková & Chenglong Zou (2014): *Quotient complexity of closed languages*. *Theory Comput. Syst.* 54(2), pp. 277–292. Available at `http://dx.doi.org/10.1007/s00224-013-9515-7`.

[6] Cezar Câmpeanu, Kai Salomaa & Sheng Yu (2002): *Tight lower bound for the state complexity of shuffle of regular languages*. *J. Autom. Lang. Comb.* 7(3), pp. 303–310.

[7] J.-M. Champarnaud, A. Khorsi & T. Paranthoën: *Split and join for minimizing: Brzozowski's algorithm*. Available at http://jmc.feydakins.org/ps/c09psc02.ps.

[8] Ian Glaister & Jeffrey Shallit (1996): *A lower bound technique for the size of nondeterministic finite automata*. *Inform. Process. Lett.* 59(2), pp. 75–77. Available at http://dx.doi.org/10.1016/0020-0190(96)00095-6.

[9] Edward A. Hirsch, Juhani Karhumäki, Arto Lepistö & Michail Prilutskii, editors (2012): *Computer Science - Theory and Applications - 7th International Computer Science Symposium in Russia, CSR 2012, Nizhny Novgorod, Russia, July 3-7, 2012. Proceedings*. *Lecture Notes in Computer Science* 7353, Springer. Available at http://dx.doi.org/10.1007/978-3-642-30642-6.

[10] Markus Holzer & Martin Kutrib (2003): *Nondeterministic descriptional complexity of regular languages*. *Internat. J. Found. Comput. Sci.* 14(6), pp. 1087–1102. Available at http://dx.doi.org/10.1142/S0129054103002199.

[11] Juraj Hromkovic (1997): *Communication complexity and parallel computing*. Springer. Available at http://dx.doi.org/10.1007/978-3-662-03442-2.

[12] Galina Jirásková (2005): *State complexity of some operations on binary regular languages*. *Theoret. Comput. Sci.* 330(2), pp. 287–298. Available at http://dx.doi.org/10.1016/j.tcs.2004.04.011.

[13] Galina Jirásková (2012): *Descriptional complexity of operations on alternating and boolean automata*. In Hirsch et al. [9], pp. 196–204. Available at http://dx.doi.org/10.1007/978-3-642-30642-6_19.

[14] Galina Jirásková & Tomáš Masopust (2011): *Complexity in union-free regular languages*. *Internat. J. Found. Comput. Sci.* 22(7), pp. 1639–1653. Available at http://dx.doi.org/10.1142/S0129054111008933.

[15] Galina Jirásková & Alexander Okhotin (2008): *State complexity of cyclic shift*. *RAIRO Theor. Inform. Appl.* 42(2), pp. 335–360. Available at http://dx.doi.org/10.1051/ita:2007038.

[16] Galina Jirásková & Juraj Šebej (2012): *Reversal of binary regular languages*. *Theoret. Comput. Sci.* 449, pp. 85–92. Available at http://dx.doi.org/10.1016/j.tcs.2012.05.008.

[17] David S. Johnson, Ronald Fagin, Michael L. Fredman, David Harel, Richard M. Karp, Nancy A. Lynch, Christos H. Papadimitriou, Ronald L. Rivest, Walter L. Ruzzo & Joel I. Seiferas, editors (1983): *Proceedings of the 15th Annual ACM Symposium on Theory of Computing, 25-27 April, 1983, Boston*. ACM.

[18] Jui-Yi Kao, Narad Rampersad & Jeffrey Shallit (2009): *On NFAs where all states are final, initial, or both*. *Theoret. Comput. Sci.* 410(47-49), pp. 5010–5021. Available at http://dx.doi.org/10.1016/j.tcs.2009.07.049.

[19] Ernst L. Leiss (1981): *Succint representation of regular languages by boolean automata*. *Theoret. Comput. Sci.* 13, pp. 323–330. Available at http://dx.doi.org/10.1016/S0304-3975(81)80005-9.

[20] A. N. Maslov (1970): *Estimates of the number of states of finite automata*. *Dokl. Akad. Nauk SSSR* 194, pp. 1266–1268 (Russian). English translation: Soviet Math. Dokl. **11** (1970) 1373–1375.

[21] Tomáš Masopust (2010): *Personal communication*.

[22] B. G. Mirkin (1970): *On dual automata*. *Kibernetika (Kiev)* 2, pp. 7–10 (Russian). Available at http://dx.doi.org/10.1007/BF01072247. English translation: Cybernetics **2**, (1966) 6–9.

[23] Narad Rampersad (2006): *The state complexity of L^2 and L^k*. *Inform. Process. Lett.* 98(6), pp. 231–234. Available at http://dx.doi.org/10.1016/j.ipl.2005.06.011.

[24] Michael Sipser (1997): *Introduction to the theory of computation*. PWS Publishing Company.

[25] Sheng Yu, Qingyu Zhuang & Kai Salomaa (1994): *The state complexities of some basic operations on regular languages*. *Theoret. Comput. Sci.* 125(2), pp. 315–328. Available at http://dx.doi.org/10.1016/0304-3975(92)00011-F.

Commutative Languages and their Composition by Consensual Methods[*†]

Stefano Crespi Reghizzi
Pierluigi San Pietro
DEIB, Politecnico di Milano and CNR-IEIIT
stefano.crespireghizzi@polimi.it pierluigi.sanpietro@polimi.it

Commutative languages with the semilinear property (SLIP) can be naturally recognized by real-time NLOG-SPACE multi-counter machines. We show that unions and concatenations of such languages can be similarly recognized, relying on – and further developing, our recent results on the family of consensually regular (CREG) languages. A CREG language is defined by a regular language on the alphabet that includes the terminal alphabet and its marked copy. New conditions, for ensuring that the union or concatenation of CREG languages is closed, are presented and applied to the commutative SLIP languages. The paper contributes to the knowledge of the CREG family, and introduces novel techniques for language composition, based on arithmetic congruences that act as language signatures. Open problems are listed.

1 Introduction

This paper focuses on commutative languages having the semilinear property (SLIP). We recall that a language has the *linear property* (LIP) if, in any word, the number of letter occurrences (also named Parikh image) satisfies a linear equation; it has the *semilinear property* (SLIP) [5] if the number satisfies one out of finitely many linear equations. A language is *commutative* (COM) if, for every word, all permutations are in the language; thus, the legality of a word is based only on the Parikh image, not on the positions of the letters. Here we deal with the subclass of COM languages enjoying the SLIP, denoted by COM-SLIP, for which we recall some known properties. For a binary alphabet, COM-SLIP languages are context-free whereas, in the general case, they can be recognized by *multi-counter machines* (MCM), in particular by non-deterministic quasi-real-time *blind* MCM (equivalent to *reversal-bounded* MCM [7]). The COM-SLIP family is closed under all Boolean operations, homomorphism and inverse homomorphism, but it is not closed under concatenation.

Our contribution is to relate two seemingly disparate language families: on one hand, the COM-SLIP languages and their closure under union and concatenation (denoted by COM-SLIP$^{\cup,\cdot}$), on the other hand, the family of *consensually regular* languages (CREG), recently introduced by the authors, to be later presented. We briefly explain the intuition behind it. Given a terminal alphabet, a CREG language is specified by means of a regular language (the *base*) having a *double* alphabet: the original one and a *dotted* copy. Two or more words in the base language *match*, if they are all identical when the dots are disregarded and, in every position, exactly one word has an undotted letter (thus in all remaining words the same position is dotted). In our metaphor, we say that, position by position, one of the base words "places" a letter and the remaining words "consent" to it. A word is in the consensual language if the base language contains a set of matching words, identical to the given word when the

[*]Work partially supported by *PRIN 2010LYA9RH-006* "Automi e linguaggi formali: Aspetti Matematici e Applicativi".
[†]The main results have been announced in [15], with preliminary sketchy proofs entirely superseded by the present ones.

dots are disregarded. This mechanism somewhat resembles the model of alternating non-deterministic finite automata, but the criterion by which the parallel computations match is more flexible and produces a recognition device which is a MCM working in NLOG-SPACE. This MCM can be viewed as a token or multi-set machine; it has one counter for each state of the DFA recognizing the base language; each counter value counts the number of parallel threads that are currently active in each state. Our main result is that the COM-SLIP$^{\cup,\cdot}$ family is strictly included in CREG; we also prove some non-closure properties of COM-SLIP$^{\cup,\cdot}$.

To construct the regular language that serves as base for the consensual definition of a COM-SLIP$^{\cup,\cdot}$ language, we have devised a new method, which may be also useful to study the inclusion in consensual classes of other families closed union or concatenation. It is easy to consensually specify a COM-LIP language by means of a regular base; however, in general, union or concatenation of two regular bases consensually specifies a larger language than the union or concatenation of the components. To prevent this to happen, we assign a distinct numeric congruence class to each base, which determines the positions where a letter may be placed as dotted or as undotted. For a given word, such positions are not the letter orders, but they are the orders of the letters in the projections of the word on each letter of the alphabet. The congruence acts as a sort of signature that cannot be mismatched with other signatures.

To hint to a potential application, COM-SLIP$^{\cup,\cdot}$ offers a rather suitable schema for certain parallel computation systems, such as Valiant's "bulk synchronous parallel computer" [16]. There, when all threads in a parallel computational phase, which we suggest to model by a commutative language, terminate, the next phase can start; the sequential composition of such phases can be represented by language concatenation; and the composition of alternative subsystems can be modeled by language union. As said, such computation schema is not finite-state but it is a MCM.

Paper organization: Sect. 2 contains preliminaries, some simple properties of COM-SLIP$^{\cup,\cdot}$ and the consensual model. Sect. 3 introduces the decomposed form, states and proves the conditions that ensure union- and concatenation-closure, and details the congruence based constructions. Sect. 4 proves the main result through a series of lemmas. The last section refers to related work and mentions some unanswered questions.

2 Preliminary Definitions and Properties

The terminal alphabet is denoted by $\Sigma = \{a_1, \dots, a_k\}$, the empty word by ε and $|x|$ is the length of a word x. The projection of x on $\Delta \subseteq \Sigma$ is denoted by $\pi_\Delta(x)$; $|x|_a$ is shorthand for $|\pi_{\{a\}}(x)|$ for $a \in \Sigma$, and $|x|_\Delta$ stands for $|\pi_\Delta(x)|$. The i-th letter of x is $x(i)$ and $x(i,j)$ is the substring $x(i)\dots x(j)$, $1 \le i \le j \le |x|$. The *shuffle* operation is denoted by $\sqcup\!\sqcup$.

The *Parikh image* or *vector* of a word $x \in \Sigma^*$ is $\Psi(x) = [|x|_{a_1}, \dots, |x|_{a_k}]$; it can be naturally extended to a language. The component-wise addition of two vectors is denoted by $\vec{p'} + \vec{p''}$. The *commutative closure* of $L \in \Sigma^*$ is $com(L) = \{x \in \Sigma^* \mid \Psi(x) \in \Psi(L)\}$. A language L is *commutative* if $com(L) = L$; the corresponding language family is named COM. A language $L \subseteq \Sigma^*$ has the *linear property* (LIP) if there exist $q+1 > 0$ vectors $\vec{c}, \vec{p}^{(1)}, \dots, \vec{p}^{(q)}$ over $\mathbb{N}^k$, (resp. the *constant* and the *periods*) such that $\Psi(L) = \left\{\vec{c} + n_1 \cdot \vec{p}^{(1)} + \dots + n_q \cdot \vec{p}^{(q)} \mid n_1, \dots, n_q \ge 0\right\}$.

A language has the *semilinear property* (SLIP) if it is the finite union of LIP languages. The families of commutative LIP/SLIP languages are denoted by *COM-LIP*/ *COM-SLIP*, respectively. It is well known that COM-SLIP is closed under the Boolean operations, inverse homomorphism, homomorphism and Kleene star, but not under concatenation, which in general destroys commutativity. However, the concatenation of COM-SLIP languages still enjoys the SLIP.

Let COM-SLIP$^{\cup,\cdot}$ be the smallest family including COM-SLIP languages and closed under union and concatenation. Let BLIND denote the class of languages accepted by nondeterministic, blind multi-counter machines [7], which, we recall, are restricted to perform a test for zero only at the end of a computation; they are equivalent to reversal-bounded counter machines. The following facts, although to our knowledge not stated in the literature, are straightforward.

Proposition 1. Main Properties of COM-SLIP$^{\cup,\cdot}$.

1. *Every COM-SLIP$^{\cup,\cdot}$ language on a binary alphabet is context-free.*
2. *COM-SLIP$^{\cup,\cdot}$ $\subsetneq$ BLIND.*
3. *The COM-SLIP$^{\cup,\cdot}$ family is not closed under intersection and Kleene star.*

Proof. Let $L' = com((ab)^+)$. Statement (1) is immediate: since all COM-SLIP on a a binary alphabet are context-free [9, 13], also their union and concatenation is context-free. Statement (2) is also immediate, since COM-SLIP is clearly included in BLIND, and BLIND is closed by union and concatenation. The inclusion is strict since BLIND includes also non-context-free languages on a binary alphabet [7]. To prove non-closure of intersection – Statement (3) – assume by contradiction that the language $L_0 = L' \cap a^+b^+ = \{a^nb^n \mid n > 0\}$ is in COM-SLIP$^{\cup,\cdot}$. Hence, also the languages $L_1 = \{a^+b^na^n \mid n > 0\}$, $L_2 = \{a^mb^ma^+ \mid m > 0\}$ and $L_1 \cap L_2 = \{a^nb^na^n \mid n > 0\}$ are in COM-SLIP$^{\cup,\cdot}$. But the latter language is not context-free, contradicting Statement (1). To complete the proof of Statement (3), if COM-SLIP$^{\cup,\cdot}$ were closed under Kleene star, then language $L_3 = (L'c)^*$ would be COM-SLIP$^{\cup,\cdot}$, with $c \notin \{a,b\}$. However, COM-SLIP$^{\cup,\cdot}$ is included in BLIND, which is an intersection-closed full semiAFL (see Section 5 of [1] and also Theorem 1 of [7]), i.e., BLIND is closed under intersection, union, arbitrary homomorphism, inverse homomorphism, and intersection with regular languages. Hence, the language $L_4 = L_3 \cap (a^+b^+c)^* = \{a^nb^nc \mid n > 0\}^*$ would be in BLIND. Letter c can be deleted by a homomorphism, hence also the language $\{a^nb^n \mid n > 0\}^*$, is BLIND, contradicting Corollary 3 of [1] and also Theorem 6, Part (2), of [7]. □

2.1 Consensual Languages.

We present the necessary elements of consensual language theory [2, 3]. Let $\mathring{\Sigma}$ be the *dotted* (or marked) copy of alphabet Σ. For each $a \in \Sigma$, $\tilde{a}$ denotes the set $\{a, \mathring{a}\}$. The alphabet $\widetilde{\Sigma} = \Sigma \cup \mathring{\Sigma}$ is named *double* (or internal). To express a sort of agreement between words over the double alphabet, we introduce a binary relation, called *match*, over $\widetilde{\Sigma}^*$.

Definition 1 (Match)**.** The partial, symmetrical, and associative binary operator, called *match*, $@ : \widetilde{\Sigma} \times \widetilde{\Sigma} \to \widetilde{\Sigma}$ is defined as follows, for all $a \in \Sigma$:

$$\begin{cases} a@\mathring{a} = \mathring{a}@a = a \\ \mathring{a}@\mathring{a} = \mathring{a} \\ \text{undefined} & \text{in every other case.} \end{cases}$$

The match is naturally extended to strings of equal length, as a letter-by-letter application, by assuming $\varepsilon@\varepsilon = \varepsilon$: for every $n > 1$, for all $w, w' \in \widetilde{\Sigma}^n$, if $w(i)@w'(i)$ is defined for every $i, 1 \le i \le n$, then

$$w \,@\, w' = \big(w(1)@w'(1)\big) \cdot \ldots \cdot \big(w(n)@w'(n)\big)\,. \quad \text{In every other case, } w@w' \text{ is undefined.}$$

Hence, the match is undefined on strings w, w' of unequal lengths, or else if there exists a position j such that $w(j)@w'(j)$ is undefined, which occurs in three cases: when both characters are in Σ, when both are in $\mathring{\Sigma}$ and differ, and when either one is dotted but is not the dotted copy of the other. Syntactically, the precedence of the match operator is just under the precedence of the concatenation. The match w of two or more strings is further qualified as *strong* if $w \in \Sigma^*$, or as *weak* otherwise. By Def. 1, if $w = w_1@w_2@\dots@w_m$ is a strong match of $m \geq 1$ words $w_1, \dots, w_m$, then in each position $1 \leq i \leq |w|$, exactly one word, say w_h, is undotted, i.e., $w_h(i) \in \Sigma$, and $w_j(i) \in \mathring{\Sigma}$ for all $j \neq h$; we say that word w_h *places* the letter at position i and the other words *consent* to it. Metaphorically, the words that strongly match provide mutual consensus on the validity of the corresponding word over Σ, thereby motivating the name "consensual" of the language family.
The match is extended to two languages B', B'' on the double alphabet, as $B'@B'' = \{w'@w'' \mid w' \in B', w'' \in B''\}$. The iterated match $B^{i@}$ is defined for all $i \geq 0$, as $B^{0@} = B$, $B^{i@} = B^{(i-1)@}@B$, if $i > 0$.

Definition 2 (Consensual language)**.** The *closure under match*, or *@-closure*, of a language $B \subseteq \widetilde{\Sigma}^*$ is $B^@ = \bigcup_{i \geq 0} B^{i@}$. The *consensual language with base B* is defined as $\mathscr{C}(B) = B^@ \cap \Sigma^*$. The family of *consensually regular* languages, denoted by CREG, is the collection of all languages $\mathscr{C}(B)$, such that the base B is regular.

It follows that a CREG language can be *consensually specified* by a regular expression over $\widetilde{\Sigma}$.

Example 1. The LIP language $L = \{a^n b^n c^n \mid n > 0\}$ is consensually specified by the base (that we may call a "consensual regular expression") $\mathring{a}^* a \mathring{a}^* \mathring{b}^* b \mathring{b}^* \mathring{c}^* c \mathring{c}^*$. For instance, *aabbcc* is the (strong) match of $\mathring{a}\, a\, \mathring{b}\, b\, \mathring{c}\; c$ and $a\, \mathring{a}\, b\; \mathring{b}\, c\, \mathring{c}$. The commutative closure of L is also in CREG, with base: $com(abc) \sqcup\!\sqcup \mathring{\Sigma}^*$.
Similarly, the COM-LIP language $L' = com\big((ab)^+\big) = \mathscr{C}(B_1)$, where $B_1 = com(ab) \sqcup\!\sqcup \mathring{\Sigma}^*$.
The COM-LIP language $L'' = com\big((abb)^+\big)$ is specified by the base $B_2 = com(abb) \sqcup\!\sqcup \mathring{\Sigma}^*$.
The languages $L' \cup L''$ and $L' \cdot L''$ are in CREG, but, counter to a naive intuition, they are not specified by the bases obtained by composition, respectively, $B_1 \cup B_2$ and $B_1 B_2$. In general $\mathscr{C}(B_1 \cup B_2) \supset \mathscr{C}(B_1) \cup \mathscr{C}(B_2)$: in the examples, $\mathscr{C}(B_1 \cup B_2)$ contains also undesirable "cross-matching" words, such as $ababb = ab\mathring{a}\mathring{b}\mathring{b}@\mathring{a}\mathring{b}abb$. A systematic compositional technique for obtaining the correct bases for the union and concatenation is the main contribution of this paper.

Summary of known and relevant CREG properties. *Language family comparisons*: CREG includes the regular languages, is incomparable with the context-free and deterministic context-free families, is included within the context-sensitive family, and it contains non-SLIP languages. CREG strictly includes the family of languages accepted by partially-blind multi-counter machines that are deterministic and quasi-real-time, as well as their union [4].
Closure properties: CREG is is closed under marked concatenation, marked iteration, inverse alphabetic homomorphism, reversal, and intersection and union with regular languages. The marked concatenation of two languages $L_1, L_2 \subseteq \Sigma^*$ is the language $L_1 \# L_2$, where $\# \notin \Sigma$, while the marked iteration of $L \subseteq \Sigma^*$ is the language $(L\#)^*$. A language family enjoying such properties is known as a *pre-Abstract Family of Languages* (see, e.g., [14]). A precise characterization of the bases that consensually specify regular languages is in [3]; an analysis of the reduction in descriptional complexity of the consensual base with respect to the specified regular language is in [2].
Complexity: CREG is in NLOGSPACE, i.e., NSPACE$(\log n)$ (often called NL): it can be recognized by a nondeterministic multitape Turing machine working in $\log n$ space. The recognizer of CREG languages is a special kind of nondeterministic, real-time multi-counter machine.

Useful notations for consensual languages. The following mappings will be used:

switching	$switch : \widetilde{\Sigma} \to \widetilde{\Sigma}$ where $switch(a) = \mathring{a}$, $switch(\mathring{a}) = a$, for all $a \in \Sigma$
marking	$dot : \widetilde{\Sigma} \to \mathring{\Sigma}$ where $dot(x) = x$, if $x \in \mathring{\Sigma}$, and $dot(x) = \mathring{a}$, if $x = a \in \Sigma$
unmarking	$undot : \widetilde{\Sigma} \to \Sigma$ where $undot(a) = switch(dot(a))$, for all $a \in \Sigma$.

These mappings are naturally extended to words and languages, e.g., given $x \in \widetilde{\Sigma}^*$, $switch(x)$ is the word obtained interchanging a and $\mathring{a}$ in x (a sort of "complement").

In the remainder of the paper, we assume that each base language is a subset of $\tilde{\Sigma}^* - \mathring{\Sigma}^+$, since words in $\mathring{\Sigma}^+$ are clearly useless in a match. Let B, B' be languages included in $\tilde{\Sigma}^+ - \mathring{\Sigma}^+$. We say that B is *unproductive* if $\mathscr{C}(B) = \emptyset$, and that the pair (B, B') is *unmatchable* if $B@B' = \emptyset$.

3 Consensual specifications composable by union and concatenation

Since it is unknown whether the whole CREG family is closed under union and concatenation, we first introduce a normal form, named decomposed,[1] of the base languages, which is convenient to ensure such closure properties. Second, we state two further conditions, named joinability and concatenability, for decomposed forms, and we prove that they, respectively, guarantee closure under union and concatenation. Such results hold for every consensual language, but the difficulty remains to find a systematic method for constructing base languages that meets such conditions. Third, in Sect. 3.1 we introduce an implementation of decomposed forms, relying on numerical congruences, that will permit us to prove in Sect. 4 that the $(\cup, \cdot)$-closure of commutative SLIP languages is in CREG.

Definition 3 (Decomposed form)**.** A base $B \subseteq \tilde{\Sigma}^* - \mathring{\Sigma}^+$ has the *decomposed form* if there exist a (disjoint) partition of B into two languages, named the *scaffold* sc and the *fill* fl of B, such that fl is unproductive, and the pair (sc, sc) is unmatchable.

The names scaffold and fill are meant to convey the idea of an arrangement superposed just once on each word of the base and, respectively, of an optional (but repeatable) component to complete the letters which are dotted in the scaffold. Three straightforward remarks follow. For every base B there exists a consensually equivalent decomposed base: it suffices to take as scaffold the language $\{a\, dot(y) \mid ay \in B, a \in \Sigma, y \in \widetilde{\Sigma}^*\}$, and as fill the language $\{dot(x)y \mid x \in \widetilde{\Sigma}, y \in \widetilde{\Sigma}^*, xy \in B\}$. For every $s \subseteq sc$, $f \subseteq fl$, the base $s \cup f$ is a decomposed form. The scaffold, but not the fill, may include words over Σ.

Consider a word $w \in \mathscr{C}(B)$. Since the fill is unproductive, its match closure cannot *place* all the letters of w and such letters must be placed by the scaffold. Since by definition the match closure of the scaffold alone is the scaffold itself, the following fundamental lemma immediately holds.

Lemma 1. *If $B = sc \cup fl$ is in decomposed form, as in Def. 3, then $\mathscr{C}(B) = \big(sc \cup (sc @ fl^{@})\big) \cap \Sigma^*$.*

Example 2. The table shows the decomposed bases of languages $com\big((ab)^+\big)$ and $com\big((abb)^+\big)$ of Sect. 2.1, considering for brevity only the case that the number of a's is a multiple of 3. Let $L' = com\left(\{a^{3n}b^{3n} \mid n \geq 1\}\right)$, with scaffold sc' and fill fl', and $L'' = com\left(\{a^{3n}b^{6n} \mid n \geq 1\}\right)$, with scaffold sc'' and fill fl'':

	scaffold	*fill*	*a strong match*
L'	$(a\mathring{a}a)^+ \sqcup\!\sqcup (b\mathring{b}b)^+$	$(\mathring{a}^3)^*\, \mathring{a}a\mathring{a}\, (\mathring{a}^3)^* \sqcup\!\sqcup (\mathring{b}^3)^*\, \mathring{b}b\mathring{b}\, (\mathring{b}^3)^*$	$\begin{array}{ccccccc} & a & b & \mathring{a} & a & \mathring{b} & b \in sc' \\ @ & \mathring{a} & \mathring{b} & a & \mathring{a} & b & \mathring{b} \in fl' \end{array}$
L''	$(\mathring{a}aa)^+ \sqcup\!\sqcup (\mathring{b}bb)^+$	$(\mathring{a}^3)^*\, a\mathring{a}\mathring{a}\, (\mathring{a}^3)^* \sqcup\!\sqcup (\mathring{b}^3)^*\, (b\mathring{b}\mathring{b})^2\, (\mathring{b}^3)^*$	$\begin{array}{cccccccccc} & \mathring{a} & \mathring{b} & a & a & b & b & \mathring{b} & b & b \in sc' \\ @ & a & b & \mathring{a} & \mathring{a} & \mathring{b} & \mathring{b} & b & \mathring{b} & \mathring{b} \in fl' \end{array}$

[1] In [4], we introduced the idea of a decomposed form for certain multi-counter machines, but that definition does not work for commutative languages.

Clearly, every word in sc' is unmatchable with every other word in sc', hence $sc'@sc' = \emptyset$. Similarly, every fill is unproductive. Every word in L' is the match of exactly one word in the scaffold with one or more words in the fill. Analogous remarks hold for L''.

Next, imagine to consensually specify two languages by bases in decomposed form $B' = sc' \cup fl'$ and $B'' = sc'' \cup fl''$. By imposing additional conditions on the bases, we obtain two very useful theorems about composition by union and concatenation.

Definition 4 (Joinability)**.** Two base languages B', B'' in decomposed form are *joinable* if their union $B' \cup B''$ is decomposed, with scaffold $sc' \cup sc''$ and fill $fl' \cup fl''$, and the pairs (sc', fl'') and (sc'', fl') are unmatchable.

Theorem 1 (Union of consensual languages in decomposed form)**.** *Let the base languages B', B'' be in decomposed form. If B' and B'' are joinable then $\mathscr{C}(B') \cup \mathscr{C}(B'') = \mathscr{C}(B' \cup B'')$.*

Proof. It suffices to prove the inclusion $\mathscr{C}(B' \cup B'') \subseteq \mathscr{C}(B') \cup \mathscr{C}(B'')$, since the opposite inclusion is obvious by Def. 2. Let $x \in \mathscr{C}(B)$. Since B is decomposed, by Lemma 1 it must be either $x \in sc@fl^{@}$ or $x \in sc$. In the latter case, x is in B' or in B'', and the inclusion follows. In the former case, there exist $n \geq 2$ words $w_1, w_2 \ldots, w_n$, with $n \leq |x|$, $w_1 \in sc$, $w_2, \ldots, w_n \in fl$ and $w_1@w_2@\ldots@w_n = x$. We claim that either $w_1 \in sc'$ and every other $w_i \in B'$, or $w_1 \in sc''$ and every other $w_i \in B''$, from which the thesis follows. Assume $w_1 \in sc'$ (the case $w_1 \in sc''$ is symmetrical). If there exists j, $2 \leq j \leq n$, such that $w_j \in fl''$ (with $w_j \notin \mathring{\Sigma}^+$), then $sc'@fl''$ is not empty (it includes at least $w_1@w_j$), a contradiction with the hypothesis that B' and B'' are joinable. □

Example 3. Returning to Ex. 2, we check that the two bases are joinable. The union of the bases is in decomposed form: $fl' \cup fl''$ is unproductive (because letters at positions 3, 6, ... cannot be placed); the pair (sc', sc'') is unmatchable, hence also $(sc' \cup sc'', sc' \cup sc'')$ is unmatchable. Moreover, (sc', fl''), and (sc'', fl') are unmatchable. Therefore $L' \cup L'' = \mathscr{C}(sc' \cup sc'' \cup fl' \cup fl'')$.

For concatenation, a similar, though more involved, reasoning requires a new technical definition.

Definition 5 (Dot-product $\odot$ and concatenability)**.** Let B', B'' be in decomposed form, and define their *dot-product* as $B' \odot B'' = (sc' \cdot sc'') \cup fl' \cup fl''$. B' and B'' are *concatenable* if $B' \odot B''$ is in decomposed form, with scaffold $sc' \cdot sc''$ and fill $fl' \cup fl''$, and the next two clauses hold for all words $w', w'' \in \widetilde{\Sigma}^+$, $y' \in sc'$, $y'' \in sc''$:

$$\exists x' \in fl' : w' = x' \cdot dot(y'') \ \wedge \ x'@y' \text{ is defined if, and only if, } w' \in fl' \wedge w'@y' \cdot y'' \text{ is defined} \quad (1)$$

$$\exists x'' \in fl'' : w'' = dot(y') \cdot x'' \ \wedge \ x''@y'' \text{ is defined if, and only if, } w'' \in fl'' \wedge w''@y' \cdot y'' \text{ is defined} \quad (2)$$

The two clauses are symmetrical. In loose terms, Clause (1) says that the fill fl' contains a word w' that matches $y'y''$, if, and only if, the word has a prefix x' , also in fl', which matches y', hence it is aligned with the point of concatenation. Therefore, the match $w'@y' \cdot y''$ does not produce a word that is illegal for $\mathscr{C}(B') \cdot \mathscr{C}(B'')$. This reasoning is formalized and proved next.

Theorem 2 (Concatenation of consensual languages in decomposed form)**.** *Let the bases B', B'' be in decomposed form. If B', B'' are concatenable, then $\mathscr{C}(B') \cdot \mathscr{C}(B'') = \mathscr{C}(B' \odot B'')$.*

Proof. Let $B = B' \odot B''$.
Case $\mathscr{C}(B') \cdot \mathscr{C}(B'') \subseteq \mathscr{C}(B)$. If $x \in \mathscr{C}(B') \cdot \mathscr{C}(B'')$, then $x = x'x''$ with $x' \in \mathscr{C}(B')$, $x'' \in \mathscr{C}(B'')$. Hence, x' is the strong match of one $w' \in sc'$ (resp. $w'' \in sc''$) with $n \geq 0$ words $w'_1, \ldots, w'_n \in fl' \subseteq fl$; analogously, x'' is the strong match of one $w'' \in sc''$ with $m \geq 0$ words $w''_1, \ldots w''_m \in fl''$. By definition of concatenability,

since for $1 \le i \le n$, every word w'_i is in fl', then also all words $w'_1 \cdot dot(w''), w'_2 \cdot dot(w''), \ldots$ are in fl', hence also in fl. Similarly, also $dot(w'') \cdot w''_1, \ldots dot(w'') \cdot w'_n$ are in fl''. Since $w' \cdot w''$ is in $sc'sc''$, it is possible to define a strong match yielding $x'x'' = x$, namely,

$$x = w'w''@\left(w'_1 \cdot dot(w'')\right)@\left(w'_2 \cdot dot(w'')\right)@\ldots\left(dot(w') \cdot w''_1\right)@\left(dot(w') \cdot w''_2\right)@\ldots$$

that is the concatenation of $w'@w'_1@\ldots@w'_n = x'$ with $w''@w''_1@\ldots@w''_m = x''$.

Case $\mathscr{C}(B) \subseteq \mathscr{C}(B') \cdot \mathscr{C}(B'')$. Let $x \in \mathscr{C}(B)$. Then there exist $n \ge 1$ words $w_1, w_2, \ldots, w_n$, with $n \le |x|$, such that $w_1@w_2@\ldots@w_n = x$, $w_1 \in sc' \cdot sc''$ and $w_2, \ldots, w_n \in fl' \cup fl''$. By definition, w_1 can be decomposed into $w_1 = w'_1w'_2$ for some $w'_1 \in sc', w''_2 \in sc''$. Let $q = |w'_1|$. Assume, by contradiction, that $x \notin \mathscr{C}(B') \cdot \mathscr{C}(B'')$. Since x is the match of word $w_1 = w'_1w'_2$ and words in $fl' \cup fl''$, the only possibility for w not being in $\mathscr{C}(B') \cdot \mathscr{C}(B'')$ is that there exists $j, 2 \le j \le n$, such that:

1. $w_j \in fl'$, and the substring $w_j(1,q) \notin fl'$, or
2. $w_j \in fl''$, and the substring $w_j(q+1,|x|) \notin fl''$.

We consider only Case (1) since the other is symmetrical. Since $w_j \in fl'$ and $w_j@w'_1w''_1$ is defined, then, by definition of concatenability, there exists $x' \in fl'$ such that $w_j = x' \cdot dot(w''_1)$, i.e., $w_j(1,q) = x'$, a contradiction with the assumption of Case (1). □

Example 4. Consider again Ex. 2. It is easy to check that the pair $(sc' \cdot sc'', sc' \cdot sc'')$ is unmatchable, for the same reason that (sc', sc'') is unmatchable. Then, we check that the bases $sc' \cup fl'$ and $sc'' \cup fl''$ are concatenable. We only discuss the case of Clause (1) since Clause (2) is symmetrical. Let $w' \in \widetilde{\Sigma}^+$, $y' \in sc'$, $fl'' \in sc''$. If there exists $x' \in fl'$ such that $w' = x'dot(y'')$, then obviously both $w' \in fl'$ and $w'@y' \cdot y''$ are defined.
For the converse case, assume that $w' \in fl'$ and $w'@y' \cdot y''$ is defined. Consider the projections $\alpha = \pi_{\widetilde{a}}(w')$, $\alpha' = \pi_{\widetilde{a}}(y') \in (a\mathring{a}a)^+$ and $\alpha'' = \pi_{\widetilde{a}}(y'') \in (\mathring{a}aa)^+$. Then $\alpha \in (\mathring{a}\mathring{a}\mathring{a})^*\mathring{a}a\mathring{a}(\mathring{a}\mathring{a}\mathring{a})^*$. Since $w'@y' \cdot y''$ is defined, the factor $\mathring{a}a\mathring{a}$ of α must be matched with a factor of $\alpha'\alpha''$: by its form and alignment, the only possibility is that it is matched with a factor of α'. Hence, α has the form $(\mathring{a}\mathring{a}\mathring{a})^*\mathring{a}a\mathring{a}(\mathring{a}\mathring{a}\mathring{a})^*dot(\alpha'')$. We omit the analogous reasoning for the projections on b. Since $w'@y' \cdot y''$ is defined, then w' must have the form $x' \cdot dot(y'')$ for some $x' \in fl'$. Therefore $L' \cdot L'' = \mathscr{C}(sc' \cdot sc'' \cup fl' \cup fl'')$. For instance

$$a^3b^3a^3b^6 = \begin{array}{l} a\mathring{a}ab\mathring{b}b \cdot \mathring{a}aa\mathring{b}bb\mathring{b}bb\ @ \\ \mathring{a}a\mathring{a}\mathring{b}b\mathring{b} \cdot \mathring{a}\mathring{a}\mathring{a}\mathring{b}\mathring{b}\mathring{b}\mathring{b}\mathring{b}\mathring{b}\ @ \\ \mathring{a}\mathring{a}\mathring{a}\mathring{b}\mathring{b}\mathring{b} \cdot a\mathring{a}\mathring{a}b\mathring{b}\mathring{b}b\mathring{b}\mathring{b} \end{array}$$

This example relies on a numerical congruence with module 3 for positioning the dotted and undotted letters. We shall see how to generalize this approach to handle words of any congruence class (with respect to the length of the projections on each letter). The generalization will carry the cost of taking larger values for the congruence module.

Incidentally, we observe that the theorems of this section may have a more general use than for commutative languages. Moreover, the theorems do not require the base languages to be regular; in fact, Def. 2 applies as well to non-regular bases (as a matter of fact [3] studies context-free/sensitive bases).

3.1 A Decomposed Form Relying on Congruences

Having stated some sufficient conditions for ensuring that the union/concatenation of two consensual languages can be obtained by composing (as described by Th. 1 and Th. 2) the corresponding base languages, we design a decomposed form, suitable for supporting joinability and concatenability, that uses module arithmetic for assigning the positions to the dotted and undotted letters within a word w over $\widetilde{\Sigma}$; the preceding examples offered some intuition for the next formal developments.[2] Loosely speaking, each decomposed base language is "personalized" by a sort of unique pattern of dotted/undotted letters, such that, when we want to unite or concatenate two languages, the match of two words with different patterns is undefined, thus ensuring that the union or catenation of the two decomposed bases specifies the intended language composition.

For every $a \in \Sigma$, consider the projection of w on $\widetilde{a} = \{a, \mathring{a}\}$ and, in there, the numbered positions of each a and $\mathring{a}$. Let m be an integer. By prescribing that for each base language, each undotted letter a may only occur in positions j characterized by a specified value of the congruence $j \mod m$, we make the bases decomposed. We need a new definition.

Definition 6 (Slots and modules)**.** Let $m > 3$, called *module*, be an even number. Let $R \subseteq \{1, \ldots, (m/2 - 1)\}$ be a nonempty set, called a *set of slots of module m*. For every $a \in \Sigma$, define a finite language $R_m(a) \subset \tilde{a}^m$, where only positions 1 and $r+1$ are dotted:

$$R_m(a) = \{\mathring{a}\, a^{r-1} \mathring{a}\, a^{m-r-1} \mid r \in R\} \tag{3}$$

The disjoint regular languages $sc\text{-}R_m, fl\text{-}R_m \widetilde{\Sigma}^*$ are defined as:

$$\begin{aligned} sc\text{-}R_m &= \left\{x \mid \forall a \in \Sigma, \pi_{\widetilde{a}}(x) \in (R_m(a) \cup a)^*\right\} && (4)\\ fl\text{-}R_m &= switch(sc\text{-}R_m) - \mathring{\Sigma}^*. && (5)\end{aligned}$$

The definition of $fl\text{-}R_m$ is clearly equivalent to $\left\{x \mid \forall a \in \Sigma, \pi_{\widetilde{a}}(x) \in (switch(R_m(a)) \cup \mathring{a})^*\right\} - \mathring{\Sigma}^*$. It is fairly obvious that $\mathscr{C}(B) = \Sigma^+$, since $\Sigma^+ \subseteq sc\text{-}R_m$. Also, $sc\text{-}R_m @ sc\text{-}R_m = \emptyset$ and $fl\text{-}R_m$ is unproductive. The following lemma is also obvious.

Lemma 2. *For all even numbers $m > 3$ and non-empty sets R of slots of module m, every base $E \subseteq sc\text{-}R_m \cup fl\text{-}R_m$ is in decomposed form, with scaffold: $E \cap sc\text{-}R_m$ and fill: $E \cap fl\text{-}R_m$.*

Example 5. Let $m = 6, R = \{1,2\}$ and $\Sigma = \{a,b\}$. Then

$$\begin{aligned} R_6(a) &= \{\mathring{a}\mathring{a}aaaa,\ \mathring{a}a\mathring{a}aaa\}\\ sc\text{-}R_6 &= (\mathbf{\mathring{a}\mathring{a}aaaa} \cup \mathbf{\mathring{a}a\mathring{a}aaa} \cup a)^* \sqcup\mkern-3mu\sqcup (\mathbf{\mathring{b}\mathring{b}bbbb} \cup \mathbf{\mathring{b}b\mathring{b}bbb} \cup b)^*\\ fl\text{-}R_6 &= \Big((\mathbf{aa\mathring{a}\mathring{a}\mathring{a}\mathring{a}} \cup \mathbf{a\mathring{a}a\mathring{a}\mathring{a}\mathring{a}} \cup \mathring{a})^* \sqcup\mkern-3mu\sqcup (\mathbf{bb\mathring{b}\mathring{b}\mathring{b}\mathring{b}} \cup \mathbf{b\mathring{b}b\mathring{b}\mathring{b}\mathring{b}} \cup \mathring{b})^*\Big) - \{\mathring{a}, \mathring{b}\}^*\end{aligned}$$

For clarity, in this example the characters in $sc\text{-}R_6$ and in $fl\text{-}R_6$, belonging to factors in $R_6(a), R_6(b)$, or $switch(R_6(a)), switch(R_6(b))$ respectively, are in bold. Examples of words in $\mathscr{C}(B)$ are:

$$a^6b^6 \in sc\text{-}R_6,\ \text{also } a^6b^6 = \begin{array}{l|l} \mathbf{\mathring{a}a\mathring{a}aaa\mathring{b}b\mathring{b}bbb}\ @ & \text{in } sc\text{-}R_6\\ \mathbf{a\mathring{a}a\mathring{a}\mathring{a}\mathring{a}b\mathring{b}b\mathring{b}\mathring{b}\mathring{b}} & \text{in } fl\text{-}R_6 \end{array}$$

$$a^9b^8 \in sc\text{-}R_6,\ \text{also } a^9b^8 = \begin{array}{l|l} \mathbf{\mathring{a}a\mathring{a}aaa}aaa\mathbf{\mathring{b}b\mathring{b}bbb}bb\ @ & \text{in } sc\text{-}R_6\\ \mathbf{a\mathring{a}a\mathring{a}\mathring{a}\mathring{a}}\mathring{a}\mathring{a}\mathring{a}\mathbf{b\mathring{b}b\mathring{b}\mathring{b}\mathring{b}}\mathring{b}\mathring{b} & \text{in } fl\text{-}R_6 \end{array}$$

$$(ab)^4aaabb \in sc\text{-}R_6,\ \text{also } (ab)^4aaabb = \begin{array}{l|l} \mathbf{\mathring{a}\mathring{b}ab\mathring{a}\mathring{b}abaa}a\mathbf{bb}@ & \text{in } sc\text{-}R_6\\ \mathbf{ab\mathring{a}\mathring{b}ab\mathring{a}\mathring{b}\mathring{a}\mathring{a}}\mathring{a}\mathbf{\mathring{b}\mathring{b}} & \text{in } fl\text{-}R_6 \end{array}$$

[2]As said, similar ideas have been used for a different language family in [4] and have been sketched for COM-SLIP languages in our communication [15].

To ensure that a base, included in $sc\text{-}R_m \cup fl\text{-}R_m$, can be used when two such languages are concatenated, we need the next simple concept.

Definition 7 (Shiftability). A language $R \subseteq \tilde{\Sigma}^*$ is *shiftable* if $R = \mathring{\Sigma}^* R \mathring{\Sigma}^*$.

This means that any word in R remains legal, when it is padded to the left/right with any dotted words.

Next we show that by taking disjoint sets of slots over the same module, we obtain two bases that are joinable; if, in addition, the fills are shiftable, the condition for concatenability is satisfied.

Theorem 3. *Let $m > 3$ and let R', R'' be two disjoint sets of slots of module m, and let $E' \subseteq sc\text{-}R'_m \cup fl\text{-}R'_m$ and $E'' \subseteq sc\text{-}R''_m \cup fl\text{-}R''_m$ be two bases. Then:*

- *E' and E'' are joinable;*
- *if the fills of E' and E'' are shiftable, then the fills of $E' \cup E''$ and $E' \odot E''$ are also shiftable, and E' and E'' are concatenable.*

Proof. Let $R = R' \cup R''$. Bases E' and E'' are in decomposed form by Lm. 2. Also $E' \cup E''$ and $E' \odot E''$ are in decomposed form, since they are both subsets of $sc\text{-}R_m \cup fl\text{-}R$.

Part (1): To show that E' and E'' are joinable, we only need to prove that $(fl\text{-}R''_m, sc\text{-}R'_m)$ is unmatchable (the case $(fl\text{-}R'_m, sc\text{-}R''_m)$ being unmatchable is symmetrical). By contradiction, assume that there exist $x \in fl\text{-}R''_m$ and $y \in sc\text{-}R'_m$ such that $x@y$ is defined. Let $a \in \Sigma$ be a letter occurring in $x \notin \mathring{\Sigma}^+$ and consider the projection $\alpha = \pi_{\tilde{a}}(x)$. By definition of $fl\text{-}R''_m$, there exist a position q of α and a value $r \in R''$ such that $\alpha(q) = \alpha(q + r'') = a$. Then, there exists $\alpha' \in \pi_{\tilde{a}}(y)$ such that $\alpha@\alpha'$ is defined. But in α' for all positions p, $1 \le p \le |\alpha'|$, if $\alpha'(p) = \mathring{a}$ then $\alpha'(p + r') = a$ for all $r' \notin R'$. Therefore, if $p = q$ then $\alpha(p + r) = \alpha'(p + r) = a$, which is impossible by definition of matching. The same argument could be applied to show that also the other two pairs are unmatchable.

Part (2): Define as $fl\text{-}E', sc\text{-}E'$ and as $fl\text{-}E'', sc\text{-}E''$ the fills and the scaffolds of E' and E'', respectively. If $fl\text{-}E'$ and $fl\text{-}E''$ are shiftable, then also the fill $fl\text{-}E' \cup fl\text{-}E''$ of both $E' \cup E''$ and $E' \odot E''$ is shiftable, since the union of two shiftable languages is shiftable. We now prove that in this case E', E'' are also concatenable. Let $w' \in fl\text{-}E', y' \in sc\text{-}E', y'' \in sc\text{-}E''$. If there exists $x' \in fl\text{-}E'$ such that $x'@y'$ is defined and $w' = x'dot(y'')$, then it is obvious that $w' \in fl\text{-}E' = \mathring{\Sigma}^* fl\text{-}E' \mathring{\Sigma}^*$ and that $w'@(y' \cdot y'')$ is defined. We are left to show that:

$$\text{if } w'@(y' \cdot y'') \text{ is defined then } \exists x' \in fl\text{-}E' \text{such that } w' = x'dot(y'') \text{ and } x'@y' \text{ is defined.} \tag{6}$$

The proof of Claim (6) requires another technical definition. Given a set R of slots with module m, for $a \in \Sigma$, for every $\alpha \in \pi_{\tilde{a}}(sc\text{-}R_m)$ a *restarting point* for projection α is a position i, $1 \le i \le |\alpha| - m$, such that $\alpha(i, i + m - 1) \in R_m(a)$. Hence, at i there is a factor in $R_m(a)$. A symmetrical definition holds if $\alpha \in \pi_{\tilde{a}}(fl\text{-}R_m)$: factor $\alpha(i, i + m - 1) \in switch(R_m(a))$. A restarting point always exists for all $\alpha \in \pi_{\tilde{a}}(sc\text{-}R_m)$ or $\alpha \in \pi_{\tilde{a}}(fl\text{-}R_m)$, provided that $\alpha \notin \Sigma^+$. We claim that if $s \in sc\text{-}R_m$, $f \in fl\text{-}\hat{R}_m$ for some (possibly equal) sets of slots $R, \hat{R}$ with module m, and the match $s@f$ is defined, then both the following conditions hold:

$$R \cap \hat{R} \neq \emptyset, \tag{7}$$

$$\forall a \in \Sigma, \text{ the set of restarting points for } \pi_{\tilde{a}}(f) \text{ is included in the set of restarting points for } \pi_{\tilde{a}}(s). \tag{8}$$

Since $f \notin \mathring{\Sigma}^*$, there exists at least one $a \in \Sigma$ such that $\pi_{\tilde{a}}(f)$ has a factor in $switch(\hat{R}_m(a))$ i.e., there exists a restarting point p for $\pi_{\tilde{a}}(f)$. For brevity, let $\alpha = \pi_{\tilde{a}}(f)$. Hence, $1 \le p \le |\alpha| - m$. Therefore, there exists $r \in \hat{R}$ such that $\alpha(p) = \alpha(p + r) = a$. Consider now $\beta = \pi_{\tilde{a}}(s)$. Since $s@f$ was assumed to be defined,

$\beta(p) = \beta(p+r) = \mathring{a}$. By definition of *sc*-$R_m$, $\beta \in (R_m(a) \cup a)^*$.
There are two possibilities: either p is a restarting point also for β, hence $r \in R$ and the above claims follow, or p is not a restarting point for β. The latter case is however impossible. In fact, in this case $p+r$ would be a restarting point for β, because of the form of $R_m(a)$. Therefore, since $\beta(p) = \mathring{a}$, there would be a restarting point also at position $p-r'$, for some $r' \in R$. However, both r, r', by definition, are smaller than $m/2$, therefore $2 \leq r + r' \leq m-2$. Hence, the restarting point at $p - r'$ would be at a distance less than m from the restarting point at $p+r$, which is impossible by definition of $R_m(a)$.

We prove Claim (6) to finish. For every $a \in \Sigma$, let $q'_a = |\pi_{\widetilde{a}}(y')|$, and let $q''_a = |\pi_{\widetilde{a}}(y')|$. Consider the rightmost restarting point p_a for $\pi_{\widetilde{a}}(w')$. By definition of *fl-E'*, there exists $r' \in R'$ such that $\pi_{\widetilde{a}}(w')(p_a, p_a+m) = a\mathring{a}^{r'-1}a\mathring{a}^{m-r'-1}$. By Claim (8), p_a is also a restarting point for $\pi_{\widetilde{a}}(y' \cdot y'')$: there exists $r \in R' \cup R''$ such that $\pi_{\widetilde{a}}(y'y'')(p_a, p_a+m) = \mathring{a}a^{r-1}\mathring{a}a^{m-r-1}$. We claim that $p_a \leq q'_a$. In fact, if $p_a > q_a$, then p_a must be a restarting point for y'', hence $r \in R''$: but $r = r'$, a contradiction with the hypothesis that $R' \cap R'' = \emptyset$. If $p_a \leq q'_a$ then p_a must be a restarting point for $\pi_{\widetilde{a}}(y')$, hence $r = r'$ and actually $p_a \leq q_a - m$. Since p_a is the rightmost restarting point, $\pi_{\widetilde{a}}(w')(p_a - m + 1, q'_a + q''_a) \in \mathring{\Sigma}^+$. Choose x' to be the prefix of w' such that such that $w' = x' dot(y'')$. $\square$

4 Commutative SLIP languages and their $(\cup, \cdot)$-closure

This section proves the main result:

Theorem 4 (Closure under union and concatenation)**.** *The family COM-SLIP*$^{\cup,\cdot}$ *is strictly included in the family of consensually regular languages: COM-SLIP*$^{\cup,\cdot} \subset$ *CREG.*

Every language in COM-SLIP$^{\cup,\cdot}$ can be defined by an expression that combines finitely many COM-SLIP languages, using union and concatenation; since COM-SLIP is the finite union of COM-LIP languages, we may assume that the expression includes only COM-LIP, rather than COM-SLIP, languages.

In the sequel, we prove that every COM-LIP language can be consensually defined in a decomposed form such that it permits to satisfy the additional assumptions needed for union and concatenation, hence all COM-SLIP$^{\cup,\cdot}$ languages are in CREG.

Decomposed form for COM-LIP languages To expedite handling the constant terms of LIP systems, we introduce a new operation *append* that combines a language and a commutative language, the latter penetrating into the former.

Definition 8 (Appending)**.** Let B be a language over the double alphabet $\widetilde{\Sigma}$. For $a \in \Sigma$, define the (unique) factorization

$$B = B_{\widetilde{a}} \cdot B_{\widetilde{\Sigma} - \widetilde{a}}$$

where $B_{\widetilde{a}} \subseteq \widetilde{\Sigma}^* \cdot \widetilde{a}$ and $B_{\widetilde{\Sigma}-\widetilde{a}} \subseteq \left(\widetilde{\Sigma} - \widetilde{a}\right)^*$ are languages, resp. ending by $\widetilde{a}$, and not using the letters $a, \mathring{a}$. If neither a nor $\mathring{a}$ occurs in B, let $B_{\widetilde{a}} = \varepsilon$. Let $A \subseteq a^+$; we define the operation, named *appending A to B*, as follows:

$$B \lhd A = B_{\widetilde{a}} \cdot (B_{\widetilde{\Sigma}-\widetilde{a}} \sqcup\!\sqcup A).$$

Given a commutative language $F \subseteq \Sigma^*$, $\Sigma = \{a_1, \ldots, a_k\}$, the iterative application of the previous operation to every letter of the alphabet (in any order) defines the operation, named *letter-by-letter appending F* to B, as:

$$B \lhd F = (\ldots(B \lhd \pi_{a_1}(F)) \lhd \pi_{a_2}(F)) \ldots) \lhd \pi_{a_k}(F).$$

To illustrate, we compute:

$$\begin{gathered}\{\mathring{a}b\mathring{a}\mathring{b}\} \lhd \{ac, ca\} = \Big(\{\mathring{a}b\mathring{a}\mathring{b}\} \lhd \pi_a\{ac, ca\}\Big) \lhd \pi_c\{ac, ca\} = \\ = \Big(\{\mathring{a}b\mathring{a}\mathring{b}\} \lhd \{a\}\Big) \lhd \{c\} = \Big(\{\mathring{a}b\mathring{a}\}(\mathring{b} \shuffle a)\Big) \lhd \{c\} = \\ = \{\mathring{a}b\mathring{a}\mathring{b}a, \mathring{a}b\mathring{a}a\mathring{b}\} \lhd \{c\} = \{\mathring{a}b\mathring{a}\mathring{b}a, \mathring{a}b\mathring{a}a\mathring{b}\} \shuffle \{c\}\end{gathered}$$

In the remainder of the Section, let L be a COM-LIP language over $\Sigma = \{a_1, \ldots, a_k\}$, $k > 0$, defined by constant $\vec{c}$ and periods $\mathscr{P} = \{\vec{p}^{(1)}, \ldots, \vec{p}^{(q)}\}$, for some $q > 0$, with the condition that for every $\vec{p} \in \mathscr{P}$, every component p_i is even.

The next definition introduces some sets, called X, Y, W, to define the COM-LIP language L with a base D in decomposed form. The assumption on each p_i being even will be lifted when defining COM-SLIP languages.

Definition 9. For all even integers $m \geq 4$, and for all sets of slots R of the form $\{r\}$ with $0 < r < m/2$, define the regular languages $X, Y, D \subseteq \widetilde{\Sigma}^*$ and the finite commutative language $W \subseteq \Sigma^*$, as follows:

$$X = \bigcup_{\vec{p} \in \mathscr{P}} \{x \in \mathit{fl}\text{-}R_m \mid \Psi(\pi_\Sigma(x)) = \vec{p}\} \tag{9}$$

$$Y = (R_m(a_1))^* \shuffle \ldots \shuffle (R_m(a_k))^* \tag{10}$$

$$\Psi(W) = \left\{\vec{c} + h_1 \cdot \vec{p}^{(1)} + \ldots + h_q \cdot \vec{p}^{(q)} \mid 0 \leq h_1, \ldots, h_q < m/2\right\}. \tag{11}$$

$$D = X \cup (Y \lhd W) \tag{12}$$

It is obvious that $X \subseteq \mathit{fl}\text{-}R_m$. To see that $Y \lhd W \subseteq \mathit{sc}\text{-}R_m$, we first describe relevant features of the formulae. By Eq. (11), W is the finite commutative language having as Parikh image the linear subspace included between $\vec{c}$ and $\vec{c} + (m/2-1)\vec{p}^{(1)} + \ldots + (m/2-1)\vec{p}^{(q)}$. For each a_i, the projection on a_i of a word in $Y \lhd W$ ends with a tail of undotted a_i's defined by Eq. (11). While the projection on a_i of $\mathit{sc}\text{-}R_m$ has necessarily length multiple of m, the tail does not need to comply with such constraint, thus allowing, in principle, the language $Y \lhd W$ to contain words whose projections on a_i has any length greater or equal to c_i (within the specified subspace). The following lemma is immediate:

Lemma 3. *Let X, Y, W, D as in Def. 9. Then, D is a decomposed base included in $\mathit{sc}\text{-}R_m \cup \mathit{fl}\text{-}R_m$, with $Y \lhd W \subseteq \mathit{sc}\text{-}R_m$ being the scaffold and $X \subseteq \mathit{fl}\text{-}R_m$ being the fill; moreover, the fill of D is shiftable, i.e., $X = \mathring{\Sigma}^* X \mathring{\Sigma}^*$.*

Example 6. Consider the language $L''_{even} = com\big((a^2b^4)^*\big)$ having the period $p_a = 2, p_b = 4$ and null constant. Notice that to obtain language $com\big((ab^2)^*\big)$, it is enough to apply union to L''_{even} and to the language $L''_{odd} = com\left(abb(a^2b^4)^*\right)$, which can be defined with the same period $p_a = 2, p_b = 4$, and with constant $c_a = 1, c_b = 2$. If module $m = 6$ and set of slots $R = \{2\}$ then $R_6(a) = \mathring{a}a\mathring{a}a^3$, $R_6(b) = \mathring{b}b\mathring{b}b^3$. Also, $\mathit{fl}\text{-}R_6 = \left(\left(a\mathring{a}a\mathring{a}^3 \cup \mathring{a}\right)^* \shuffle \left(b\mathring{b}b\mathring{b}^3 \cup \mathring{b}\right)^*\right) - \{\mathring{a}, \mathring{b}\}^*$. Let

$$\begin{aligned} X &= \{x \in \mathit{fl}\text{-}R_6 \mid \Psi\left(\pi_{\{a,b\}}(x)\right) = (2,4)\} \\ &= \left(\mathring{a}^* \cdot a\mathring{a}a\mathring{a}^3 \cdot \mathring{a}^*\right) \shuffle \left(\mathring{b}^* \cdot b\mathring{b}b\mathring{b}^3 \cdot \mathring{b}^* \cdot b\mathring{b}b\mathring{b}^3 \cdot \mathring{b}^*\right) \\ Y &= (R_6(a))^* \shuffle (R_6(b))^* = \left(\mathring{a}a\mathring{a}a^3\right)^* \shuffle \left(\mathring{b}b\mathring{b}b^3\right)^* \end{aligned}$$

Both X and Y satisfy Def. 9. To complete the base of language L''_{even}, we define

$$W = \bigcup_{0 \le i \le 2} com\left(a^{2i}b^{4i}\right)$$

The fill $\{\mathring{a},\mathring{b}\}^* X \{\mathring{a},\mathring{b}\}^*$ and the scaffold $Y \lhd W$ are a decomposed form for L''_{even}. Similarly, to define L''_{odd}, we have to define the sets X', Y', W'; for X', Y' we select as set of slots $R' = \{1\}$, which satisfies $R' \cap R = \emptyset$. At last, $W' = \bigcup_{0 \le i \le 2} com\left(abba^{2i}b^{4i}\right)$.

The important property of the language in Eq. (9) is stated next.

Lemma 4. *1. For all $n > 0$, for every $u \in X^{n@}$ there exist $q \ge 1$ integers $n_1, \ldots, n_q \ge 0$ with $n = n_1 + \ldots + n_q$ such that*

$$\Psi\left(\pi_\Sigma(u)\right) = n_1 \cdot \vec{p}^{(1)} + \ldots + n_q \cdot \vec{p}^{(q)}.$$

2. For all $n, n_1, \ldots, n_q \ge 0$, with $n_1 + \ldots + n_q = n$, if

$$u \in \textit{fl-}R_m \ \text{ and } \ \Psi\left(\pi_\Sigma(u)\right) = n_1 \cdot \vec{p}^{(1)} + \ldots + n_q \cdot \vec{p}^{(q)}$$

then $u \in X^{n@}$.

Proof. Part (1). By definition of X, if $x \in X$, then there exists $\vec{p}^j \in \mathscr{P}$, $1 \le j \le q$, such that $\Psi\left(\pi_{\Sigma(x)}\right) = \vec{p}^j$. By definition of match closure, there exists $n > 0$ words $x_1, \ldots x_n \in X$ such that $u = x_1@x_2@\ldots@x_n$. Then, for all $1 \le i \le n$, $\Psi(\pi_{\Sigma(x_i)} = \vec{p}^{j_i}$ for some j_i, with $1 \le j_i \le q$. Hence, $\Psi\left(\pi_\Sigma(u)\right) = \sum_{1 \le i \le n} \Psi(\pi_{\Sigma(x_i)})$, from which the thesis follows immediately. Part (2). By definition of X, for every vector $\vec{p}^j$, $1 \le j \le q$, language X includes all words x of *fl*-R_m such that $\Psi(\pi_{\Sigma(x)}) = \vec{p}^j$. Hence, one can always select n_1 words $x_1^{[1]}, \ldots, x_{n_1}^{[1]} \in X$, n_2 words $x_1^{[2]}, \ldots, x_{n_2}^{[2]} \in X$, etc., such that:

i) $\Psi\left(\pi_\Sigma\left(x_i^{[j]}\right)\right) = \vec{p}^j$, for every $1 \le j \le q$, $1 \le i \le n_j$;

ii) $x_1^{[1]}@\ldots@x_{n_2}^{[1]}@x_1^{[2]}@\ldots@x_{n_2}^{[2]}@\ldots@x_1^{[q]}@\ldots@x_{n_q}^{[q]} = u$. □

Lemma 5. *The consensual language $\mathscr{C}(D)$ is commutative.*

Proof. We notice first that $Y \lhd W$ and X obviously verify the following two conditions:

I) $Y \lhd W = \pi_{\widetilde{a}_1}(Y \lhd W) \sqcup \pi_{\widetilde{a}_2}(Y \lhd W) \sqcup \ldots \sqcup \pi_{\widetilde{a}_k}(Y \lhd W)$;

II) if $x \in X$ then $\pi_{\widetilde{a}_1}(x) \sqcup \pi_{\widetilde{a}_2}(x) \sqcup \ldots \sqcup \pi_{\widetilde{a}_k}(x) \subseteq X$.

Let $u \in \mathscr{C}(D)$ and let $v \in \Sigma^+$ be such that $\Psi(v) = \Psi(u)$. Word u is defined as $z@x_1@\ldots@x_n$, for some $z \in Y \lhd W$, $n > 0$ and some $x_1, \ldots, x_n \in X$. Word v is a permutation of u, hence for all $a_i \in \Sigma$ $\pi_{a_i}(u) = \pi_{a_i}(v)$. By Prop. (I) above, there exists a permutation z' of z, such that $z' \in sc\text{-}R_m \lhd W$, with $undot(z') = v$. Similarly, by Prop. (II) above, for all $1 \le j \le n$, there exists a permutation x'_j of x_j such that, for all $a_i \in \Sigma$, $\pi_{\widetilde{a}_i}(x'_j) = \pi_{\widetilde{a}_i}(x_j)$ and, moreover, such that $z'@x'_i$ is defined, with $\pi_{\widetilde{a}_i}(z'@x'_i) = \pi_{\widetilde{a}_i}(z@x_i)$. Hence, also $z'@x'_1@\ldots@x'_n$ is defined, therefore $z'@x'_1@\ldots@x'_n = undot(z') = v$. □

Next, Th. 5 shows that D consensually defines L, with m and r arbitrarily large.

Theorem 5. *For all even integers $m \ge 4$ and for every R of the form $\{r\}$, with $1 \le r \le m/2 - 1$, there exists a decomposed base D as in Def. 9 such that the COM-LIP language $L = \mathscr{C}(D)$.*

Proof. Let m,R,D,X,Y,W be defined as in Def. 9, with $k=|\Sigma|, q=|\mathscr{P}|$. We first notice that, by definition of $Y \lhd W$ and of X:
(*) if $z' \in Y$ then, for every $a_i \in \Sigma$, $|z'|_{\tilde{a}_i}$ is a multiple of m, $|z'|_{\mathring{a}_i} = 2\cdot|z|_{\tilde{a}_i}/m$ and $|z'|_{a_i} = (m-2)\cdot|z|_{\tilde{a}_i}/m$.
Proof of $\mathscr{C}(D) \subseteq L$. Let $u \in \mathscr{C}(D)$. We show that $\Psi(u) \in \Psi(L)$. Since D is in decomposed form, u must be the match of a word $z \in (Y \lhd W)$ with $h \geq 0$ words $x_1,\dots,x_h \in X$. Let $x = x_1@x_2@\dots@x_h$. Word z has the form $z' \lhd w$ for some $z' \in Y$ and some $w \in W \subseteq \Sigma^*$. By Lm. 4, Part (1), there exist $d_1,\dots,d_q \geq 0$ such that $\Psi(\pi_\Sigma(x)) = \vec{c} + d_1\cdot\vec{p}^{(1)} + \dots + d_q\cdot\vec{p}^{(q)}$. Also, by definition of W, there exist q integers $0 \leq h_1,\dots,h_q < m/2$ such that $\Psi(w) = \vec{c} + h_1\cdot\vec{p}^{(1)}\dots + h_q\cdot\vec{p}^{(q)}$. Since $u = (z' \lhd w)@x$ is a strong match, $\Psi(u) = \Psi(\pi_\Sigma(z')) + \Psi(\pi_\Sigma(x)) + \Psi(\pi_\Sigma(w))$. Notice that each component of $\Psi(\pi_\Sigma(x))$ must be even: by $(z' \lhd w)@x$ being a strong match it follows that $|x|_{a_i}$ is equal to $|z'|_{\mathring{a}_i}$, which is even. Again because $(z' \lhd w)@x$ is a strong match, $\Psi(\pi_\Sigma(z')) = (m-2)/2\cdot\Psi(\pi_\Sigma(x))$. Therefore:

$$\begin{aligned}\Psi(u) \quad &= (m-2)\cdot\Psi(\pi_\Sigma(x))/2 + \Psi(\pi_\Sigma(x)) + \Psi(w) = \\ &= m\cdot\Psi(\pi_\Sigma(x)) + \Psi(w) = \\ &= m\cdot(d_1\cdot\vec{p}^{(1)} + \dots + d_q\cdot\vec{p}^{(q)}) + \vec{c} + h_1\cdot\vec{p}^{(1)} + \dots + h_q\cdot\vec{p}^{(q)} = \\ &= \vec{c} + (m\cdot d_1 + h_1)\cdot\vec{p}^{(1)} + \dots + (m\cdot d_q + h_q)\cdot\vec{p}^{(q)}\end{aligned}$$

Hence, $\Psi(u) \in \Psi(L)$.
Proof of $L \subseteq \mathscr{C}(D)$. For all $u \in L$ there exist q integers $n_1,\dots,n_q$ such that $\Psi(u) = \vec{c} + n_1\cdot\vec{p}^{(1)} + \dots + n_q\cdot\vec{p}^{(q)}$. For every j, $1 \leq j \leq q$, let $h_j = n_j \bmod (m/2)$. Let $d_j = n_j - h_j$ if $n_j p_i^{(j)} > 0$, and $d_j = 0$ otherwise. Then, every d_j and h_j are such that $0 \leq h_j < m/2$ and d_j is a (possibly zero) multiple of $m/2$. By definition of W, there exists $w \in W$ such that $\Psi(w) = h_1\cdot\vec{p}^{(1)} + \dots + h_q\cdot\vec{p}^{(q)}$. For all $a_i \in \Sigma$, let z_i be the word in $(R_m(a_i))^*$ such that $|z_i| = d_1 p_i^{(1)} + \dots + d_q p_i^{(q)}$. Such a word does exist, since each d_j is a (possibly zero) multiple of $m/2$, hence $d_1 p_i^{(1)} + \dots + d_q p_i^{(q)}$ is a multiple of $m/2$; if this multiple is 0, then $z_i = \varepsilon$. By definition of $R_m(a_i)$, word z_i (when not empty) has, in every segment of length m belonging to $R_m(a_i)$, exactly two occurrences of $\mathring{a}_i$, and $(m-2)$ occurrences of a_i. Hence, $|z_i|_{\mathring{a}_i} = 2(d_1 p_i^{(1)} + \dots + d_q p_i^{(q)})/m$ and $|z_i|_{a_i} = (m-2)\cdot(d_1 p_i^{(1)} + \dots + d_q p_i^{(q)})/m$. We claim that there exists $z' \in Y$ such that $\Psi(\mathit{undot}(z')) = d_1\cdot\vec{p}^{(1)} + \dots + d_q\cdot\vec{p}^{(q)}$. In fact, by Prop. (*) above, there exists $z' \in Y$ such that $\pi_{\tilde{a}_i}(z') = z_i$. Hence, $\Psi(\pi_\Sigma(z')) = (m-2)\cdot(d_1\cdot\vec{p}^{(1)} + \dots + d_q\cdot\vec{p}^{(q)}$. By definition of W, there exists $w \in W$ such that

$$\Psi(w) = \vec{c} + h_1\cdot\vec{p}^{(1)} + \dots + h_q\cdot\vec{p}^{(q)}.$$

Let $z'' = \mathit{switch}(z')$. By Lm. 4, Part (2), there exist $n = 2d_1/m + 2d_2/m + \dots + 2d_q/m$ words $x_1,\dots,x_n \in X$ such that

$$z'' = x_1@\dots@x_n, \text{ with } \Psi(\pi_\Sigma(z'')) = 2\cdot(d_1\cdot\vec{p}^{(1)} + \dots + d_q\cdot\vec{p}^{(q)})/m.$$

Consider now $x_i \lhd \mathit{dot}(w)$. This word is in X, since the fills included in X may end with arbitrarily many $\mathring{a}$, for every $a \in \Sigma$. Clearly, from $x_i \lhd \mathit{dot}(w)$ one can obtain a strong match v with $z' \lhd w$:

$$\begin{gathered} v = (z' \lhd w)@(x_1 \lhd \mathit{dot}(w))@\dots@(x_n \lhd \mathit{dot}(w)) \\ \text{with } \Psi(v) = \Psi(\pi_\Sigma(z')) + \Psi(\pi_\Sigma(z'')) + \Psi(\pi_\Sigma(w)) = \Psi(u). \end{gathered}$$

Since the language $\mathscr{C}(D)$ is commutative, and $v \in \mathscr{C}(D)$, also $u \in \mathscr{C}(D)$. □

We can now complete the proof of Th. 4. Since a COM-SLIP language is the finite union of COM-LIP languages, a COM-SLIP$^{\cup,\cdot}$ language is the union and concatenation of COM-LIP languages. It can be assumed that these COM-LIP languages comply with Def. 9 having only even components in every vector of the set $\mathscr{P}$ of periods (since otherwise they can be represented as the finite union of COM-LIP languages with this property). Select the same module and disjoint sets of slots for the decomposed bases of these COM-LIP languages. By Th. 3, since each COM-LIP is defined by a shiftable base with disjoint sets of slots, the various bases can be combined with $\cup$ and $\odot$, resulting in a shiftable base. By Th. 1 and and Th. 2, the result is still a consensual language (with a decomposed base). The inclusion is strict, since language $\{ba^1ba^2ba^3\dots ba^k \mid k \geq 1\}$ has a non-SLIP commutative image, but it is in CREG [2].

5 Related Work and Conclusion

By classical results, COM-SLIP$^{\cup,\cdot}$ is included in the class of languages recognized by *reversal-bounded* multi-counter machines [1, 8] (which is also closed under concatenation). The latter class admits different, but equivalent, characterizations: as the class of languages recognized by (nondeterministic) *blind MCMs'* [7], or as the minimal, intersection-closed full semi-AFL including language $com((ab)^*)$ [1, 6]. However, the cited papers are not concerned with actual construction methods for the MCMs'.

Although COM-SLIP languages have been much studied, we are not aware of any specific study on the effect on COM-SLIP of operations such as concatenation.

Concerning the techniques to specify COM-SLIP languages, our specification, using as patterns the commutative Parikh vectors, bears some similarity to Kari's [10] “scattered deletion” operation.

It is known that family COM-SLIP, when restricted to a binary alphabet, is context-free [9, 13], therefore it enjoys closure under concatenation and star. On the other hand, we observe that the intersection $I = L'^4 \cap a^+L'^2b^+$, where $L' = com((ab)^+)$, is not context-free, since

$$I \cap (a^+b^+)^4 = \{a^nb^na^nb^na^nb^na^nb^n \mid n > 1\}.$$

In [13], the context-free grammar rules for COM-LIP again resemble our consensual specification.

Also, the context-sensitive grammars in [11], obtained by adding *permutative rules* of the form $AB \to BA$ to context-free grammars, include COM-SLIP and of course its closure by concatenation and star, but not its intersection with regular languages.

Last, the COM-SLIP languages are included in the SLIP language family recognized by a formal device, based on so called restarting automata, studied in [12], but the grounds covered by CREG and by that family are quite different. Beyond the mentioned similarities, we are unaware of anything related to our congruence-based decomposed form.

Unanswered questions This paper has added a piece to our knowledge of the languages included in CREG; it has introduced a novel compositional construction for the union/concatenation, which is very general and hence likely to be useful for other language subfamilies included in CREG. Some natural questions concern the closures of COM-SLIP under other basic operations: is the intersection of two COM-SLIP languages, or the Kleene star of a COM-SLIP language, in CREG?

A different kind of problem is whether the only commutative languages that are in CREG are semilinear; for instance, the nonsemilinear non-commutative language $\{ba^1ba^2ba^3\dots ba^k \mid k \geq 1\}$ is in CREG, but, for its commutative closure, we do not know of a consensually regular specification. Last, a more general problem is whether CREG is closed under union, concatenation, and star. A possible approach is to investigate whether every CREG language may be defined by a base which is joinable and shiftable, thus obtaining closure under union and concatenation by virtue of the lemmas presented in this paper.

References

[1] Brenda S. Baker & Ronald V. Book (1974): *Reversal-bounded multipushdown machines*. *Journal of Computer and System Sciences* 8(3), pp. 315 – 332, doi:10.1016/S0022-0000(74)80027-9.

[2] Stefano Crespi Reghizzi & Pierluigi San Pietro (2011): *Consensual languages and matching finite-state computations*. *RAIRO - Theor. Inf. and Applic* 45(1), pp. 77–97, doi:10.1051/ita/2011012.

[3] Stefano Crespi-Reghizzi & Pierluigi San Pietro (2012): *Strict Local Testability with Consensus Equals Regularity*. In Nelma Moreira & Rogério Reis, editors: *CIAA*, *Lecture Notes in Computer Science* 7381, Springer, pp. 113–124, doi:10.1007/978-3-642-31606-7_10.

[4] Stefano Crespi Reghizzi & Pierluigi San Pietro (2013): *Deterministic Counter Machines and Parallel Matching Computations*. In Stavros Konstantinidis, editor: *Impl. and Appl. of Automata - 18th Int. Conf., CIAA 2013, Halifax, Nova Scotia, Canada, July 16-19, 2013.*, *Lecture Notes in Computer Science* 7982, Springer, pp. 280–291, doi:10.1007/978-3-642-39274-0_25.

[5] Seymour Ginsburgh (1966): *The mathematical theory of context-free languages*. McGraw-Hill.

[6] Sheila A. Greibach (1976): *Remarks on the complexity of nondeterministic counter languages*. *Theor. Comp. Sc.* 1(4), pp. 269–288, doi:10.1016/0304-3975(76)90072-4.

[7] Sheila A. Greibach (1978): *Remarks on Blind and Partially Blind One-Way Multicounter Machines*. *Theor. Comput. Sci.* 7, pp. 311–324, doi:10.1016/0304-3975(78)90020-8.

[8] Oscar H. Ibarra (1978): *Reversal-Bounded Multicounter Machines and Their Decision Problems*. *J. ACM* 25(1), pp. 116–133, doi:10.1145/322047.322058.

[9] Michel Latteux (1979): *Cônes rationnels commutatifs*. *J. Comput. Syst. Sci.* 18(3), pp. 307–333, doi:10.1016/0022-0000(79)90039-4.

[10] Alexandru Mateescu (1994): *Scattered deletion and commutativity*. *Theor. Comp. Sc.* 125(2), pp. 361–371, doi:10.1016/0304-3975(94)90259-3.

[11] Benedek Nagy (2009): *Languages Generated by Context-Free Grammars Extended by Type AB -> BA Rules*. *Journal of Automata, Languages and Combinatorics* 14(2), pp. 175–186.

[12] Benedek Nagy & Friedrich Otto (2012): *On CD-systems of stateless deterministic R-automata with window size one*. *J. Comput. Syst. Sci* 78(3), pp. 780–806, doi:10.1016/j.jcss.2011.12.009.

[13] Michel Rigo (2003): *The commutative closure of a binary slip-language is context-free: a new proof*. *Discrete Appl. Math.* 131(3), pp. 665–672, doi:10.1016/S0166-218X(03)00335-4.

[14] Arto Salomaa (1987): *Formal languages*. Academic Press, San Diego, CA, USA.

[15] Stefano Crespi Reghizzi & Pierluigi San Pietro (2013): *Commutative consensual counter languages. Talk given at ICTCS 2013,* 14th Italian Conference on Theoretical Computer Science, Palermo, Italia, Sept. 9-11, 2013.

[16] Leslie G. Valiant (1990): *A bridging model for parallel computation*. *Comm. ACM* 33(8), p. 103, doi:10.1145/79173.79181.

Similarity density of the Thue-Morse word with overlap-free infinite binary words

Chen Fei Du and Jeffrey Shallit
School of Computer Science, University of Waterloo, Waterloo, ON N2L 3G1, Canada
cfdu@uwaterloo.ca,
shallit@uwaterloo.ca

We consider a measure of similarity for infinite words that generalizes the notion of asymptotic or natural density of subsets of natural numbers from number theory. We show that every overlap-free infinite binary word, other than the Thue-Morse word $\mathbf{t}$ and its complement $\bar{\mathbf{t}}$, has this measure of similarity with $\mathbf{t}$ between $\frac{1}{4}$ and $\frac{3}{4}$. This is a partial generalization of a classical 1927 result of Mahler.

1 Introduction

The Thue-Morse word

$$\mathbf{t} = \mathtt{011010011001011010010110011010001}\cdots$$

is one of the most studied objects in combinatorics on words. It can be defined in a number of different ways, such as the fixed point of the morphism μ defined by $\mu(\mathtt{0}) := \mathtt{01}$ and $\mu(\mathtt{1}) := \mathtt{10}$ beginning with 0, or as the word whose n^{th} position is the number of 1s (modulo 2) in the binary representation of n.

The word $\mathbf{t}$ has a large number of interesting properties, many of which are covered in the survey [1]. For example, $\mathbf{t}$ is *overlap-free*: it contains no factor of the form $axaxa$, where x is a (possibly empty) word and a is a single letter. One that concerns us here is the following "fragility" property [4]: if the bits in any *finite* non-empty set of positions are "flipped" (i.e., changed to their binary complement) in the Thue-Morse word, the resulting word is no longer overlap-free.[1]

Of course, this is not true of arbitrary *infinite* sets of positions; for example, we can transform $\mathbf{t}$ to $\bar{\mathbf{t}}$ by flipping *all* the positions. Chao Hsien Lin (personal communication, October 2013) raised the following natural question.

Problem 1. Is it possible to flip an *infinite*, but density 0, set of positions in $\mathbf{t}$ and still get an overlap-free word?

Our main result (Theorem 18) solves Problem 1 in the negative. After making precise what we mean by "density", we use a certain automaton [10] encoding all the overlap-free infinite binary words to compare $\mathbf{t}$ to all other overlap-free infinite binary words and show that they differ from $\mathbf{t}$ in at least density $\frac{1}{4}$ of the positions. Furthermore, computational evidence suggests that the true lower bound is density $\frac{1}{3}$. However, we were unable to obtain a proof of this tighter bound. Finally, we consider the possibility of similar results holding for other words (in place of $\mathbf{t}$) or for larger classes of words (in place of overlap-free words).

[1]Note that the "fragility" property does not hold for an arbitrary overlap-free binary word; for example, both $\mathtt{0}\mathbf{t}$ and $\mathtt{1}\mathbf{t}$ are overlap-free. There are even overlap-free words in which blocks arbitrarily far from the beginning may be flipped and still remain overlap-free [10].

2 Notation

We observe the following notational conventions throughout this paper. We let $\mathbb{N} := \{0, 1, 2, \dots\}$ denote the natural numbers. The upper-case Greek letters Σ, Δ, Γ represent finite alphabets. For each $n \in \mathbb{N}$, we let $\Sigma_n := \{0, 1, 2, \dots, n-1\}$.

As usual, Σ^ω denotes the set of all (right-)infinite words over Σ and $L^\omega := \{x_0 x_1 x_2 \cdots \ : \ x_i \in L \setminus \{\varepsilon\}\}$ denote the set of all infinite words formed by concatenation from nonempty words of L. By x^ω we mean the infinite periodic word $xxx\cdots$.

We adopt the convention that, in the context of words, lower-case letters such as x, y, z refer to finite words (i.e., $x, y, z \in \Sigma^*$), while boldface letters $\mathbf{x}, \mathbf{y}, \mathbf{z}$ refer to infinite words (i.e., $\mathbf{x}, \mathbf{y}, \mathbf{z} \in \Sigma^\omega$).

To be consistent with $0 \in \mathbb{N}$, all words are zero-indexed, i.e., the first letter of the word is in position 0. For $x \in \Sigma^*$ and $m \leq n \in \mathbb{N}$, $x[n]$ denotes the letter at the n^{th} position of x and $x[m..n]$ denotes the subword consisting of the letters from the m^{th} through n^{th} positions (inclusive) of x. For $x \in \Sigma_2^*$, $\overline{x}$ denotes the binary complement of x, i.e., the word obtained by changing all $\mathtt{0}$s to $\mathtt{1}$s and vice versa. We use the same notation just described for infinite words. In addition, for $\mathbf{x} \in \Sigma^\omega$ and $n \in \mathbb{N}$, $\mathbf{x}[n..\infty]$ denotes the (infinite) suffix of $\mathbf{x}$ starting from the n^{th} position of $\mathbf{x}$.

For a morphism $g : \Sigma^* \to \Sigma^*$ and $n \in \mathbb{N}$, we let g^n denote the n-fold composition of g, and $g^\omega : \Sigma^* \to \Sigma^\omega$ denote $\lim_{n\to\infty} g^n$ if the limit exists. The Thue-Morse morphism $\mu : \Sigma_2^* \to \Sigma_2^*$ is defined by $\mu(\mathtt{0}) := \mathtt{01}$ and $\mu(\mathtt{1}) := \mathtt{10}$. Iterates of the Thue-Morse morphism acting on $\mathtt{0}$ are denoted by $t_n := \mu^n(\mathtt{0})$. Note that $\mathbf{t} = \mu^\omega(\mathtt{0})$.

3 Similarity density of words

Let us express Problem 1 in another way: how similar can an arbitrary overlap-free word $\mathbf{w}$ be to $\mathbf{t}$? For $\mathbf{w}$ a shift of $\mathbf{t}$, this was essentially determined by the following result from a surprisingly little-known 1927 paper of Kurt Mahler on autocorrelation [7].

Theorem 2. *For all $k \in \mathbb{N}$, the limit*

$$\sigma(k) := \lim_{n\to\infty} \frac{1}{n} \sum_{i=0}^{n-1} (-1)^{\mathbf{t}[i]+\mathbf{t}[i+k]}$$

exists. Furthermore, we have $\sigma(0) = 1$, $\sigma(1) = -\frac{1}{3}$, and for all $n \in \mathbb{N}$, $\sigma(2n) = \sigma(n)$ and $\sigma(2n+1) = -\frac{1}{2}(\sigma(n) + \sigma(n+1))$.

(Also see [11, 12].) Then an easy induction on k gives

Corollary 3. *For all $k \in \mathbb{N} \setminus \{0\}$, $-\frac{1}{3} \leq \sigma(k) \leq \frac{1}{3}$.*

Mahler's result is not exactly what we want, but we can easily transform it. Rather than autocorrelation, we are more interested in a quantity we call "similarity density"; it measures how similar two words of the same length are, with a simple and intuitive definition for finite words that generalizes to infinite words by way of limits.

Definition 4. We interpret the Kronecker delta as a function of two variables $\delta : \Sigma^2 \to \Sigma_2$ as follows.

$$\delta(a, b) := \begin{cases} 0, & \text{if } a \neq b; \\ 1, & \text{if } a = b. \end{cases}$$

Definition 5. Let $n \in \mathbb{N} \setminus \{0\}$ and $x, y \in \Sigma^n$. The *similarity density* of x and y is

$$\mathrm{SD}(x,y) := \frac{1}{n} \sum_{i=0}^{n-1} \delta(x[i], y[i]).$$

Thus, two finite words of the same length have similarity density 1 if and only if they are equal.

Definition 6. Let $\mathbf{x}, \mathbf{y} \in \Sigma^\omega$. The *lower* and *upper similarity densities* of $\mathbf{x}$ and $\mathbf{y}$ are, respectively,

$$\mathrm{LSD}(\mathbf{x},\mathbf{y}) := \liminf_{n\to\infty} \mathrm{SD}(\mathbf{x}[0..n-1], \mathbf{y}[0..n-1]),$$
$$\mathrm{USD}(\mathbf{x},\mathbf{y}) := \limsup_{n\to\infty} \mathrm{SD}(\mathbf{x}[0..n-1], \mathbf{y}[0..n-1]).$$

Remark 7. Our notion of similarity density is not a new idea. (Similar ideas can be found, e.g., in [8, 6].) It is inspired by the well-studied number-theoretic notion of *asymptotic* or *natural density* of subsets of natural numbers. The *lower* and *upper asymptotic densities* of $A \subseteq \mathbb{N}$ are, respectively,

$$\underline{d}(A) := \liminf_{n\to\infty} \frac{1}{n} \left| A \cap \{0, \ldots, n-1\} \right|,$$
$$\overline{d}(A) := \limsup_{n\to\infty} \frac{1}{n} \left| A \cap \{0, \ldots, n-1\} \right|.$$

Similarity density generalizes asymptotic density in the following way. For $A \subseteq \mathbb{N}$, let $\chi_A \in \Sigma_2^\omega$ denote the characteristic sequence of A (i.e., $\chi_A[n] = \mathtt{1}$ iff $n \in A$). Then

$$\underline{d}(A) = \mathrm{LSD}(\chi_A, \mathtt{1}^\omega),$$
$$\overline{d}(A) = \mathrm{USD}(\chi_A, \mathtt{1}^\omega).$$

Mahler's result can now be restated as follows.

Theorem 8. *For all* $k \in \mathbb{N} \setminus \{0\}$, $\frac{1}{3} \le \mathrm{LSD}(\mathbf{t}, \mathbf{t}[k..\infty]) = \mathrm{USD}(\mathbf{t}, \mathbf{t}[k..\infty]) \le \frac{2}{3}$.

Proof. Note that for all $i, k \in \mathbb{N}$, $(-1)^{\mathbf{t}[i]+\mathbf{t}[i+k]} = 2\delta(\mathbf{t}[i], \mathbf{t}[i+k]) - 1$. Hence, by Definition 6, Theorem 2, and Corollary 3, we obtain

$$\mathrm{LSD}(\mathbf{t}, \mathbf{t}[k..\infty]) = \mathrm{USD}(\mathbf{t}, \mathbf{t}[k..\infty]) = \frac{1}{2}(\sigma(k)+1) \in \frac{1}{2}\left(\left[-\frac{1}{3}, \frac{1}{3}\right] + 1\right) = \left[\frac{1}{3}, \frac{2}{3}\right]. \qquad \blacksquare$$

Remark 9. There exist overlap-free infinite binary words $\mathbf{w}$ with $\mathrm{LSD}(\mathbf{t},\mathbf{w}) < \mathrm{USD}(\mathbf{t},\mathbf{w})$. One example is the word $\mathbf{h} = \mathtt{00100110100101100110100110010110}\cdots$ whose n^{th} position is the number of $\mathtt{0}$s (modulo 2) in the binary representation of n. (Note that $\mathbf{h}[0] = \mathtt{0}$ as we take the binary representation of 0 to be ε.) We prove in Proposition 17 that $\mathrm{LSD}(\mathbf{t},\mathbf{h}) = \frac{1}{3}$ while $\mathrm{USD}(\mathbf{t},\mathbf{h}) = \frac{2}{3}$. See Figure 1, where this similarity density is graphed as a function of the length of the prefix.

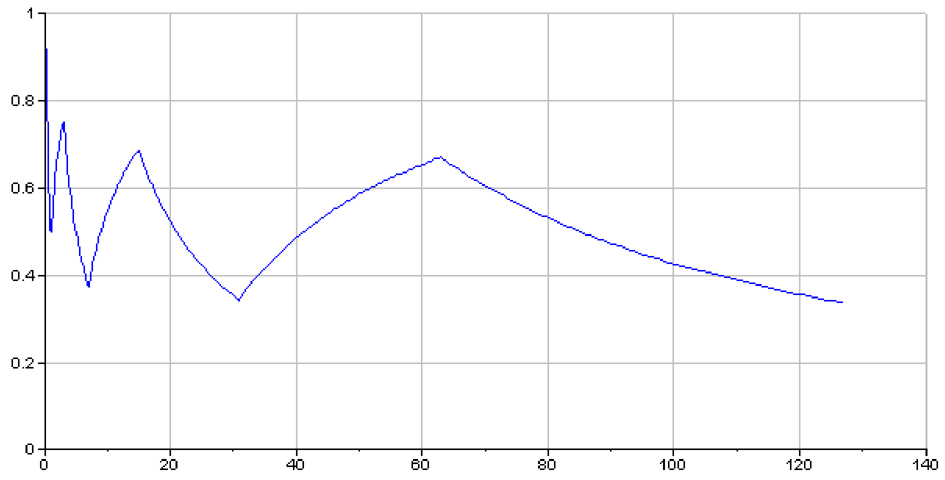


Figure 1: Similarity density of prefixes of **t** and **h**

Our main result (Theorem 18) is that the lower and upper similarity densities of **t** with *any* overlap-free infinite binary word other than **t** and $\overline{\mathbf{t}}$ are bounded below and above as in Theorem 8, but with the constants $\frac{1}{4}$ and $\frac{3}{4}$ instead of $\frac{1}{3}$ and $\frac{2}{3}$ respectively. However, computational evidence suggests that the tighest bounds are indeed $\frac{1}{3}$ and $\frac{2}{3}$, which, if true, would fully generalize Theorem 8 from nontrivial shifts of **t** to all overlap-free infinite binary words (other than **t** and $\overline{\mathbf{t}}$).

The following are basic properties of similarity density that we will use later. Their statements are all intuitive and their proofs are just basic exercises in algebra. Observation 10 states that similarity density can be computed using weighted averages. Observation 11 and Corollary 12 explain how complementation affects similarity density. Observation 13 states that the similarity densities of infinite words depends only on their tails, so we can ignore arbitrarily long prefixes. Observation 14 states that the similarity densities of infinite words can be obtained by considering similarity densities of prefixes where the length of the prefix grows by any constant instead of just by one in each iteration.

Observation 10. *Let $n,m \in \mathbb{N} \setminus \{0\}$, $u,v \in \Sigma^n$, and $x,y \in \Sigma^m$. Then*

$$\mathrm{SD}(ux,vy) = \frac{n}{n+m}\,\mathrm{SD}(u,v) + \frac{m}{n+m}\,\mathrm{SD}(x,y).$$

Proof.

$$\begin{aligned}
\mathrm{SD}(ux,vy) &= \frac{1}{n+m}\sum_{i=0}^{n+m-1} \delta((ux)[i],(vy)[i]) \\
&= \frac{1}{n+m}\left(\sum_{i=0}^{n-1} \delta(u[i],v[i]) + \sum_{i=0}^{m-1} \delta(x[i],y[i])\right) \\
&= \frac{n}{n+m}\cdot\frac{1}{n}\sum_{i=0}^{n-1} \delta(u[i],v[i]) + \frac{m}{n+m}\cdot\frac{1}{m}\sum_{i=0}^{m-1} \delta(u[i],v[i]) \\
&= \frac{n}{n+m}\,\mathrm{SD}(u,v) + \frac{m}{n+m}\,\mathrm{SD}(x,y).
\end{aligned}$$

■

Observation 11. *For all $n \in \mathbb{N} \setminus \{0\}$ and $x, y \in \Sigma_2^n$,*

(i) $\mathrm{SD}(\overline{x}, y) = 1 - \mathrm{SD}(x, y)$.

(ii) $\mathrm{SD}(\overline{x}, \overline{y}) = \mathrm{SD}(x, y)$.

Proof.

(i) $\mathrm{SD}(\overline{x}, y) = \frac{1}{n}\sum_{i=0}^{n-1}\delta(\overline{x}[i], y[i]) = \frac{1}{n}\sum_{i=0}^{n-1}(1 - \delta(x[i], y[i])) = 1 - \mathrm{SD}(x, y)$.

(ii) By (i) and symmetry of SD, we have $\mathrm{SD}(\overline{x}, \overline{y}) = 1 - \mathrm{SD}(x, \overline{y}) = 1 - (1 - \mathrm{SD}(x, y)) = \mathrm{SD}(x, y)$. ■

Corollary 12. *For all $\mathbf{x}, \mathbf{y} \in \Sigma_2^\omega$,*

(i) $\mathrm{LSD}(\overline{\mathbf{x}}, \mathbf{y}) = 1 - \mathrm{USD}(\mathbf{x}, \mathbf{y})$ *and* $\mathrm{USD}(\overline{\mathbf{x}}, \mathbf{y}) = 1 - \mathrm{LSD}(\mathbf{x}, \mathbf{y})$.

(ii) $\mathrm{LSD}(\overline{\mathbf{x}}, \overline{\mathbf{y}}) = \mathrm{LSD}(\mathbf{x}, \mathbf{y})$ *and* $\mathrm{USD}(\overline{\mathbf{x}}, \overline{\mathbf{y}}) = \mathrm{USD}(\mathbf{x}, \mathbf{y})$.

Proof. Immediate by Definition 6, Observation 11, and basic properties of limits. ■

Observation 13. *Let $l \in \mathbb{N}$, $u, v \in \Sigma^l$ and $\mathbf{x}, \mathbf{y} \in \Sigma^\omega$. Then* $\mathrm{LSD}(u\mathbf{x}, v\mathbf{y}) = \mathrm{LSD}(\mathbf{x}, \mathbf{y})$ *and* $\mathrm{USD}(u\mathbf{x}, v\mathbf{y}) = \mathrm{USD}(\mathbf{x}, \mathbf{y})$.

Proof. If $l = 0$, then the proof is trivial. If $l > 0$, then we have

$$\begin{aligned}
\mathrm{LSD}(u\mathbf{x}, v\mathbf{y}) &= \liminf_{n\to\infty} \frac{1}{n}\sum_{i=0}^{n-1}\delta((u\mathbf{x})[i], (v\mathbf{y})[i]) \\
&= \liminf_{n\to\infty} \frac{1}{n+l}\sum_{i=0}^{n+l-1}\delta((u\mathbf{x})[i], (v\mathbf{y})[i]) \\
&= \liminf_{n\to\infty} \left(\underbrace{\frac{1}{n+l}\sum_{i=0}^{l-1}\delta(u[i], v[i])}_{\in[0,\frac{l}{n+l}]\xrightarrow{n\to\infty}0} + \frac{1}{n+l}\sum_{i=0}^{n-1}\delta(\mathbf{x}[i], \mathbf{y}[i]) \right) \\
&= \liminf_{n\to\infty} \left(0 + \left(\frac{1}{n} - \frac{l}{n(n+l)}\right)\sum_{i=0}^{n-1}\delta(\mathbf{x}[i], \mathbf{y}[i]) \right) \\
&= \liminf_{n\to\infty} \left(\frac{1}{n}\sum_{i=0}^{n-1}\delta(\mathbf{x}[i], \mathbf{y}[i]) - \underbrace{\frac{l}{n(n+l)}\sum_{i=0}^{n-1}(1 - \delta(\mathbf{x}[i], \mathbf{y}[i]))}_{\in[0,\frac{l}{n+l}]\xrightarrow{n\to\infty}0} \right) \\
&= \liminf_{n\to\infty} \left(\frac{1}{n}\sum_{i=0}^{n-1}(1 - \delta(\mathbf{x}[i], \mathbf{y}[i])) - 0 \right) \\
&= \mathrm{LSD}(\mathbf{x}, \mathbf{y}).
\end{aligned}$$

The proof is exactly the same for USD with lim inf replaced by lim sup. ■

Observation 14. *Let $M \in \mathbb{N} \setminus \{0\}$. Then*

$$\mathrm{LSD}(\mathbf{x}, \mathbf{y}) = \liminf_{n\to\infty} \mathrm{SD}(\mathbf{x}[0..Mn-1], \mathbf{y}[0..Mn-1]),$$
$$\mathrm{USD}(\mathbf{x}, \mathbf{y}) = \limsup_{n\to\infty} \mathrm{SD}(\mathbf{x}[0..Mn-1], \mathbf{y}[0..Mn-1]).$$

Proof. For any $n \in \mathbb{N} \setminus \{0\}$ and $k \in \{Mn, Mn+1, \ldots, M(n+1)-2\}$, by Observation 10, we have

$$\begin{aligned}
\mathrm{SD}(\mathbf{x}[0..k], \mathbf{y}[0..k]) &= \frac{Mn}{k+1}\mathrm{SD}(\mathbf{x}[0..Mn-1], \mathbf{y}[0..Mn-1]) \\
&\quad + \frac{k-Mn+1}{k+1}\mathrm{SD}(\mathbf{x}[Mn..k], \mathbf{y}[Mn..k]) \\
&\in \left[\frac{Mn}{M(n+1)-1}, \frac{Mn}{Mn+1}\right]\mathrm{SD}(\mathbf{x}[0..Mn-1], \mathbf{y}[0..Mn-1]) \\
&\quad + \left[\frac{1}{M(n+1)-1}, \frac{M-1}{Mn+1}\right]\mathrm{SD}(\mathbf{x}[Mn..k], \mathbf{y}[Mn..k]),
\end{aligned}$$

so since $\lim_{n\to\infty}[\frac{Mn}{M(n+1)-1}, \frac{Mn}{Mn+1}] = [1,1] = \{1\}$ and $\lim_{n\to\infty}[\frac{1}{M(n+1)-1}, \frac{M-1}{Mn+1}] = [0,0] = \{0\}$, all of the intermediate values $\mathrm{SD}(\mathbf{x}[0..k], \mathbf{y}[0..k])$ for $k \in \{Mn, Mn+1, .., M(n+1)-2\}$ get arbitrarily close to $\mathrm{SD}(\mathbf{x}[0..Mn-1], \mathbf{y}[0..Mn-1])$ as $n \to \infty$. Hence,

$$\begin{aligned}
\liminf_{n\to\infty} \mathrm{SD}(\mathbf{x}[0..n-1], \mathbf{y}[0..n-1]) &= \liminf_{n\to\infty} \mathrm{SD}(\mathbf{x}[0..Mn-1], \mathbf{y}[0..Mn-1]), \\
\limsup_{n\to\infty} \mathrm{SD}(\mathbf{x}[0..n-1], \mathbf{y}[0..n-1]) &= \limsup_{n\to\infty} \mathrm{SD}(\mathbf{x}[0..Mn-1], \mathbf{y}[0..Mn-1]).
\end{aligned}$$ ■

4 Fife automaton for overlap-free infinite binary words

We recall the so-called "Fife automaton" for overlap-free infinite binary words from [10]. (Note that this automaton does not appear in the original paper of Fife [5].)

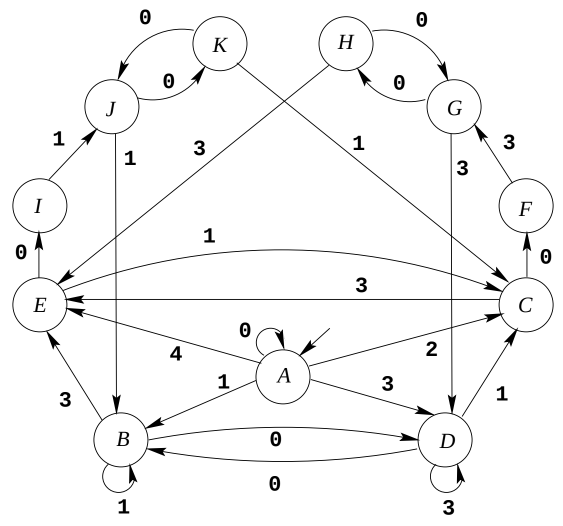


Figure 2: Automaton encoding all overlap-free infinite binary words

Here, infinite paths through the automaton encode all overlap-free infinite binary words, as follows.

Definition 15. First, each of the edge labels encodes a binary word, via $c : \Sigma_5 \to \Sigma_2^*$ defined by

$$\begin{aligned} c(\mathtt{0}) &:= \varepsilon, \\ c(\mathtt{1}) &:= \mathtt{0}, \\ c(\mathtt{2}) &:= \mathtt{00}, \\ c(\mathtt{3}) &:= \mathtt{1}, \\ c(\mathtt{4}) &:= \mathtt{11}. \end{aligned}$$

Then, the Fife-to-binary encoding $\mathrm{FBE} : (\Sigma_5^\omega \setminus \Sigma_5^* \mathtt{0}^\omega) \cup \left(\Sigma_5^* \mathtt{0}^\omega \times \Sigma_2\right) \to \Sigma_2^\omega$ is defined by

$$\begin{aligned} \mathrm{FBE}(\mathbf{x}) &:= \prod_{n=0}^{\infty} \mu^n(c(\mathbf{x}[n])) && \text{for } \mathbf{x} \in \Sigma_5^\omega \setminus \Sigma_5^* \mathtt{0}^\omega; \\ \mathrm{FBE}(\mathbf{x}, a) &:= \left(\prod_{n=0}^{\infty} \mu^n(c(\mathbf{x}[n])) \right) \mu^\omega(a) && \text{for } (\mathbf{x}, a) \in \Sigma_5^* \mathtt{0}^\omega \times \Sigma_2. \end{aligned}$$

Note that FBE is well-defined because c is only erasing for the letter $\mathtt{0}$ and μ is non-erasing, so for $\mathbf{x} \in \Sigma_5^\omega$, the concatenation $\prod_{n=0}^{\infty} \mu^n(c(\mathbf{x}[n]))$ is finite iff $\mathbf{x}$ ends in $\mathtt{0}^\omega$.

We now recall the basic property of the automaton from [10].

Theorem 16. *Let $\mathbf{w} \in \Sigma_2^\omega$. Then $\mathbf{w}$ is overlap-free iff there exists $\mathbf{x} \in \Sigma_5^\omega$ that encodes a valid path through the Fife automaton for overlap-free infinite binary words such that* $\mathrm{FBE}(\mathbf{x}) = \mathbf{w}$ *(if $\mathbf{x}$ does not end in $\mathtt{0}^\omega$) or* $\mathrm{FBE}(\mathbf{x}, a) = \mathbf{w}$ *(if $\mathbf{x}$ ends in $\mathtt{0}^\omega$) for some $a \in S$, where $S \subseteq \Sigma_2$ depends on the eventual cycle corresponding to the suffix $\mathtt{0}^\omega$ of the path encoded by $\mathbf{x}$: on state A and between states B and D ($S = \Sigma_2$), between states G and H ($S = \{\mathtt{1}\}$), or between states J and K ($S = \{\mathtt{0}\}$).*

Recall $\mathbf{h}$ as defined in Remark 9. Note that the definitions of $\mathbf{h}$ and $\mathbf{t}$ are very similar. This is related to the special path that encodes $\mathbf{h}$ in the Fife automaton for overlap-free infinite binary words [10]: $\mathbf{h} = \mathrm{FBE}(\mathtt{2}(\mathtt{31})^\omega)$. We will see later in our proof of our main result why this path is special. For now, we can use this path to compute the following result.

Proposition 17. $\mathrm{LSD}(\mathbf{h}, \mathbf{t}) = \mathrm{LSD}(\overline{\mathbf{h}}, \mathbf{t}) = \frac{1}{3}$ *and* $\mathrm{USD}(\mathbf{h}, \mathbf{t}) = \mathrm{USD}(\overline{\mathbf{h}}, \mathbf{t}) = \frac{2}{3}$.

Proof. Note that

$$\begin{aligned} \mathbf{h} = \mathrm{FBE}(\mathtt{2}(\mathtt{31})^\omega) &= \mu^0(p(\mathtt{2})) \prod_{n=0}^{\infty} \left(\mu^{2n+1}(p(\mathtt{3})) \mu^{2n+2}(p(\mathtt{1})) \right) \\ &= \mu^0(\mathtt{00}) \prod_{n=0}^{\infty} \left(\mu^{2n+1}(\mathtt{1}) \mu^{2n+2}(\mathtt{0}) \right) = \mathtt{0} t_0 \prod_{n=0}^{\infty} \left(\overline{t_{2n+1}} t_{2n+2} \right) = \mathtt{0} \prod_{n=0}^{\infty} \left(t_{2n} \overline{t_{2n+1}} \right), \end{aligned}$$

and since for each $n \in \mathbb{N}$, we have $|t_n| = 2^n$ and $1 + \sum_{i=0}^{n} 2^i = 2^{n+1}$, it follows that

$$\mathbf{h}[2^n..2^{n+1}-1] = \begin{cases} t_n, & \text{if } n \equiv 0 \pmod 2; \\ \overline{t_n}, & \text{if } n \equiv 1 \pmod 2. \end{cases}$$

Note that for each $n \in \mathbb{N}$, we have $\mathbf{t}[2^n..2^{n+1}-1] = t_{n+1}[2^n..2^{n+1}-1] = \overline{t_n}$. Hence, for all $n \in \mathbb{N}$,

$$\mathbf{h}[2^n..2^{n+1}-1] = \begin{cases} \overline{\mathbf{t}}[2^n..2^{n+1}-1], & \text{if } n \equiv 0 \pmod 2; \\ \mathbf{t}[2^n..2^{n+1}-1], & \text{if } n \equiv 1 \pmod 2, \end{cases}$$

whence

$$\mathrm{SD}(\mathbf{h}[2^n..2^{n+1}-1],\mathbf{t}[2^n..2^{n+1}-1]) = \begin{cases} 0, & \text{if } n \equiv 0 \pmod 2; \\ 1, & \text{if } n \equiv 1 \pmod 2. \end{cases}$$

If we consider two of these blocks at a time, we obtain, by Observation 10, that for all $n \in \mathbb{N}$,

$$\begin{aligned} \mathrm{SD}(\mathbf{h}[2^n..2^{n+2}-1],\mathbf{t}[2^n..2^{n+2}-1]) &= \frac{2^n}{2^n+2^{n+1}}\mathrm{SD}(\mathbf{h}[2^n..2^{n+1}-1],\mathbf{t}[2^n..2^{n+1}-1]) \\ &\quad + \frac{2^{n+1}}{2^n+2^{n+1}}\mathrm{SD}(\mathbf{h}[2^{n+1}..2^{n+2}-1],\mathbf{t}[2^{n+1}..2^{n+2}-1]) \\ &= \begin{cases} \frac{2}{3}, & \text{if } n \equiv 0 \pmod 2; \\ \frac{1}{3}, & \text{if } n \equiv 1 \pmod 2. \end{cases} \end{aligned}$$

Iterating Observation 10 finitely many times, we obtain that for all $n \in \mathbb{N}$,

$$\mathrm{SD}(\mathbf{h}[1..2^{2n}-1],\mathbf{t}[1..2^{2n}-1]) = \frac{2}{3},$$
$$\mathrm{SD}(\mathbf{h}[2..2^{2n+1}-1],\mathbf{t}[2..2^{2n+1}-1]) = \frac{1}{3}.$$

Furthermore, applying Observation 10 one letter at a time, we see that for $k \in [2^{2n}-1, 2^{2n+1}-1]$, $\mathrm{SD}(\mathbf{h}[1..k],\mathbf{t}[1..k])$ monotonically decreases (from $\frac{2}{3}$), and for $k \in [2^{2n+1}-1, 2^{2n+2}-1]$, $\mathrm{SD}(\mathbf{h}[1..k],\mathbf{t}[1..k])$ monotonically increases (back to $\frac{2}{3}$). Thus,

$$\mathrm{USD}(\mathbf{h}[1..\infty],\mathbf{t}[1..\infty]) = \limsup_{n\to\infty} \mathrm{SD}(\mathbf{h}[1..n],\mathbf{t}[1..n]) = \frac{2}{3}.$$

Similarly, for $k \in [2^{2n+1}-1, 2^{2n+2}-1]$, $\mathrm{SD}(\mathbf{h}[2..k],\mathbf{t}[2..k])$ monotonically increases (from $\frac{1}{3}$), and for $k \in [2^{2n+2}-1, 2^{2n+3}-1]$, $\mathrm{SD}(\mathbf{h}[2..k],\mathbf{t}[2..k])$ monotonically decreases (back to $\frac{1}{3}$), so

$$\mathrm{LSD}(\mathbf{h}[2..\infty],\mathbf{t}[2..\infty]) = \liminf_{n\to\infty} \mathrm{SD}(\mathbf{h}[2..n+1],\mathbf{t}[2..n+1]) = \frac{1}{3}.$$

Finally, by Observation 13, we conclude that $\mathrm{LSD}(\mathbf{h},\mathbf{t}) = \mathrm{LSD}(\mathbf{h}[2..\infty],\mathbf{t}[2..\infty]) = \frac{1}{3}$ and $\mathrm{USD}(\mathbf{h},\mathbf{t}) = \mathrm{USD}(\mathbf{h}[1..\infty],\mathbf{t}[1..\infty]) = \frac{2}{3}$, whence by Corollary 12(i), we obtain $\mathrm{LSD}(\overline{\mathbf{h}},\mathbf{t}) = 1 - \mathrm{USD}(\mathbf{h},\mathbf{t}) = 1 - \frac{2}{3} = \frac{1}{3}$ and $\mathrm{USD}(\overline{\mathbf{h}},\mathbf{t}) = 1 - \mathrm{LSD}(\mathbf{h},\mathbf{t}) = 1 - \frac{1}{3} = \frac{2}{3}$. ■

5 Main result

We now state and prove our main result.

Theorem 18. *For all overlap-free* $\mathbf{w} \in \Sigma_2^\omega \setminus \{\mathbf{t}, \overline{\mathbf{t}}\}$, $\frac{1}{4} \le \mathrm{LSD}(\mathbf{w},\mathbf{t}) \le \mathrm{USD}(\mathbf{w},\mathbf{t}) \le \frac{3}{4}$.

Our approach to proving Theorem 18 is to consider each overlap-free infinite binary word in terms of the path through the Fife automaton that encodes it. We divide the paths into four cases.

(1) ends in $\mathtt{0}^\omega$.

(2) does not end in $\mathtt{0}^\omega$, begins with $\mathtt{0}^n\mathtt{2}$ or $\mathtt{0}^n\mathtt{4}$ for some $n \in \mathbb{N}$, and contains exactly n $\mathtt{0}$s.

(3) does not end in $\mathtt{0}^\omega$, begins with $\mathtt{0}^n\mathtt{2}$ or $\mathtt{0}^n\mathtt{4}$ for some $n \in \mathbb{N}$, and contains more than n $\mathtt{0}$s.

(4) does not end in $\mathtt{0}^\omega$ and begins with $\mathtt{0}^n\mathtt{1}$ or $\mathtt{0}^n\mathtt{3}$ for some $n \in \mathbb{N}$.

Upon closer examination of the Fife automaton, case (2) can be subdivided into two cases: $\mathtt{0}^n\mathtt{2}(\mathtt{31})^\omega$ and their complements under FBE, $\mathtt{0}^n\mathtt{4}(\mathtt{13})^\omega$. It turns out that we can bootstrap Proposition 17 to obtain the same bounds for both of these cases. Case (1) follows from Mahler's theorem 8, but it will also follow from our own generalized version of it (albeit with weaker bounds). For cases (3) and (4), we observe that the infinite binary word corresponding to the path eventually "lags behind" the prefixes t_n of $\mathbf{t}$ in the sense that each successive n^{th} symbol in the path can only generate positions prior to 2^n, whence we can use a technical lemma that bounds the similarity density of t_n with nontrivial subwords of t_{n+1} to complete the proof.

Proposition 19. *For all $n \in \mathbb{N}$ we have* $\mathrm{LSD}(\mathrm{FBE}(\mathtt{0}^n\mathtt{2}(\mathtt{31})^\omega), \mathbf{t}) = \frac{1}{3}$ *and* $\mathrm{USD}(\mathrm{FBE}(\mathtt{0}^n\mathtt{2}(\mathtt{31})^\omega), \mathbf{t}) = \frac{2}{3}$.

Proof. Note that

$$\begin{aligned}
\mathrm{FBE}(\mathtt{0}^n\mathtt{2}(\mathtt{31})^\omega) &= \prod_{k=0}^{n-1}\Big(\mu^k(p(\mathtt{0}))\Big)\,\mu^n(p(\mathtt{2}))\prod_{k=0}^{\infty}\Big(\mu^{n+2k+1}(p(\mathtt{3}))\mu^{n+2k+2}(p(\mathtt{1}))\Big)\\
&= \prod_{k=0}^{n-1}\Big(\mu^k(\varepsilon)\Big)\,\mu^n(\mathtt{00})\prod_{k=0}^{\infty}\Big(\mu^{n+2k+1}(\mathtt{1})\mu^{n+2k+2}(\mathtt{0})\Big)\\
&= t_n t_n \prod_{k=0}^{\infty}(\overline{t_{n+2k+1}}\,t_{n+2k+2})\\
&= t_n \prod_{k=0}^{\infty}(t_{n+2k}\overline{t_{n+2k+1}})\,.
\end{aligned}$$

From the proof of Proposition 17, we see that

$$\mathrm{FBE}(\mathtt{0}^n\mathtt{2}(\mathtt{31})^\omega)[2^n..\infty] = \begin{cases}\mathbf{h}[2^n..\infty], & \text{if } n \equiv 0 \pmod 2;\\ \overline{\mathbf{h}}[2^n..\infty], & \text{if } n \equiv 1 \pmod 2.\end{cases}$$

Hence, by Observation 13 and Proposition 17, we have

$$\begin{aligned}
(\mathrm{LSD},\mathrm{USD})(\mathrm{FBE}(\mathtt{0}^n\mathtt{2}(\mathtt{31})^\omega),\mathbf{t}) &= (\mathrm{LSD},\mathrm{USD})(\mathrm{FBE}(\mathtt{0}^n\mathtt{2}(\mathtt{31})^\omega)[2^n..\infty],\mathbf{t}[2^n..\infty])\\
&= \begin{cases}(\mathrm{LSD},\mathrm{USD})(\mathbf{h}[2^n..\infty],\mathbf{t}[2^n..\infty]), & \text{if } n \equiv 0 \pmod 2;\\ (\mathrm{LSD},\mathrm{USD})(\overline{\mathbf{h}}[2^n..\infty],\mathbf{t}[2^n..\infty]), & \text{if } n \equiv 1 \pmod 2,\end{cases}\\
&= \begin{cases}(\mathrm{LSD},\mathrm{USD})(\mathbf{h},\mathbf{t}), & \text{if } n \equiv 0 \pmod 2;\\ (\mathrm{LSD},\mathrm{USD})(\overline{\mathbf{h}},\mathbf{t}), & \text{if } n \equiv 1 \pmod 2,\end{cases}\\
&= \left(\frac{1}{3},\frac{2}{3}\right).
\end{aligned}$$

■

Lemma 20. *For all $n \in \mathbb{N}$ and $i \in [1, 2^n - 1]$,*

$$(a) \qquad \mathrm{SD}(t_n, t_{n+1}[i\,.\,.\,2^n+i-1]) \in \begin{cases} \{\frac{1}{2}\}, & \textit{if } i = 2^{n-1}; \\ [\frac{1}{4}, \frac{3}{4}], & \textit{otherwise}. \end{cases}$$

$$\overline{(}a) \qquad \mathrm{SD}(\overline{t_n}, t_{n+1}[i\,.\,.\,2^n+i-1]) \in \begin{cases} \{\frac{1}{2}\}, & \textit{if } i = 2^{n-1}; \\ [\frac{1}{4}, \frac{3}{4}], & \textit{otherwise}. \end{cases}$$

$$(a\overline{)} \qquad \mathrm{SD}(t_n, \overline{t_{n+1}}[i\,.\,.\,2^n+i-1]) \in \begin{cases} \{\frac{1}{2}\}, & \textit{if } i = 2^{n-1}; \\ [\frac{1}{4}, \frac{3}{4}], & \textit{otherwise}. \end{cases}$$

$$\overline{(}a\overline{)} \qquad \mathrm{SD}(\overline{t_n}, \overline{t_{n+1}}[i\,.\,.\,2^n+i-1]) \in \begin{cases} \{\frac{1}{2}\}, & \textit{if } i = 2^{n-1}; \\ [\frac{1}{4}, \frac{3}{4}], & \textit{otherwise}. \end{cases}$$

$$(b) \qquad \mathrm{SD}(t_n, t_n^2[i\,.\,.\,2^n+i-1]) \in \begin{cases} \{0\}, & \textit{if } i = 2^{n-1}; \\ [\frac{1}{4}, \frac{3}{4}], & \textit{otherwise}. \end{cases}$$

$$\overline{(}b) \qquad \mathrm{SD}(\overline{t_n}, t_n^2[i\,.\,.\,2^n+i-1]) \in \begin{cases} \{1\}, & \textit{if } i = 2^{n-1}; \\ [\frac{1}{4}, \frac{3}{4}], & \textit{otherwise}. \end{cases}$$

$$(b\overline{)} \qquad \mathrm{SD}(t_n, \overline{t_n}^2[i\,.\,.\,2^n+i-1]) \in \begin{cases} \{1\}, & \textit{if } i = 2^{n-1}; \\ [\frac{1}{4}, \frac{3}{4}], & \textit{otherwise}. \end{cases}$$

$$\overline{(}b\overline{)} \qquad \mathrm{SD}(\overline{t_n}, \overline{t_n}^2[i\,.\,.\,2^n+i-1]) \in \begin{cases} \{0\}, & \textit{if } i = 2^{n-1}; \\ [\frac{1}{4}, \frac{3}{4}], & \textit{otherwise}. \end{cases}$$

Proof. By induction on n.

- For $n = 0$, all eight cases are vacuously true due to $i \in \emptyset$.
- Suppose all eight cases hold for some $n \in \mathbb{N}$. For $i \in [1, 2^{n+1} - 1]$, using Observation 10 followed by the induction hypothesis, we calculate

$$\begin{aligned}
&\mathrm{SD}(t_{n+1}, t_{n+2}[i\,.\,.\,2^{n+1}+i-1]) \\
&= \mathrm{SD}(t_n\overline{t_n}, (t_n\overline{t_n}\overline{t_n}t_n)[i\,.\,.\,2^{n+1}+i-1]) \\
&= \begin{cases} \mathrm{SD}(t_n\overline{t_n}, (t_n\overline{t_n}\overline{t_n})[i\,.\,.\,2^{n+1}+i-1]), & \text{if } i \in [1, 2^n-1]; \\ \mathrm{SD}(t_n\overline{t_n}, \overline{t_n}\overline{t_n}), & \text{if } i = 2^n; \\ \mathrm{SD}(t_n\overline{t_n}, (\overline{t_n}\overline{t_n}t_n)[i-2^n\,.\,.\,2^n+i-1]), & \text{if } i \in [2^n+1, 2^{n+1}-1], \end{cases} \\
&= \begin{cases} \frac{2^n}{2^{n+1}}\,\mathrm{SD}(t_n, (t_n\overline{t_n})[i\,.\,.\,2^n+i-1]) + \frac{2^n}{2^{n+1}}\,\mathrm{SD}(\overline{t_n}, (\overline{t_n}\overline{t_n})[i\,.\,.\,2^n+i-1]), & \text{if } i \in [1, 2^n-1]; \\ \frac{2^n}{2^{n+1}}\,\mathrm{SD}(t_n, \overline{t_n}) + \frac{2^n}{2^{n+1}}\,\mathrm{SD}(\overline{t_n}, \overline{t_n}), & \text{if } i = 2^n; \\ \frac{2^n}{2^{n+1}}\,\mathrm{SD}(t_n, (\overline{t_n}\overline{t_n})[i-2^n\,.\,.\,i-1]) + \frac{2^n}{2^{n+1}}\,\mathrm{SD}(\overline{t_n}, (\overline{t_n}t_n)[i-2^n\,.\,.\,i-1]), & \text{if } i \in [2^n+1, 2^{n+1}-1], \end{cases} \\
&\in \begin{cases} \frac{1}{2}\{\frac{1}{2}\} + \frac{1}{2}\{0\}, & \text{if } i = 2^{n-1};\ (\text{by (a)}, \overline{(}\text{b}\overline{)}) \\ \frac{1}{2}[\frac{1}{4}, \frac{3}{4}] + \frac{1}{2}[\frac{1}{4}, \frac{3}{4}], & \text{if } i \in [1, 2^n-1] \setminus \{2^{n-1}\};\ (\text{by (a)}, \overline{(}\text{b}\overline{)}) \\ \frac{1}{2}\{0\} + \frac{1}{2}\{1\}, & \text{if } i = 2^n; \\ \frac{1}{2}\{1\} + \frac{1}{2}\{\frac{1}{2}\}, & \text{if } i = 2^n + 2^{n-1};\ (\text{by (b}\overline{)}, \overline{(}\text{a)}) \\ \frac{1}{2}[\frac{1}{4}, \frac{3}{4}] + \frac{1}{2}[\frac{1}{4}, \frac{3}{4}], & \text{if } i \in [2^n+1, 2^{n+1}-1] \setminus \{2^n + 2^{n-1}\},\ (\text{by (b}\overline{)}, \overline{(}\text{a)}) \end{cases}
\end{aligned}$$

$$= \begin{cases} \{\frac{1}{4}\}, & \text{if } i = 2^{n-1}; \\ [\frac{1}{4}, \frac{3}{4}], & \text{if } i \in [1, 2^n - 1] \setminus \{2^{n-1}\}; \\ \{\frac{1}{2}\}, & \text{if } i = 2^n; \\ \{\frac{3}{4}\}, & \text{if } i = 2^n + 2^{n-1}; \\ [\frac{1}{4}, \frac{3}{4}], & \text{if } i \in [2^n + 1, 2^{n+1} - 1] \setminus \{2^n + 2^{n-1}\}, \end{cases}$$

$$\begin{aligned} &\mathrm{SD}(t_{n+1}, t_{n+1}^2[i..2^{n+1} + i - 1]) \\ &= \mathrm{SD}(t_n\overline{t_n}, (t_n\overline{t_n}\,\overline{t_n}t_n)[i..2^{n+1} + i - 1]) \\ &= \begin{cases} \mathrm{SD}(t_n\overline{t_n}, (t_n\overline{t_n}\,\overline{t_n})[i..2^{n+1} + i - 1]), & \text{if } i \in [1, 2^n - 1]; \\ \mathrm{SD}(t_n\overline{t_n}, \overline{t_n}t_n), & \text{if } i = 2^n; \\ \mathrm{SD}(t_n\overline{t_n}, (\overline{t_n}t_n\overline{t_n})[i - 2^n..2^n + i - 1]), & \text{if } i \in [2^n + 1, 2^{n+1} - 1], \end{cases} \\ &= \begin{cases} \frac{2^n}{2^{n+1}}\,\mathrm{SD}(t_n, (t_n\overline{t_n})[i..2^n + i - 1]) + \frac{2^n}{2^{n+1}}\,\mathrm{SD}(\overline{t_n}, (\overline{t_n}t_n)[i..2^n + i - 1]), & \text{if } i \in [1, 2^n - 1]; \\ \frac{2^n}{2^{n+1}}\,\mathrm{SD}(t_n, \overline{t_n}) + \frac{2^n}{2^{n+1}}\,\mathrm{SD}(\overline{t_n}, t_n), & \text{if } i = 2^n; \\ \frac{2^n}{2^{n+1}}\,\mathrm{SD}(t_n, (\overline{t_n}t_n)[i - 2^n..i - 1]) + \frac{2^n}{2^{n+1}}\,\mathrm{SD}(\overline{t_n}, (t_n\overline{t_n})[i - 2^n..i - 1]), & \text{if } i \in [2^n + 1, 2^{n+1} - 1], \end{cases} \\ &\in \begin{cases} \frac{1}{2}\{\frac{1}{2}\} + \frac{1}{2}\{\frac{1}{2}\}, & \text{if } i = 2^{n-1};\ (\text{by (a)}, \overline{\text{(a)}}) \\ \frac{1}{2}[\frac{1}{4}, \frac{3}{4}] + \frac{1}{2}[\frac{1}{4}, \frac{3}{4}], & \text{if } i \in [1, 2^n - 1] \setminus \{2^{n-1}\};\ (\text{by (a)}, \overline{\text{(a)}}) \\ \frac{1}{2}\{0\} + \frac{1}{2}\{0\}, & \text{if } i = 2^n; \\ \frac{1}{2}\{\frac{1}{2}\} + \frac{1}{2}\{\frac{1}{2}\}, & \text{if } i = 2^n + 2^{n-1};\ (\text{by } \overline{\text{(a)}}, \text{(a)}) \\ \frac{1}{2}[\frac{1}{4}, \frac{3}{4}] + \frac{1}{2}[\frac{1}{4}, \frac{3}{4}], & \text{if } i \in [2^n + 1, 2^{n+1} - 1] \setminus \{2^n + 2^{n-1}\},\ (\text{by } \overline{\text{(a)}}, \text{(a)}) \end{cases} \\ &= \begin{cases} \{\frac{1}{2}\}, & \text{if } i = 2^{n-1}; \\ [\frac{1}{4}, \frac{3}{4}], & \text{if } i \in [1, 2^n - 1] \setminus \{2^{n-1}\}; \\ 0, & \text{if } i = 2^n; \\ \{\frac{1}{2}\}, & \text{if } i = 2^n + 2^{n-1}; \\ [\frac{1}{4}, \frac{3}{4}], & \text{if } i \in [2^n + 1, 2^{n+1} - 1] \setminus \{2^n + 2^{n-1}\}, \end{cases} \end{aligned}$$

hence proving (a) and (b) also hold for $n+1$. By Observation 11, the remaining six cases also hold for $n+1$. ∎

Corollary 21. *For all* $n \in \mathbb{N}$, $i \in [0, 2^n - 1]$ *with* $\gcd(i, 2^n) \le 2^{n-2}$, *and* $x, y_0, y_1 \in \{t_n, \overline{t_n}\}$,

$$\mathrm{SD}(x, (y_0y_1)[i..i + 2^n - 1]) \in [\tfrac{1}{4}, \tfrac{3}{4}].$$

Proof. Follows immediately from Lemma 20. ∎

Corollary 22. *For all* $n, i \in \mathbb{N}$ *with* $\gcd(i, 2^n) \le 2^{n-2}$ *and* $\mathbf{x}, \mathbf{y} \in \{t_n, \overline{t_n}\}^\omega$,

$$\frac{1}{4} \le \mathrm{LSD}(\mathbf{x}, \mathbf{y}[i..\infty]) \le \mathrm{USD}(\mathbf{x}, \mathbf{y}[i..\infty]) \le \frac{3}{4}.$$

Proof. Note that for any $j \in \mathbb{N}$, $\gcd(i + j \cdot 2^n, 2^n) = \gcd(i, 2^n) \le 2^{n-2}$. Also for any $j \in \mathbb{N}$, since $\mathbf{x}, \mathbf{y} \in \{t_n, \overline{t_n}\}^\omega$ and $|t_n| = |\overline{t_n}| = 2^n$, we have $\mathbf{x}[2^n j..2^n(j+1) - 1] \in \{t_n, \overline{t_n}\}$ and $\mathbf{y}[i + 2^n j..i + 2^n(j+1) - 1] = (y_0y_1)[(i \bmod 2^n)..(i \bmod 2^n) + 2^n - 1]$ for some $y_0, y_1 \in \{t_n, \overline{t_n}\}$. Hence, for any $j \in \mathbb{N}$, by Corollary 21,

$$\mathrm{SD}(\mathbf{x}[2^n j..2^n(j+1) - 1], \mathbf{y}[i + 2^n j..i + 2^n(j+1) - 1]) \in \left[\frac{1}{4}, \frac{3}{4}\right],$$

whence by Observation 10,

$$\mathrm{SD}(\mathbf{x}[0\,.\,.\,2^n(j+1)-1], \mathbf{y}[i\,.\,.\,i+2^n(j+1)-1]) \in \left[\frac{1}{4}, \frac{3}{4}\right],$$

whence by Observation 14,

$$\begin{aligned}(\mathrm{LSD}, \mathrm{USD})(\mathbf{x}, \mathbf{y}[i\,.\,.\,\infty]) &= \left(\liminf_{j\to\infty}, \limsup_{j\to\infty}\right) \mathrm{SD}(\mathbf{x}[0\,.\,.\,2^n j-1], \mathbf{y}[i\,.\,.\,i+2^n j-1]) \\ &\in \left(\left[\frac{1}{4}, \frac{3}{4}\right], \left[\frac{1}{4}, \frac{3}{4}\right]\right). \qquad \blacksquare\end{aligned}$$

Corollary 23. *For all $i \in \mathbb{N} \setminus \{0\}$, $\frac{1}{4} \le \mathrm{LSD}(\mathbf{t}, \mathbf{t}[i\,.\,.\,\infty]) \le \mathrm{USD}(\mathbf{t}, \mathbf{t}[i\,.\,.\,\infty]) \le \frac{3}{4}$.*

Proof. Since $i > 0$, we have $4\max_{m\in\mathbb{N}} \gcd(i, 2^m) = 2^n$ for some $n \in \mathbb{N}$. Note that $\gcd(i, 2^n) = 2^{n-2}$. Also note that $\mathbf{t} = \mu^n(\mathbf{t}) \in \{t_n, \overline{t_n}\}^\omega$. Hence, by Corollary 22,

$$\frac{1}{4} \le \mathrm{LSD}(\mathbf{t}, \mathbf{t}[i\,.\,.\,\infty]) \le \mathrm{USD}(\mathbf{t}, \mathbf{t}[i\,.\,.\,\infty]) \le \frac{3}{4}. \qquad \blacksquare$$

We now have all the tools needed to prove Theorem 18.

Proof of Theorem 18. Let $\mathbf{w} \in \Sigma_2^\omega \setminus \{\mathbf{t}, \bar{\mathbf{t}}\}$. By Theorem 16, there exists $\mathbf{x} \in \Sigma_5^\omega$ that encodes a valid path through the Fife automaton for overlap-free infinite binary words such that $\mathrm{FBE}(\mathbf{x}) = \mathbf{w}$ or $\mathrm{FBE}(\mathbf{x}, a) = \mathbf{w}$ for some $a \in \Sigma_2$. From inspection of the Fife automaton for overlap-free infinite binary words, we see that $\mathbf{x}$ must fall into one of the following four cases.

(1) $\mathbf{x}$ ends in $\mathtt{0}^\omega$.

(2) $\mathbf{x}$ does not end in $\mathtt{0}^\omega$, begins with $\mathtt{0}^n\mathtt{2}$ or $\mathtt{0}^n\mathtt{4}$ for some $n \in \mathbb{N}$, and contains exactly n $\mathtt{0}$s.

(3) $\mathbf{x}$ does not end in $\mathtt{0}^\omega$, begins with $\mathtt{0}^n\mathtt{2}$ or $\mathtt{0}^n\mathtt{4}$ for some $n \in \mathbb{N}$, and contains more than n $\mathtt{0}$s.

(4) $\mathbf{x}$ does not end in $\mathtt{0}^\omega$ and begins with $\mathtt{0}^n\mathtt{1}$ or $\mathtt{0}^n\mathtt{3}$ for some $n \in \mathbb{N}$.

Case 1: $\mathbf{w}$ ends in either $\mathbf{t}$ or $\bar{\mathbf{t}}$, so since $\mathbf{w} \notin \{\mathbf{t}, \bar{\mathbf{t}}\}$, it follows that $\mathbf{w} \in \{z\mathbf{t}, z\bar{\mathbf{t}}\}$ for some $z \in \Sigma_2^+$. By Observation 13, we have

$$(\mathrm{LSD}, \mathrm{USD})(\mathbf{w}, \mathbf{t}) \in \{(\mathrm{LSD}, \mathrm{USD})(\mathbf{t}, \mathbf{t}[|z|\,.\,.\,\infty]), (\mathrm{LSD}, \mathrm{USD})(\bar{\mathbf{t}}, \mathbf{t}[|z|\,.\,.\,\infty])\},$$

whence by Corollary 23 and Corollary 12, we obtain $(\mathrm{LSD}, \mathrm{USD})(\mathbf{w}, \mathbf{t}) \in (\{[\frac{1}{4}, \frac{3}{4}], [1-\frac{3}{4}, 1-\frac{1}{4}]\}, \{[\frac{1}{4}, \frac{3}{4}], [1-\frac{3}{4}, 1-\frac{1}{4}]\}) = ([\frac{1}{4}, \frac{3}{4}], [\frac{1}{4}, \frac{3}{4}])$, as desired.

Case 2: From inspection of the Fife automaton for overlap-free infinite binary words, we see that $\mathbf{x} \in \{\mathtt{0}^n\{\mathtt{2}(\mathtt{31})^\omega, \mathtt{4}(\mathtt{13})^\omega\} : n \in \mathbb{N}\}$. Note that $\mathrm{FBE}(\mathtt{0}^n\mathtt{4}(\mathtt{13})^\omega) = \overline{\mathrm{FBE}(\mathtt{0}^n\mathtt{2}(\mathtt{31})^\omega)}$. Hence, by Proposition 19 and Corollary 12, we obtain $(\mathrm{LSD}, \mathrm{USD})(\mathbf{w}, \mathbf{t}) \in \{(\frac{1}{3}, \frac{2}{3}), (1-\frac{2}{3}, 1-\frac{1}{3})\} = \{(\frac{1}{3}, \frac{2}{3})\} \subset ([\frac{1}{4}, \frac{3}{4}], [\frac{1}{4}, \frac{3}{4}])$, as desired.

Case 3: From inspection of the Fife automaton for overlap-free infinite binary words, we see that $\mathbf{x} \in \{\mathtt{0}^n\{\mathtt{2}(\mathtt{31})^{\frac{m}{2}}, \mathtt{4}(\mathtt{13})^{\frac{m}{2}}\}\mathtt{0}\{\mathtt{1}, \mathtt{3}\}\mathbf{y} : n, m \in \mathbb{N}, \mathbf{y} \in \{\mathtt{0}, \mathtt{1}, \mathtt{3}\}^\omega\}$, whence

$$\begin{aligned}\mathbf{w} &\in \Sigma_2^\omega \cap \left(\bigcup_{n,m\in\mathbb{N}} \Sigma_2^{2^{n+m+1}} \{t_{n+m+2}, \overline{t_{n+m+2}}\} \prod_{k=n+m+3}^{\infty} \{\varepsilon, t_k, \overline{t_k}\}\right) \\ &\subseteq \bigcup_{n,m\in\mathbb{N}} \Sigma_2^{2^{n+m+1}} \{t_{n+m+2}, \overline{t_{n+m+2}}\}\{t_{n+m+3}, \overline{t_{n+m+3}}\}^\omega,\end{aligned}$$

so there is a $k \in \mathbb{N}$ such that $\mathbf{w}[2^k..\infty] \in \{t_{k+1}, \overline{t_{k+1}}\}\{t_{k+2}, \overline{t_{k+2}}\}^\omega$. By Observation 13 and Corollary 22, we obtain

$$\begin{aligned}(\mathrm{LSD}, \mathrm{USD})(\mathbf{t}, \mathbf{w}) &= (\mathrm{LSD}, \mathrm{USD})(\mathbf{t}[2^{k+2}..\infty], \mathbf{w}[2^{k+2}..\infty]) \\ &= (\mathrm{LSD}, \mathrm{USD})(\underbrace{\mathbf{t}[2^{k+2}..\infty]}_{\in\{t_{k+2},\overline{t_{k+2}}\}^\omega}, \underbrace{(\mathbf{w}[3\cdot 2^k..\infty])}_{\in\{t_{k+2},\overline{t_{k+2}}\}^\omega}[2^k..\infty]) \\ &\in \left(\left[\frac{1}{4}, \frac{3}{4}\right], \left[\frac{1}{4}, \frac{3}{4}\right]\right),\end{aligned}$$

as desired.

Case 4: From inspection of the Fife automaton for overlap-free infinite binary words, we see that $\mathbf{x} \in \{\mathtt{0}^n\{\mathtt{1},\mathtt{3}\}\mathtt{0}^m\{\mathtt{1},\mathtt{3}\}\mathbf{y} \,:\, n,m \in \mathbb{N}, \mathbf{y} \in \{\mathtt{0},\mathtt{1},\mathtt{3}\}^\omega\}$, whence

$$\begin{aligned}\mathbf{w} &\in \Sigma_2^\omega \cap \left(\bigcup_{n,m\in\mathbb{N}} \{t_n, \overline{t_n}\}\{t_{n+m+1}, \overline{t_{n+m+1}}\} \prod_{k=n+m+2}^{\infty} \{\varepsilon, t_k, \overline{t_k}\}\right) \\ &\subseteq \bigcup_{n,m\in\mathbb{N}} \{t_n, \overline{t_n}\}\{t_{n+m+1}, \overline{t_{n+m+1}}\}\{t_{n+m+2}, \overline{t_{n+m+2}}\}^\omega,\end{aligned}$$

so there are $k, l \in \mathbb{N}$ such that $\mathbf{w} \in \{t_k, \overline{t_k}\}\{t_{k+l+1}, \overline{t_{k+l+1}}\}\{t_{k+l+2}, \overline{t_{k+l+2}}\}^\omega$. By Observation 13 and Corollary 22, we obtain

$$\begin{aligned}(\mathrm{LSD}, \mathrm{USD})(\mathbf{t}, \mathbf{w}) &= (\mathrm{LSD}, \mathrm{USD})(\mathbf{t}[2^{k+l+2}..\infty], \mathbf{w}[2^{k+l+2}..\infty]) \\ &= (\mathrm{LSD}, \mathrm{USD})(\underbrace{\mathbf{t}[2^{k+l+2}..\infty]}_{\in\{t_{k+l+2},\overline{t_{k+l+2}}\}^\omega}, \underbrace{(\mathbf{w}[2^k+2^{k+l+1}..\infty])}_{\in\{t_{k+l+2},\overline{t_{k+l+2}}\}^\omega}[2^{k+l+1}-2^k..\infty]) \\ &\in \left(\left[\frac{1}{4}, \frac{3}{4}\right], \left[\frac{1}{4}, \frac{3}{4}\right]\right),\end{aligned}$$

as desired. ∎

6 Future work

Using the Fife automaton for overlap-free infinite binary words, we computed similarity densities of long prefixes of all overlap-free infinite binary words (up to a certain length) with prefixes of **t**. Inspection of the compuation results immediately suggests the following improvement to Theorem 18.

Conjecture 24. For all overlap-free $\mathbf{w} \in \Sigma_2^\omega \setminus \{\mathbf{t}, \overline{\mathbf{t}}\}$, we have $\frac{1}{3} \leq \mathrm{LSD}(\mathbf{w}, \mathbf{t}) \leq \mathrm{USD}(\mathbf{w}, \mathbf{t}) \leq \frac{2}{3}$.

Note that the bounds in Conjecture 24 are tight due to Proposition 17. Computational evidence also suggests that these bounds are also tight for many other overlap-free infinite binary words.

However, Conjecture 24 cannot be proved just by using the technique we used to prove Theorem 18. This is because the bounds in Lemma 20 (and, more transparently, Corollary 21) are tight. For example, $\mathrm{SD}(t_2, t_3[1..4]) = \mathrm{SD}(\mathtt{0110}, \mathtt{1101}) = \frac{1}{4}$. More generally, for any $n \in \mathbb{N}$, we have $\mathrm{SD}(t_{n+2}, t_{n+3}[2^n..2^{n+2} + 2^n - 1]) = \frac{1}{4}$.

On the other hand, our proof of Theorem 18 never used the overlap-free property directly; we merely used it indirectly via the Fife automaton. As such, our proof of Theorem 18 works for all images of FBE provided the argument to FBE is of the form required for one of the four cases presented in the proof, regardless of whether the resulting word is overlap-free. Namely, we have the following more general, but much more cumbersome, theorem.

Theorem 25. *For all* $\mathbf{x} \in \{\mathtt{1},\mathtt{2},\mathtt{3},\mathtt{4}\}\Sigma_5^*\mathtt{0}^\omega \cup \mathtt{0}^*\{\mathtt{2}(\mathtt{31}),\mathtt{4}(\mathtt{13})\}^\omega$

$$\cup\, \mathtt{0}^*\{\mathtt{2}(\mathtt{31})^*\{\varepsilon,\mathtt{3}\},\mathtt{4}(\mathtt{13})^*\{\varepsilon,\mathtt{1}\}\}\mathtt{0}\{\mathtt{1},\mathtt{3}\}\{\mathtt{0},\mathtt{1},\mathtt{3}\}^\omega \cup (\mathtt{0}^*\{\mathtt{1},\mathtt{3}\})^2\{\mathtt{0},\mathtt{1},\mathtt{3}\}^\omega$$

and

$$\mathbf{w} \in \begin{cases} \{\mathrm{FBE}(\mathbf{x},\mathtt{0}),\mathrm{FBE}(\mathbf{x},\mathtt{1})\}, & \textit{if } \mathbf{x} \textit{ ends in } \mathtt{0}^\omega; \\ \{\mathrm{FBE}(\mathbf{x})\}, & \textit{otherwise}, \end{cases}$$

we have

$$\frac{1}{4} \le \mathrm{LSD}(\mathbf{w},\mathbf{t}) \le \mathrm{USD}(\mathbf{w},\mathbf{t}) \le \frac{3}{4}.$$

Note that Theorem 25 is indeed more general than Theorem 18, since, for example, $\mathtt{13}^\omega$ is not a valid path in the Fife automaton for overlap-free infinite binary words (indeed, $\mathrm{FBE}(\mathtt{13}^\omega)$ begins with the overlap $\mathtt{01010}$) and $\mathrm{FBE}(\mathtt{13}^\omega)$ also is not just a shift of $\mathbf{t}$ or $\bar{\mathbf{t}}$, but Theorem 25 nevertheless implies that $\frac{1}{4} \le \mathrm{LSD}(\mathrm{FBE}(\mathtt{13}^\omega),\mathbf{t}) \le \mathrm{USD}(\mathrm{FBE}(\mathtt{13}^\omega),\mathbf{t}) \le \frac{3}{4}$.

Together, Conjecture 24 and Theorem 25 suggest the following more general question.

Question 26. For each $n \in \mathbb{N} \setminus \{0,1\}$, $r,s \in [0,1]$, and $\mathbf{x} \in \Sigma_n^\omega$, let

$$S_{n,r,s}(\mathbf{x}) := \{\mathbf{y} \in \Sigma_n^\omega \;:\; r \le \mathrm{LSD}(\mathbf{x},\mathbf{y}) \le \mathrm{USD}(\mathbf{x},\mathbf{y}) \le s\}.$$

What are $S_{2,\frac{1}{4},\frac{3}{4}}(\mathbf{t})$ and $S_{2,\frac{1}{3},\frac{2}{3}}(\mathbf{t})$?

Another avenue of investigation is to consider what makes $\mathbf{t}$ so special in the sense of Theorem 18. As mentioned in the introduction, Theorem 18 is false if we replace $\mathbf{t}$ with an arbitrary overlap-free infinite binary word. However, perhaps there are specific words other than $\mathbf{t}$ and $\bar{\mathbf{t}}$ that do share similar properties. In other words, we raise the following question.

Question 27. Let $\mathscr{O}$ denote the set of all overlap-free infinite binary words.

What is $\{\mathbf{x} \in \Sigma_2^\omega \;:\; \mathscr{O} \subseteq S_{2,\frac{1}{4},\frac{3}{4}}(\mathbf{x})\}$? What if we replace $\frac{1}{4},\frac{3}{4}$ with $\frac{1}{3},\frac{2}{3}$?

A third avenue of investigation is to consider what occurs in words that avoid higher powers in place of being overlap-free (which are essentially $(2+\varepsilon)$- or 2^+-powers). In fact, there is a Fife automaton characterizing $\frac{7}{3}$-power-free infinite binary words having the same encoding mechanism as the Fife automaton for overlap-free infinite binary words but with more states and different transitions [3, 9]. However, initial inspection of the automaton for $\frac{7}{3}$-power-free infinite binary words suggests that our proof of Theorem 18 cannot be extended to account for all $\frac{7}{3}$-power-free infinite binary words because there are many more edges labeled $\mathtt{2}$ and $\mathtt{4}$ in the Fife automaton for $\frac{7}{3}$-power-free infinite binary words, resulting in valid paths that contain infinitely many $\mathtt{2}$s and $\mathtt{4}$s, but our proof of Theorem 18 heavily relied on there being at most one occurrence of $\mathtt{2}$ or $\mathtt{4}$ (which must be preceeded by a string of $\mathtt{0}$s if it occurs) in the path taken through the automaton so that the infinite binary word corresponding to the path eventually "lags behind" the prefixes t_n of $\mathbf{t}$ in the sense that each successive n^{th} symbol in the path can only generate positions prior to 2^n. Nevertheless, computational evidence suggests that Theorem 18 and even Conjecture 24 can be generalized even further.

Conjecture 28. For all $\frac{7}{3}$-power-free $\mathbf{w} \in \Sigma_2^\omega \setminus \{\mathbf{t},\bar{\mathbf{t}}\}$, $\frac{1}{3} \le \mathrm{LSD}(\mathbf{w},\mathbf{t}) \le \mathrm{USD}(\mathbf{w},\mathbf{t}) \le \frac{2}{3}$.

Finally, we revisit the notion, already mentioned in Remark 7, that LSD and USD are not new ideas, and not just in number theory. In fact, $1-\mathrm{LSD}$ is a pseudometric on $\Sigma^{\mathbb{N}}$, called the Besicovitch pseudometric, which has already been studied from the perspective of discrete dynamical systems such as

[2]. Also studied in [2] is the Weyl pseudometric, which suggests the following slightly different notion of similarity density, considering all blocks of a given size instead of just blocks from the beginning.

$$\mathrm{LSD}_{\mathrm{Weyl}}(\mathbf{x},\mathbf{y}) = \liminf_{n\to\infty} \inf_{k\in\mathbb{N}} \mathrm{SD}(\mathbf{x}[k..k+n-1],\mathbf{y}[k..k+n-1]),$$
$$\mathrm{USD}_{\mathrm{Weyl}}(\mathbf{x},\mathbf{y}) = \limsup_{n\to\infty} \sup_{k\in\mathbb{N}} \mathrm{SD}(\mathbf{x}[k..k+n-1],\mathbf{y}[k..k+n-1]).$$

With this notion of Weyl similarity density, analogous to the Besicovitch case, we have that $1-\mathrm{LSD}_{\mathrm{Weyl}}$ is the Weyl pseudometric. The Besicovitch and Weyl pseudometrics share some topological properties, but the Besicovitch pseudometric is complete while the Weyl pseudometric is not [2]. This fact suggests one might be able to shed further light on some of the questions above by also considering the Weyl similarity density; perhaps several different notions of similarity density, when taken together, can characterize the overlap-free infinite binary words.

Acknowledgments. We are grateful to Chao Hsien Lin for having suggested the question we study here. We thank the referees for a careful reading, and Joel Ouaknine and Stefan Kiefer for having pointed out the paper [2].

References

[1] J.-P. Allouche & J. Shallit (1999): *The ubiquitous Prouhet-Thue-Morse sequence*. In C. Ding, T. Helleseth & H. Niederreiter, editors: *Sequences and Their Applications, Proceedings of SETA '98*, Springer-Verlag, pp. 1–16, doi:10.1007/978-1-4471-0551-0_1.

[2] F. Blanchard, E. Formenti & P. Kůrka (1997): *Cellular automata in the Cantor, Besicovitch, and Weyl topological spaces*. *Complex Systems* 11, pp. 107–123.

[3] V. D. Blondel, J. Cassaigne & R. M. Jungers (2009): *On the number of α-power-free binary words for $2<\alpha\le 7/3$*. *Theoret. Comput. Sci.* 410, pp. 2823–2833, doi:10.1016/j.tcs.2009.01.031.

[4] S. Brown, N. Rampersad, J. Shallit & T. Vasiga (2006): *Squares and overlaps in the Thue-Morse sequence and some variants*. *RAIRO Inform. Théor. App.* 40, pp. 473–484, doi:10.1051/ita:2006030.

[5] E. D. Fife (1980): *Binary sequences which contain no BBb*. *Trans. Amer. Math. Soc.* 261, pp. 115–136, doi:10.1090/S0002-9947-1980-0576867-5.

[6] E. Grant, J. Shallit & T. Stoll (2009): *Bounds for the discrete correlation of infinite sequences on k symbols and generalized Rudin-Shapiro sequences*. *Acta Arith.* 140, pp. 345–368, doi:10.4064/aa140-4-5.

[7] K. Mahler (1927): *On the translation properties of a simple class of arithmetical functions*. *J. Math. and Phys.* 6, pp. 158–163.

[8] P. Ochem, N. Rampersad & J. Shallit (2008): *Avoiding approximate squares*. *Internat. J. Found. Comp. Sci.* 19, pp. 633–648, doi:10.1142/S0129054108005863.

[9] N. Rampersad, J. Shallit & A. Shur (2011): *Fife's theorem for (7/3)-powers*. In P. Ambroz, S. Holub & Z. Masakova, editors: *WORDS 2011, 8th International Conference*, pp. 189–198, doi:10.4204/EPTCS.63.25.

[10] J. Shallit (2011): *Fife's theorem revisited*. In G. Mauri & A. Leporati, editors: *Developments in Language Theory*, *Lecture Notes in Computer Science* 6795, Springer-Verlag, pp. 397–405, doi:10.1007/978-3-642-22321-1_34.

[11] R. Yarlagadda & J. E. Hershey (1984): *Spectral properties of the Thue-Morse sequence*. *IEEE Trans. Commun.* 32, pp. 974–977, doi:10.1109/TCOM.1984.1096162.

[12] R. Yarlagadda & J. E. Hershey (1990): *Autocorrelation properties of the Thue-Morse sequence and their use in synchronization*. *IEEE Trans. Commun.* 38, pp. 2099–2102, doi:10.1109/26.64649.

Cooperating Distributed Grammar Systems of Finite Index Working in Hybrid Modes

Henning Fernau
Fachbereich 4—Abteilung Informatik
Universität Trier
D-54286 Trier, Germany
fernau@uni-trier.de

Rudolf Freund
Institut für Computersprachen
Technische Universität Wien
Favoritenstr. 9, A-1040 Wien, Austria
rudi@emcc.at

Markus Holzer
Institut für Informatik
Universität Gießen,
Arndtstraße 2, D-35392 Gießen, Germany
holzer@informatik.uni-giessen.de

We study cooperating distributed grammar systems working in hybrid modes in connection with the finite index restriction in two different ways: firstly, we investigate cooperating distributed grammar systems working in hybrid modes which characterize programmed grammars with the finite index restriction; looking at the number of components of such systems, we obtain surprisingly rich lattice structures for the inclusion relations between the corresponding language families. Secondly, we impose the finite index restriction on cooperating distributed grammar systems working in hybrid modes themselves, which leads us to new characterizations of programmed grammars of finite index.

Keywords: CD grammar systems; finite index; hybrid modes; programmed grammars
AMS MSC[2010] classification: 68Q42; 68Q45

1 Introduction

Cooperating distributed (CD) grammar systems first were introduced in [12] with motivations related to two-level grammars. Later, the investigation of CD grammar systems became a vivid area of research after relating CD grammar systems with Artificial Intelligence (AI) notions [2], such as multi-agent systems or blackboard models for problem solving. From this point of view, motivations for CD grammar systems can be summarized as follows: several grammars (agents or experts in the framework of AI), mainly consisting of rule sets (corresponding to scripts the agents have to obey to) are cooperating in order to work on a sentential form (representing their common work), finally generating terminal words (in this way solving the problem). The picture one has in mind is that of several grammars (mostly, these are simply classical context-free grammars called "components" in the theory of CD grammar systems) "sitting" around a table where there is lying the common workpiece, a sentential form. Some component takes this sentential form, works on it, i.e., it performs some derivation steps, and then returns it onto the table such that another component may continue the work.

In classical CD grammar systems, all components work in the same derivation mode. It is of course natural to alleviate this requirement, because it simply refers to different capabilities and working regulations of different experts in the original CD motivation. This leads to the notion of so-called hybrid CD grammar systems introduced by Mitrana and Păun in [13, 14]. We investigate internally hybrid derivation modes which partly allow for new characterizations of the external hybridizations explained above.

This paper belongs to a series of papers on hybrid modes in CD grammar systems: as predecessors, we mention that [6] introduces hybrid modes in CD array grammar systems as a natural specification tool for array languages and [10] investigates accepting CD grammar systems with hybrid modes; the two most relevant papers are [8, 9] where the most important aspects of internal and external mode hybridizations are discussed for the case of word languages.

Here, we will continue this line of research, focussing on the finite index restriction. The paper is organized as follows. In the next section, we introduce the necessary notions. In Section 3, we review important notions and results in connection with the finite index restriction. Section 4 is devoted to the study of internally hybrid CD grammar systems with the (explicit) restriction of being of finite index; we establish infinite hierarchies with respect to the number of components and the number of maximal derivation steps per component. In Section 5, we refine our previous analysis (published in [9]) showing characterizations of programmed grammars of finite index by several variants of (internally) hybrid CD grammar systems, also considering the number of grammar components as an additional descriptional complexity parameter. In the last section, we review our results again and give a prospect on possible future work.

2 Definitions

We assume the reader to be familiar with some basic notions of formal language theory and regulated rewriting, as contained in [15] and [4]. In particular, details on programmed grammars can be found there. In general, we have the following conventions: $\subseteq$ denotes inclusion, while $\subset$ denotes strict inclusion; the set of positive integers is denoted by $\mathbb{N}$. The empty word is denoted by λ; $|\alpha|_A$ denotes the number of occurrences of the symbol A in α. We consider two languages L_1, L_2 to be equal if and only if $L_1 \setminus \{\lambda\} = L_2 \setminus \{\lambda\}$, and we simply write $L_1 = L_2$ in this case. The families of languages generated by linear context-free and context-free grammars are denoted by $\mathscr{L}(\mathrm{LIN})$ and $\mathscr{L}(\mathrm{CF})$, respectively, and the family of finite languages is denoted by $\mathscr{L}(\mathrm{FIN})$. We attach $-\lambda$ in our notations for formal language classes if erasing rules are not permitted. Notice that we use bracket notations in order to express that the equation holds both in case of forbidding erasing rules and in the case of admitting erasing rules (consistently neglecting the contents between the brackets).

Next we introduce programmed grammars, a well-known concept in the area of regulated rewriting.

A *programmed grammar* is a septuple $G = (N, T, P, S, \Lambda, \sigma, \phi)$, where N, T, and $S \in N$ are the set of nonterminals, the set of terminals, and the start symbol, respectively. In the following we use V_G to denote the set $N \cup T$, which is the complete working alphabet of the grammar. P is the finite set of context-free rules $A \to z$ with $A \in N$ and $z \in V_G^*$, and Λ is a finite set of labels (for the rules in P), such that Λ can also be interpreted as a function which outputs a rule when being given a label; σ and ϕ are functions from Λ into the set of subsets of Λ. For (x, r_1), (y, r_2) in $V_G^* \times \Lambda$ and $\Lambda(r_1) = (A \to z)$, we write $(x, r_1) \Rightarrow (y, r_2)$ if and only if either

1. $x = x_1 A x_2$, $y = x_1 z x_2$, and $r_2 \in \sigma(r_1)$, or
2. $x = y$, the rule $A \to z$ is not applicable to x, and $r_2 \in \phi(r_1)$.

In the latter case, the derivation step is performed in the so-called *appearance checking mode*. The set $\sigma(r_1)$ is called success field and the set $\phi(r_1)$ is called failure field of r_1. As usual, the reflexive transitive closure of $\Rightarrow$ is denoted by $\Longrightarrow^*$. The language generated by G is defined as

$$L(G) = \{\, w \in T^* \mid (S, r_1) \Longrightarrow^* (w, r_2) \text{ for some } r_1, r_2 \in \Lambda \}.$$

The family of languages generated by [λ-free] programmed grammars containing only context-free rules is denoted by $\mathscr{L}(\mathrm{P},\mathrm{CF}[-\lambda],\mathrm{ac})$. When no appearance checking features are involved, i.e., $\phi(r)=\emptyset$ for each label $r\in\Lambda$, we obtain the family $\mathscr{L}(\mathrm{P},\mathrm{CF}[-\lambda])$.

Finally, we now define cooperating distributed (CD) and hybrid cooperating distributed (HCD) grammar systems.

A *CD grammar system* of degree n, with $n\geq 1$, is an $(n+3)$-tuple $G=(N,T,S,P_1,P_2,\ldots,P_n)$, where N, T are disjoint alphabets of nonterminal and terminal symbols, respectively, $S\in N$ is the start symbol, and $P_1,\ldots,P_n$ are finite sets of rewriting rules over $N\cup T$. Throughout this paper, we consider only regular, linear context-free, and context-free rewriting rules. For $x,y\in(N\cup T)^*$ and $1\leq i\leq n$, we write $x\Longrightarrow_i y$ if and only if $x=x_1Ax_2$, $y=x_1zx_2$ for some $A\to z\in P_i$. Hence, subscript i refers to the component to be used. Accordingly, $x\Longrightarrow_i^m y$ denotes an m-step derivation using component number i, where $x\Longrightarrow_i^0 y$ if and only if $x=y$.

We define the *classical basic modes* $B=\{*,t\}\cup\{\leq k,=k,\geq k\mid k\in\mathbb{N}\}$ and let

$$D=B\cup\{(\geq k\wedge\leq\ell)\mid k,\ell\in\mathbb{N},k\leq\ell\}\cup\{(t\wedge\leq k),(t\wedge=k),(t\wedge\geq k)\mid k\in\mathbb{N}\}.$$

For $f\in D$ we define the relation $\Longrightarrow_i^f$ by

$$x\Longrightarrow_i^f y\iff\exists m\geq 0:(x\Longrightarrow_i^m y\wedge P(f,m,i,y)),$$

where P is a predicate defined as follows (let $k\in\mathbb{N}$ and $f_1,f_2\in B$):

predicate	definition
$P(=k,m,i,y)$	$m=k$
$P(\leq k,m,i,y)$	$m\leq k$
$P(\geq k,m,i,y)$	$m\geq k$
$P(*,m,i,y)$	$m\geq 0$
$P(t,m,i,y)$	$\neg\exists z(y\Longrightarrow_i z)$
$P((f_1\wedge f_2),m,i,y)$	$P(f_1,m,i,y)\wedge P(f_2,m,i,y)$

Observe that not every combination of modes as introduced above is a genuinely hybrid mode. For example, the $(\geq k\wedge\leq k)$-mode is just another notation for the $=k$-mode. Especially, $*$ may be used as a "don't care" in our subsequent notations, since $P((*\wedge f_2),m,i,y)$ if and only if $P(f_2,m,i,y)$.

If each component of a CD grammar system may work in a different mode, then we get the notion of an *(externally) hybrid CD (HCD) grammar system* of degree n, with $n\geq 1$, which is an $(n+3)$-tuple $G=(N,T,S,(P_1,f_1),(P_2,f_2),\ldots,(P_n,f_n))$, where $N,T,S,P_1,\ldots,P_n$ are as in a CD grammar system, and $f_i\in D$, for $1\leq i\leq n$. Thus, we can define the language *generated* by a HCD grammar system as:

$$\begin{aligned}L(G)\;:=\;\{w\in T^*\mid\;&S\Rightarrow_{i_1}^{f_{i_1}}w_1\Rightarrow_{i_2}^{f_{i_2}}\ldots\Rightarrow_{i_{m-1}}^{f_{i_{m-1}}}w_{m-1}\Rightarrow_{i_m}^{f_{i_m}}w_m=w\\&\text{with } m\geq 1,\ 1\leq i_j\leq n,\text{ and }1\leq j\leq m\}\end{aligned}$$

If $F\subseteq D$ and $X\in\{\mathrm{LIN},\mathrm{CF}\}$, then the family of languages generated by [λ-free] HCD grammar systems with degree at most n using rules of type X, each component working in one of the modes contained in F, is denoted by $\mathscr{L}(\mathrm{HCD}_n,X[-\lambda],F)$. In a similar way, we write $\mathscr{L}(\mathrm{HCD}_\infty,X[-\lambda],F)$ when the number of components is not restricted. If F is a singleton $\{f\}$, we simply write $\mathscr{L}(\mathrm{CD}_n,X[-\lambda],f)$, where $n\in\mathbb{N}\cup\{\infty\}$; additionally, we write $L_f(G)$ instead of $L(G)$ to denote the language generated by the CD grammar system G in the mode f.

The following example is taken from [8, Theorem 24], as we need this language in the following of this paper. This should also help to clarify our definitions.

Example 1 *The non-context-free language $L = \{ a_1^n a_2^n \dots a_{k+1}^n \mid n \geq 1 \}$ can be generated by the CD grammar system $G = (N, T, S_1, P_1, P_2)$, where P_1, P_2 work in the $(t \wedge \geq k)$-mode, $k \geq 2$. For both components, we take $N = \{ S_i, A_i, A_i', \mid 1 \leq i \leq k \}$ as nonterminal alphabet and $T = \{a_1, \dots, a_{k+1}\}$ as terminal alphabet. The components P_1 and P_2 are defined as follows:*

$$\begin{aligned} P_1 &= \{ S_i \to S_{i+1} \mid 1 \leq i < k \} \cup \{ S_k \to A_1 \cdots A_k \} \cup \\ &\quad \{ A_i' \to A_i \mid 1 \leq i \leq k \} \quad \textit{and} \\ P_2 &= \{ A_i \to a_i A_i' \mid 1 \leq i \leq k-1 \} \cup \{ A_k \to a_k A_k' a_{k+1} \} \cup \\ &\quad \{ A_i \to a_i \mid 1 \leq i \leq k-1 \} \cup \{ A_k \to a_k a_{k+1} \}. \end{aligned}$$

Then we have $L(G) = L$, since every derivation of G leading to a terminal word is of the form

$$S_1 \Longrightarrow_1^{=k} A_1 \dots A_k \quad \cdots \quad \Longrightarrow_2^{=k} a_1^n \dots a_k^n a_{k+1}^n,$$

where the intermediate steps are of the form

$$a_1^i A_1 \dots a_k^i A_k a_{k+1}^i \Longrightarrow_2^{=k} a_1^{i+1} A_1'^{i+1}{}_k A_k'^{i+1}{}_{k+1} \Longrightarrow_1^{=k} a_1^{i+1} A_1 \dots a_k^{i+1} A_k a_{k+1}^{i+1};$$

if a non-vanishing number of occurrences of A_i' less than k is obtained by using P_2 then neither P_1 nor P_2 can perform k derivation steps any more. Hence, G generates L.

The same grammar system, viewed as a $(\mathrm{CD}_2, \mathrm{CF}, (t \wedge = k))$ grammar system, generates L, too.

3 The Finite Index Restriction

The finite index restriction is defined as follows: let G be an arbitrary grammar type (from those discussed in Section 2) and let N, T, and $S \in N$ be its nonterminal alphabet, terminal alphabet, and axiom, respectively. For a derivation

$$D : S = w_1 \Longrightarrow w_2 \Longrightarrow \cdots \Longrightarrow w_n = w \in T^*$$

according to G, we set $\mathit{ind}(D, G) = \max\{ |w_i|_N \mid 1 \leq i \leq n \}$. In the case of programmed grammars we assume to have a derivation of the form

$$D : (S, r_1) = (w_1, r_1) \Longrightarrow (w_2, r_2) \Longrightarrow \cdots \Longrightarrow (w_n, r_n) = (w, r_n) \in T^* \times \Lambda.$$

For $w \in T^*$, we define $\mathit{ind}(w, G) = \min\{ \mathit{ind}(D, G) \mid D$ is a derivation for w in $G \}$. The *index of grammar G* is defined as $\mathit{ind}(G) = \sup\{ \mathit{ind}(w, G) \mid w \in L(G) \}$. For a language L in the family $\mathscr{L}(\mathrm{X})$ of languages generated by grammars of type X, we define $\mathit{ind}_{\mathrm{X}}(L) = \inf\{ \mathit{ind}(G) \mid L(G) = L$ and G is of type X$\}$. For a family $\mathscr{L}(\mathrm{X})$, we set

$$\begin{aligned} \mathscr{L}_n(\mathrm{X}) &= \{ L \mid L \in \mathscr{L}(\mathrm{X}) \text{ and } \mathit{ind}_{\mathrm{X}}(L) \leq n \} \quad \text{for } n \in \mathbb{N}, \text{ and} \\ \mathscr{L}_{\mathit{fin}}(\mathrm{X}) &= \bigcup_{n \geq 1} \mathscr{L}_n(\mathrm{X}). \end{aligned}$$

It is well-known that the class of programmed languages of index m can be characterized in various ways, compare, e.g., [4, 11, 16]. Especially, normal forms are available. For the reader's convenience, we quote [9, Theorem 9] in the following, since we will use it to give a sharpened and broadened version of [3, Theorem 3.26], which leads us to new characterizations of the classes $\mathscr{L}_m(\mathrm{P}, \mathrm{CF})$ and $\mathscr{L}_{\mathit{fin}}(\mathrm{P}, \mathrm{CF})$.

Theorem 1 *For every* $(\mathrm{P},\mathrm{CF},\mathrm{ac})$ *grammar* $G = (N,T,P,S,\Lambda,\sigma,\phi)$ *whose generated language is of index* $n \in \mathbb{N}$*, there exists an equivalent* $(\mathrm{P},\mathrm{CF},\mathrm{ac})$ *grammar* $G' = (N',T,P',S',\Lambda,\sigma',\phi')$ *whose generated language is also of index n and which satisfies the following three properties:*

1. *There exists a special start production with a unique label* p_0*, which is the only production where the start symbol* S' *appears.*
2. *There exists a function* $f : \Lambda' \to \mathbb{N}_0^{N'}$ *such that, if* $S' \Longrightarrow^* v \Longrightarrow_p w$ *is a derivation in* G'*, then* $(f(p))(A) = |v|_A$ *for every nonterminal A.*
3. *If* $D : S' = v_0 \Longrightarrow_{r_1} v_1 \Longrightarrow_{r_2} v_2 \cdots \Longrightarrow_{r_m} v_m = w$ *is a derivation in* G' *then, for every* v_i, $0 \le i \le m$*, and every nonterminal A,* $|v_i|_A \le 1$*. In other words, every nonterminal occurs at most once in any derivable sentential form.*

Moreover, we may assume that either G' *is a* (P,CF) *grammar, i.e., we have* $\phi' = \emptyset$*, or that* G' *is a* $(\mathrm{P},\mathrm{CF},\mathrm{ut})$ *grammar, i.e., we have* $\phi' = \sigma'$*.*

In the following, we will refer to a grammar satisfying the three conditions listed above as *nonterminal separation form (NSF)*.

Theorem 1 shows that, in contrast to the general case, where $\mathscr{L}(\mathrm{P},\mathrm{CF},\mathrm{ac}) \supset \mathscr{L}(\mathrm{P},\mathrm{CF})$, the appearance checking feature does not increase the generative power of programmed grammars if the finite index restriction is imposed; especially we have $\mathscr{L}_m(\mathrm{P},\mathrm{CF}[-\lambda],\mathrm{ac}) = \mathscr{L}_m(\mathrm{P},\mathrm{CF}[-\lambda])$.

Recall that we have shown in [9, Theorem 30] the following link between hybrid CDGS and the finite index restriction on programmed grammars.

Theorem 2 *Let* $\ell \in \mathbb{N}$ *and* $\Delta \in \{\le,=\}$*. Then we have:*

$$\begin{aligned} \mathscr{L}(\mathrm{HCD}_\infty,\mathrm{CF}[-\lambda],\{\,(t \wedge \Delta k) \mid k \ge 1\,\}) &= \bigcup_{k \in \mathbb{N}} \mathscr{L}(\mathrm{CD}_\infty,\mathrm{CF}[-\lambda],(t \wedge \Delta k)) \\ &= \mathscr{L}(\mathrm{CD}_\infty,\mathrm{CF}[-\lambda],(t \wedge \Delta l)) \\ &= \mathscr{L}(\mathrm{CD}_\infty,\mathrm{CF}[-\lambda],(t \wedge \Delta 1)) \\ &= \mathscr{L}_{fin}(\mathrm{P},\mathrm{CF}[-\lambda],\mathrm{ac}). \end{aligned}$$

Unfortunately, our proof did not bound the number of components of the CD grammar system. This is not just a coincidence, as we will see in this paper.

4 Infinite Hierarchies for CD Grammar Systems Working in Hybrid Modes

Our task will be the study of the language families $\mathscr{L}(\mathrm{CD}_n,\mathrm{CF}[-\lambda],(t \wedge \Delta k))$ for different $n,k \in \mathbb{N}$ and $\Delta \in \{\le,=\}$. First we give some characterizations of well-known language families, namely the family of finite languages and the family of linear languages.

Lemma 3 *For every* $k \in \mathbb{N}$*, and* $\Delta \in \{\le,=\}$*, we have*

$$\mathscr{L}(\mathrm{FIN}) = \mathscr{L}(\mathrm{CD}_1,\mathrm{CF}[-\lambda],(t \wedge \Delta k)).$$

Proof. Since we have only one component, by definition of the $(t \wedge \Delta k)$-mode, every derivation has length at most k, so that we only get finite languages. If $L = \{w_1, w_2, \ldots, w_m\} \subseteq T^*$ is some finite language, then the grammar $G = (\{S\} \times \{1, \ldots, k\}, T, (S,1), P)$ with

$$P = \{ (S,i) \to (S,i+1) \mid 1 \leq i < k \} \cup \{ (S,k) \to w_j \mid 1 \leq j \leq m \})$$

generates L. □

Now we turn our attention to CD grammar systems with two components working in the $(t \wedge \Delta 1)$-mode for $\Delta \in \{\leq, =\}$.

Lemma 4 *For $\Delta \in \{\leq, =\}$ we have $\mathscr{L}(\mathrm{LIN}) = \mathscr{L}(\mathrm{CD}_2, \mathrm{CF}[-\lambda], (t \wedge \Delta 1))$.*

Proof. Let L be generated by the linear grammar $G = (N, T, S, P)$. Grammar G is simulated by the CD grammar system $G' = (N \cup N', T, S, P_1, P_2)$ where N' contains primed versions of the nonterminals of G, set P_1 contains colouring unit productions $B \to B'$ for every nonterminal $B \in N$, and P_2 contains, for every production $A \to w \in P$, a production $A' \to w$. The simulation of G by G' proceeds by applying P_2 and P_1 in sequence until the derivation stops.

On the other hand, it is easy to see that no sentential form generated by some $(\mathrm{CD}_2, \mathrm{CF}[-\lambda], (t \wedge \Delta 1))$-system (eventually leading to a terminal string) can contain more than one nonterminal. Otherwise, we must have applied a production $A \to w$ of say the first component, where w contains at least two nonterminals. All nonterminals occurring in w cannot be processed further by the first component, since otherwise it violates the $(t \wedge \Delta 1)$-mode restriction. But nearly the same argument applies to the second component, too: it can only process at most one of the nonterminals just introduced. Hence, no terminal string is derivable in this way.

Therefore, one can omit all productions containing more than one nonterminal on their right-hand sides, so that there are only linear rules left. Furthermore, one can also omit all productions in a component containing a nonterminal as its right-hand side which occurs also as the left-hand side of the originally given component as this would lead to more than one derivation step in the same component. Now, one can put all remaining productions together yielding the rule set of a simulating linear grammar. □

In the general case, i.e., two components working together in the $(t \wedge \Delta k)$-mode, for $k \in \mathbb{N}$ and $\Delta \in \{\leq, =\}$, we first give some lower bounds.

Theorem 5 *Let $k \in \mathbb{N}$, $\Delta \in \{\leq, =\}$. Then we have:*

1. $\mathscr{L}(\mathrm{LIN}) = \mathscr{L}_1(\mathrm{CF}) = \mathscr{L}(\mathrm{CD}_2, \mathrm{CF}[-\lambda], (t \wedge \Delta 1))$;
2. $\mathscr{L}_k(\mathrm{CF}) \subseteq \mathscr{L}(\mathrm{CD}_2, \mathrm{CF}[-\lambda], (t \wedge \Delta k))$ *for* $k \geq 1$, *and*
3. $\mathscr{L}_k(\mathrm{CF}) \subset \mathscr{L}(\mathrm{CD}_2, \mathrm{CF}[-\lambda], (t \wedge = k))$ *for* $k > 1$;
4. $\mathscr{L}_{fin}(\mathrm{CF}) \subset \bigcup_{k \in \mathbb{N}} \mathscr{L}(\mathrm{CD}_2, \mathrm{CF}[-\lambda], (t \wedge = k))$.

Proof.

1. It is easy to see that $\mathscr{L}(\mathrm{LIN}) = \mathscr{L}_1(\mathrm{CF})$. Hence, this statement is equivalent to the assertion of the previous lemma.
2. Let $G = (N, T, S, P)$ be a context-free grammar of index k. Without loss of generality, we assume that every nonterminal occurs as the left-hand side of some production in P. Let N' be the set of primed nonterminal symbols. Grammar G is simulated by the CD grammar system

$G' = (N \cup N', T, S, P_1, P_2)$, where P_1 contains colouring unit productions $B \to B'$, and $B \to B$ for every nonterminal $B \in N$, and P_2, for every production $A \to w \in P$, contains productions $A' \to w$ and $A' \to A'$. The unit productions $B \to B$ in P_1 and $A' \to A'$ in P_2 guarantee that at most k nonterminals can occur in any sentential form that can be derived in G'.

3. A separating example was already explained in Example 1: there the languages $\{ a_1^n a_2^n \ldots a_{k+1}^n \mid n \geq 1 \}$ was shown to be in $\mathscr{L}(\mathrm{CD}_2, \mathrm{CF}[-\lambda], (t\wedge = k))$ for $k \geq 2$, but obviously these languages are not context-free.

4. Follows from 3.

□

Unfortunately, we do not know whether the inclusion

$$\mathscr{L}_k(\mathrm{CF}) \subseteq \mathscr{L}(\mathrm{CD}_2, \mathrm{CF}[-\lambda], (t\wedge \leq k))$$

in the previous theorem is strict or not. By the prolongation technique introduced in [8], we know that the classes $\mathscr{L}(\mathrm{CD}_n, \mathrm{CF}[-\lambda], (t \wedge \Delta k))$, for $\Delta \in \{\leq, =\}$ form a prime number lattice, i.e.,

$$\mathscr{L}(\mathrm{CD}_n, \mathrm{CF}[-\lambda], (t \wedge \Delta k)) \subseteq \mathscr{L}(\mathrm{CD}_n, \mathrm{CF}[-\lambda], (t \wedge \Delta \ell \cdot k) \quad \text{for } \ell \in \mathbb{N},$$

with the least element $\mathscr{L}(\mathrm{CD}_n, \mathrm{CF}[-\lambda], (t \wedge \Delta 1))$. This prolongation technique is based on the simple idea to "slow down" a derivation using $A \to w$ of the original CDGS by intercalating productions of the form $A \to A'$, $A' \to A''$, ..., $A^{(j)} \to w$ within the simulating CDGS. It will be used on several occasions in this paper. Obviously, we also have the trivial inclusions

$$\mathscr{L}(\mathrm{CD}_n, \mathrm{CF}[-\lambda], (t \wedge \Delta k)) \subseteq \mathscr{L}(\mathrm{CD}_{n+1}, \mathrm{CF}[-\lambda], (t \wedge \Delta k)) \quad \text{for } \Delta \in \{\leq, =\}.$$

The question arises whether all these hierarchies are strict. At least we will be able to show that both with respect to k – for a fixed number of components n – as well as with respect to the number of components n – for a fixed derivation mode $(t \wedge \Delta k)$, $\Delta \in \{\leq, =\}$ – we obtain infinite hieararchies. In order to prove these hierarchies, we show some general theorems relating the number of components and the bound of the number of symbols to be rewritten by one component with the finite index of a simulating programmed grammar.

Theorem 6 *Let $n, k \in \mathbb{N}$ and $\Delta \in \{\leq, =\}$. Then, we have*

$$\mathscr{L}(\mathrm{CD}_n, \mathrm{CF}[-\lambda], (t \wedge \Delta k)) \subseteq \mathscr{L}_{n \cdot k}(\mathrm{P}, \mathrm{CF}[-\lambda], ac).$$

Proof. Let $G = (N, T, S, P_1, P_2, \ldots, P_n)$ be a CD grammar system working in the $(t\wedge = k)$-mode. Let $P_i = \{ A_{ij} \to w_{ij} \mid 1 \leq j \leq N(i) \}$. G can be simulated by the programmed grammar $G' = (N \cup \{F\}, T, S, P, \Lambda, \sigma, \phi)$, with label-set

$$\begin{array}{rcl} \Lambda & = & \{ (i,j,\kappa) \mid 1 \leq i \leq n, 1 \leq j \leq N(i), 1 \leq \kappa \leq k \} \\ & \cup & \{ (i,j) \mid 1 \leq i \leq n, 1 \leq j \leq N(i) \}. \end{array}$$

Now, let $1 \leq i \leq n, 1 \leq j \leq N(i), 1 \leq \kappa \leq k$. Then, we set $\Lambda((i,j,\kappa)) = A_{ij} \to w_{ij}$, success field $\sigma((i,j,\kappa)) = \{ (i,j',\kappa+1) \mid 1 \leq j' \leq N(i) \}$, if $\kappa < k$, and $\sigma((i,j,k)) = \{(i,1)\}$, and failure field $\phi((i,j,\kappa)) = \emptyset$. Moreover, we let $\Lambda((i,j)) = A_{ij} \to F$, failure field $\phi((i,j)) = \{(i,j+1)\}$ if $j < N(i)$, $\phi((i,N(i))) = \{ (i',j',1) \mid 1 \leq i' \leq n, 1 \leq j' \leq N(i') \}$, and success field $\sigma((i,j)) = \emptyset$.

An application of P_i is simulated by a sequence of productions labeled with

$$(i, j_1, 1), \ldots, (i, j_k, k), (i, 1), \ldots, (i, N(i)).$$

In each such sequence, at most k symbols can be processed. Since there are n sets of productions P_i, only sentential forms containing at most $n \cdot k$ nonterminals can hope for termination. Therefore, the simulating programmed grammar has index at most $n \cdot k$, which can be seen by induction.

The $(t\wedge \leq k)$-mode case can be treated in a similar way: we just define $\phi((i, j, \kappa))$ to equal $\sigma((i, j, \kappa))$ instead of taking $\phi((i, j, \kappa)) = \emptyset$. □

Before we can establish the infinite hierarchies for the families $\mathscr{L}(\mathrm{CD}_n, \mathrm{CF}[-\lambda], (t \wedge \Delta k))$ with respect to n and k, respectively, we need the following theorem shown in [4, page 160, Theorem 3.1.7]:

Theorem 7 $S_{n+1} = \{ b(a^i b)^{2\cdot(n+1)} \mid i \geq 1 \} \in \mathscr{L}_{n+1}(\mathrm{P}, \mathrm{CF}) \setminus \mathscr{L}_n(\mathrm{P}, \mathrm{CF})$ *for all* $n \in \mathbb{N}$.

These separating languages can also be generated by CD grammar systems working in the internally hybrid modes $(t\wedge = k+1)$ and $(t\wedge \leq k+1)$:

Theorem 8 *Let* $n, k \in \mathbb{N}$.

1. $S_{n\cdot k} \in \mathscr{L}(\mathrm{CD}_{n+1}, \mathrm{CF}-\lambda, (t\wedge = k+1))$*, i.e.,* $S_{n\cdot k}$ *can be generated by a CD-grammar system with* $n+1$ *context-free components, without erasing productions, working in the* $(t\wedge = k+1)$*-mode.*
2. $S_{n\cdot k} \in \mathscr{L}(\mathrm{CD}_{2n+1}, \mathrm{CF}-\lambda, (t\wedge \leq k+1))$*, i.e.,* $S_{n\cdot k}$ *can be generated by CD-grammar system with* $2\cdot n+1$ *context-free components, without erasing productions, working in the* $(t\wedge \leq k+1)$*-mode.*

Proof. We first construct a CD grammar system

$$G = (N, \{a, b\}, (S, 0), P_0, P_{1,1} \cup P_{1,2}, \ldots, P_{n,1} \cup P_{n,2})$$

working in the $(t\wedge = k+1)$-mode generating language $S_{n\cdot k}$. Let

$$\begin{aligned} N &= \{ (S, i) \mid 0 \leq i \leq n \} \\ &\cup \{ (q_i, 0), (q_i, 1) \mid 1 \leq i \leq n \} \\ &\cup \{ (t_i, j), (t_i', j) \mid 1 \leq i \leq n, 0 \leq j \leq k \} \\ &\cup \{ (A_i, 0), (A_i, 1) \mid 1 \leq i \leq k \cdot n \} \end{aligned}$$

and let the set of productions be as follows:

$$\begin{aligned} P_0 &= \{(S,0) \to (S,1), (S,k) \to (q_1,0)(A_1,0)b(A_2,0)b\ldots b(A_{n\cdot k},0)b\} \\ &\cup \{ (S,i) \to (S,i+1) \mid 1 \leq i < k \} \\ &\cup \{ (A_i,1) \to (A_i,0) \mid 1 \leq i \leq n\cdot k \} \\ &\cup \{ (q_i,1) \to (q_{i+1},0) \mid 1 \leq i < n \} \cup \{(q_n,1) \to (q_1,0), (q_n,1) \to (t_1,0)\} \\ &\cup \{ (t_i',j) \to (t_i',j+1) \mid 1 \leq i \leq n, 0 \leq j < k \} \\ &\cup \{ (t_i',k) \to (t_{i+1},0) \mid 1 \leq i < n \} \cup \{(t_n',k) \to b\}. \end{aligned}$$

For every $1 \leq i \leq n$ let $P_i = P_{i,1} \cup P_{i,2}$ where

$$\begin{aligned} P_{i,1} &= \{(q_i,0) \to (q_i,1)\} \\ &\cup \{ (A_j,0) \to a(A_j,1)a \mid (i-1)\cdot k \leq j \leq i\cdot k \} \quad \text{and} \\ P_{i,2} &= \{(t_i,0) \to (t_i',0)\} \\ &\cup \{ (A_j,0) \to aba \mid (i-1)\cdot k \leq j \leq i\cdot k \}. \end{aligned}$$

The only way to start the derivation is to use P_0 obtaining the word

$$(q_1,0)(A_1,0)b(A_2,0)b\ldots b(A_{n\cdot k},0)b.$$

To continue the derivation one always has to apply P_i and P_0 in sequence: P_i is only successfully applicable to a sentential form beginning with a letter $(q_i,0)$ or $(t_i,0)$, and P_0 is only successfully applicable to a sentential form beginning with a letter $(q_i,1)$ or $(t_i',0)$.

Assume we have a sentential form starting with letter $(q_i,0)$, then rules from P_i replace exactly k occurrences of nonterminals $(A_j,0)$, for $(i-1)\cdot k \leq j \leq i\cdot k$, by $a(A_j,1)a$ or aba, respectively, and the label is changed to $(q_i,1)$. Now applying the corresponding rules from P_0 (the only way to continue), the derivation would block if at least one of the symbols $(A_j,0)$ from the previous step were replaced by aba. This ensures that all previously used productions are non-terminating. In case the sentential from starts with a letter $(t_i,0)$, then an application of P_i followed by P_0 checks whether the terminating rules form P_i were all used or not.

Thus, starting with the word $(q_1,0)(A_1,0)b(A_2,0)b\ldots b(A_{n\cdot k},0)b$, one cycle, i.e., an application of $P_1,P_0,P_2,P_0,\ldots,P_n$, and P_0 leads to

$$(q_1,0)a(A_1,0)aba(A_2,0)ab\ldots ba(A_{n\cdot k},0)ab,$$

and in general, running the cycle ℓ times, to the word

$$(q_1,0)a^\ell(A_1,0)a^\ell ba^\ell(A_2,0)a^\ell b\ldots ba^\ell(A_{n\cdot k},0)a^\ell b.$$

Note that in the last application of P_0 we could have taken the rule $(q_n,1)\to(t_1,0)$ in order to terminate the derivation process after having finished the next cycle. After that cycle the grammar G has generated the word $b(a^{\ell+1}b)^{2n\cdot k}$.

In case of the $(t\wedge \leq k+1)$-mode, we use the same construction as above, but now treat each $P_{i,j}$, for $1\leq i\leq n$ and $1\leq j\leq 2$, as an independent component of the grammar. This gives the bound $2\cdot n+1$ on the number of components. □

Now we are ready to investigate the families $\mathscr{L}(\mathrm{CD}_n,\mathrm{CF}[-\lambda],(t\wedge\Delta k))$ in more detail. First, let the number of components n be fixed.

Corollary 9 *Let $\Delta\in\{\leq,=\}$ and $n\in\mathbb{N}$ be fixed. The hierarchy of the families of languages $\mathscr{L}_k := \mathscr{L}(\mathrm{CD}_n,\mathrm{CF}[-\lambda],(t\wedge\Delta k))$ with respect to $k\in\mathbb{N}$ is infinite, i.e., for every $k\in\mathbb{N}$ there exists an $m\in\mathbb{N}$, $m>k$, such that $\mathscr{L}_k\subset\mathscr{L}_m$.*

Proof. We first consider the $(t\wedge = k)$-mode. By the preceding theorem we have that

$$S_{n\cdot m}\in\mathscr{L}(\mathrm{CD}_{n+1},\mathrm{CF}[-\lambda],(t\wedge = m+1)).$$

On the other hand, $S_{n\cdot m}\notin\mathscr{L}_{n\cdot m-1}(\mathrm{P},\mathrm{CF})$ according to Theorem 7 and, because of

$$\mathscr{L}(\mathrm{CD}_{n+1},\mathrm{CF}[-\lambda],(t\wedge = k))\subseteq\mathscr{L}_{(n+1)\cdot k}(\mathrm{P},\mathrm{CF})$$

according to Theorem 6, $S_{n\cdot m}$ therefore cannot belong to $\mathscr{L}(\mathrm{CD}_{n+1},\mathrm{CF}[-\lambda],(t\wedge = k))$, provided $n\cdot m-1\geq(n+1)\cdot k$, i.e., $m\geq\frac{n+1}{n}k+\frac{1}{n}$.

For the $(t\wedge\leq k)$-mode, we can argue in a similar way: By the preceding theorem we have that

$$S_{n\cdot m}\in\mathscr{L}(\mathrm{CD}_{2n+1},\mathrm{CF}[-\lambda],(t\wedge\leq m+1))$$

and therefore $S_{n\cdot m} \in \mathscr{L}(\mathrm{CD}_{2n+2}, \mathrm{CF}[-\lambda], (t\wedge \leq m+1))$, too. On the other hand, $S_{n\cdot m} \notin \mathscr{L}_{n\cdot m-1}(\mathrm{P}, \mathrm{CF})$ according to Theorem 7 and, because of

$$\mathscr{L}(\mathrm{CD}_{2n+1}, \mathrm{CF}[-\lambda], (t\wedge \leq k)) \subseteq \mathscr{L}_{(2n+1)\cdot k}(\mathrm{P}, \mathrm{CF}) \subseteq \mathscr{L}_{(2n+2)\cdot k}(\mathrm{P}, \mathrm{CF})$$

according to Theorem 6, $S_{n\cdot m}$ therefore cannot belong to $\mathscr{L}(\mathrm{CD}_{2n+1}, \mathrm{CF}[-\lambda], (t\wedge \leq k)) \cap \mathscr{L}(\mathrm{CD}_{2n+2}, \mathrm{CF}[-\lambda], (t\wedge \leq k))$, provided $n\cdot m - 1 \geq (2n+2)\cdot k$, i.e., $m \geq 2\frac{n+1}{n}k + \frac{1}{n}$. □

We now consider the other hierarchy, i.e., we fix k and vary the number of components:

Corollary 10 *Let $\Delta \in \{\leq, =\}$ and $k \in \mathbb{N}$ be fixed. The hierarchy of the families of languages $\mathscr{L}_n := \mathscr{L}(\mathrm{CD}_n, \mathrm{CF}[-\lambda], (t\wedge \Delta k))$ with respect to $n \in \mathbb{N}$ is infinite, i.e., for every $n \in \mathbb{N}$ there exists an $m \in \mathbb{N}$, $m > n$, such that $\mathscr{L}_n \subset \mathscr{L}_m$.*

Proof. We argue in a similar way as in the preceding corollary. First consider the $(t\wedge = k)$-mode of derivation. We already know that

$$S_{m\cdot k} \in \mathscr{L}(\mathrm{CD}_{m+1}, \mathrm{CF}[-\lambda], (t\wedge = k{+}1)).$$

according to Theorem 7, but $S_{m\cdot k} \notin \mathscr{L}_{m\cdot k-1}(\mathrm{P}, \mathrm{CF})$ according to Theorem 7 and, because of

$$\mathscr{L}(\mathrm{CD}_n, \mathrm{CF}[-\lambda], (t\wedge = k+1)) \subseteq \mathscr{L}_{n\cdot(k+1)}(\mathrm{P}, \mathrm{CF})$$

according to Theorem 6, $S_{m\cdot k}$ therefore cannot belong to $\mathscr{L}(\mathrm{CD}_n, \mathrm{CF}[-\lambda], (t\wedge = k+1))$, provided $m\cdot k - 1 \geq n\cdot(k+1)$, i.e., $m \geq \frac{k+1}{k}n + \frac{1}{k}$.

By a similar reasoning, in case of the $(t\wedge \leq k)$-mode, for

$$S_{m\cdot k} \in \mathscr{L}(\mathrm{CD}_{2m+1}, \mathrm{CF}[-\lambda], (t\wedge \leq k{+}1)) \subseteq \mathscr{L}(\mathrm{CD}_{2m+2}, \mathrm{CF}[-\lambda], (t\wedge \leq k{+}1))$$

we obtain $S_{m\cdot k} \notin \mathscr{L}(\mathrm{CD}_{2n+1}, \mathrm{CF}[-\lambda], (t\wedge = k+1)) \cap \mathscr{L}(\mathrm{CD}_{2n+2}, \mathrm{CF}[-\lambda], (t\wedge = k+1))$, provided $m\cdot k - 1 \geq (2n+2)\cdot(k+1)$, i.e., $m \geq 2\frac{k+1}{k}n + \frac{2(k+1)}{k}$ $(= 2(1+\frac{1}{k})n + 2(1+\frac{1}{k}))$.

In both cases, we see that the hierarchy with respect to the number of components is infinite. □

Finally, let us consider the hierarchies for the "small cases" of n.

Lemma 11 *Let $k \in \mathbb{N}$ and $\Delta \in \{\leq, =\}$.*

$$\begin{aligned} \mathscr{L}(\mathrm{CD}_1, \mathrm{CF}[-\lambda], (t\wedge \Delta k)) &\subset \mathscr{L}(\mathrm{CD}_2, \mathrm{CF}[-\lambda], (t\wedge \Delta 1)) \\ &\subset \mathscr{L}(\mathrm{CD}_3, \mathrm{CF}[-\lambda], (t\wedge \Delta k)). \end{aligned}$$

Proof. By our previous considerations, we know that $\mathscr{L}(\mathrm{CD}_1, \mathrm{CF}[-\lambda], (t\wedge \Delta k))$ and $\mathscr{L}(\mathrm{CD}_2, \mathrm{CF}[-\lambda], (t\wedge \Delta 1))$ coincide with $\mathscr{L}(\mathrm{FIN})$ and $\mathscr{L}(\mathrm{LIN})$, respectively, which already proves the first strict inclusion.

Now consider the non-linear language $\{\, a^n b^n a^m b^m \mid n, m \in \mathbb{N}\,\}$, which is generated by a CD grammar system

$$G = (\{S, A, B, A', B'\}, \{a, b\}, S, P_1, P_2, P_3)$$

taking $k = 1$ with the following three components:

$$\begin{aligned} P_1 &= \{S \to AB, A' \to A, B' \to B\} \\ P_2 &= \{A \to aA'b, A \to ab, B' \to B'\} \\ P_3 &= \{B \to aB'b, B \to ab, A \to A, A' \to A'\}. \end{aligned}$$

First, P_1 and P_2 have to be applied in sequence, say n times, until P_2 uses the rule $A \to ab$. Now, P_3 can be applied. Then, P_1 and P_3 must be applied in sequence, say $m-1$ times, until P_3 terminates the whole derivation using $B \to ab$. In this way, a word $a^n b^n a^m b^m$ is derived. By the prolongation technique, the claimed assertion follows for $k > 1$. □

We conclude this section by remarking that the results presented in this section (originally contained in the Technical Report [7]) have been employed to show the following theorem in [1] that nicely complements our results here; we state these below with the notations of our paper.

Theorem 12 *Let $n,k \geq 1$. Then we have*

1. $\mathscr{L}(\mathrm{CD}_n, \mathrm{CF}[-\lambda], (t\wedge = k)) \subset \mathscr{L}(\mathrm{CD}_{n+2}, \mathrm{CF}[-\lambda], (t\wedge = k+1))$ *and*
2. $\mathscr{L}(\mathrm{CD}_n, \mathrm{CF}[-\lambda], (t\wedge \leq k)) \subset \mathscr{L}(\mathrm{CD}_{2\cdot(n+1)}, \mathrm{CF}[-\lambda], (t\wedge \leq k+1))$.

5 CD Grammar Systems and Programmed Grammars of Finite Index

In this section we consider the finite index property for CD and HCD grammar systems and how they relate to programmed grammars of finite index in more detail.

Theorem 13 *Let $m \in \mathbb{N}$, $FI = \{t\} \cup \{= m'', \geq m'', (\geq m'' \wedge \leq m'), (t\wedge = k), (t\wedge \leq k), (t\wedge \geq k) \mid m', m'', k \in \mathbb{N}, m' \geq m'' \geq m\}$, and F contain all the hybrid modes considered in this paper, i.e., $F = \{(t\wedge = k), (t\wedge \leq k) \mid k \in \mathbb{N}\}$. Let $f \in FI$. Then*

$$\begin{aligned} \mathscr{L}_m(\mathrm{P}, \mathrm{CF}[-\lambda]) &= \mathscr{L}_m(\mathrm{CD}_\infty, \mathrm{CF}[-\lambda], f) \\ &= \mathscr{L}_m(\mathrm{HCD}_\infty, \mathrm{CF}[-\lambda], F). \end{aligned}$$

Proof. Looking through all the proofs showing the containment of HCD languages within programmed languages with appearance checking, it is easily seen that all these constructions preserve a finite index restriction.

Hence, it only remains to show that

$$\mathscr{L}_m(\mathrm{P}, \mathrm{CF}[-\lambda]) \subseteq \mathscr{L}_m(\mathrm{CD}_\infty, \mathrm{CF}[-\lambda], f)$$

for every $f \in FI$. Let $L \in \mathscr{L}_m(\mathrm{P}, \mathrm{CF})$ be generated by a programmed grammar $G = (N, T, P, S, \Lambda, \sigma)$ in NSF. Especially, there exists a function $f : \Lambda \to \mathbb{N}_0^N$ such that, if $S \overset{*}{\Rightarrow} v \underset{p}{\Rightarrow} w$ is a derivation in G, then $(f(p))(A) = |v|_A \leq 1$ for every nonterminal A.

We construct a simulating CD grammar system G' given by

$$((N \times \Lambda) \cup (\{ i \in \mathbb{N} \mid i \leq m \} \times N), T, (1, S), \{P_I\} \cup \{ P_{p,q} \mid p \in \Lambda \wedge q \in \sigma(p) \})$$

of index m all of whose components are working in one of the modes $= m$, $\geq m$, t, $(\geq m, \leq m')$ (with $m' \geq m$), $(t\wedge = m)$, $(t\wedge \leq m)$, or $(t\wedge \geq m)$. Consider a production $\Lambda(p) = A \to w$ of G. We can assume (check) that $(f(p))(A) = 1$. Furthermore, define

$$n := \sum_{B \in N} (f(p))(B) \leq m$$

is the number of nonterminals in the current string (within a possible derivation leading to an application of p).

Let the homomorphism $h_{p,q} : (N \times \{p\} \cup T)^* \to (N \times \{q\} \cup T)^*$ be defined by $(A,p) \mapsto (A,q)$ for $A \in N$, $a \mapsto a$ for $a \in T$.

For every $q \in \sigma$, we introduce a component $P_{p,q}$ within the CD grammar system containing the following productions:

If $n = m$, then $(A,p) \to h_{p,q}(w)$ simulates the (successful) application of rule p. If $n < m$, then we prolong the derivation in the following way:

$$(A,p) \to (1,A), \quad (1,A) \to (2,A), \quad \ldots, \quad (m-n,A) \to h_{p,q}(w).$$

$(B,p) \to h_{p,q}(B)$ for $B \in N \setminus \{A\}$ keeps track of the information of the current state.

As initialization component, we take P_I containing

$$(1,S) \to (2,S), \quad \ldots, \quad (m,S) \to (S,p)$$

for every $p \in \Lambda$ such that $(f(p))(S) = 1$.

Observe that, by induction, to every sentential form derivable from the initial symbol $(1,S)$, any component applied to it can make either 0 steps (so, we selected the wrong one) or exactly m steps.

By a simple prolongation trick, we can also take components working in one of the modes $= m''$, $\geq m''$, $(\geq m'', \leq m')$ (with $m' \geq m''$), for some $m'' \geq m$.

Since the construction given in Theorem 2 is index-preserving, we can also take arbitrary $(t\wedge = k)$ or $(t\wedge \leq k)$ components instead of requiring $k \geq m$. Since the t-mode and the $(t\wedge \geq 1)$-mode are identical, the prolongation technique delivers the result for the $(t\wedge \geq k)$-mode for $k \in \mathbb{N}$ in general. □

Our theorem readily implies a characterization of programmed languages of general finite index. We summarize this fact together with results obtained *via* a different simulation in [3, Theorem 3.26] in the following corollary.

Corollary 14 *Let $m,k,k' \in \mathbb{N}$, $k \geq 2$, and $\Delta \in \{\leq,=,\geq\}$. Then following families of languages coincide with $\mathscr{L}_{fin}(\mathrm{P},\mathrm{CF}[-\lambda])$:*

1. $\mathscr{L}_{fin}(\mathrm{CD}_\infty,\mathrm{CF}[-\lambda],t)$,
2. $\mathscr{L}_{fin}(\mathrm{CD}_\infty,\mathrm{CF}[-\lambda],=k)$,
3. $\mathscr{L}_{fin}(\mathrm{CD}_\infty,\mathrm{CF}[-\lambda],\geq k)$,
4. $\mathscr{L}(\mathrm{CD}_\infty,\mathrm{CF}[-\lambda],(t\wedge = k))$,
5. $\mathscr{L}(\mathrm{CD}_\infty,\mathrm{CF}[-\lambda],(t\wedge \leq k))$, *and*
6. $\mathscr{L}_{fin}(\mathrm{CD}_\infty,\mathrm{CF}[-\lambda],(t \wedge \Delta k'))$.

As regards the number of components, we can state the following:

Theorem 15 *Let $m \in \mathbb{N}$, $\Delta \in \{\leq,=,\geq\}$. Then we have*

1. $\mathscr{L}_m(\mathrm{CD}_\infty,\mathrm{CF}[-\lambda],\Delta m) = \mathscr{L}_m(\mathrm{CD}_3,\mathrm{CF}[-\lambda],\Delta m)$ *and*
2. $\mathscr{L}_m(\mathrm{CD}_\infty,\mathrm{CF}[-\lambda],(t \wedge \Delta m)) = \mathscr{L}_m(\mathrm{CD}_3,\mathrm{CF}[-\lambda],(t \wedge \Delta m))$.

Proof. Since we restrict our attention to languages of index m, we can simply carry over the proofs of the type "three is enough" for t-mode components, see [3, Theorem 3.10] and [13, Lemma 2].

In case of the $(t\wedge = m)$-mode, we have to go back to the simulation of the programmed grammar given in the preceding theorem. It is clear that, due to the nonterminal separation form (NSF) (see Theorem 1), we can prolong the simulation of a single nonterminal symbol. □

It is quite natural to compare the families of languages defined by CD grammar systems obtained *via* the restriction of being of finite index m with their unrestricted counterparts. In our case, it is interesting to see that also these unrestricted counterparts deliver languages of finite index. However, as we have seen in Theorem 8,

$$S_{n\cdot k} \in \mathscr{L}(\mathrm{CD}_{n+1}, \mathrm{CF}[-\lambda], (t\wedge = k+1)).$$

Especially, we have

$$S_{2\cdot(m-1)} \in \mathscr{L}(\mathrm{CD}_3, \mathrm{CF}[-\lambda], (t\wedge = m)).$$

Since $S_{2\cdot(m-1)}$ is not of (programmed) index $2m-3$, we can state:

Corollary 16 *For $m \in \mathbb{N}$, we have*

$$\mathscr{L}_m(\mathrm{CD}_3, \mathrm{CF}[-\lambda], (t\wedge = m)) \subset \mathscr{L}(\mathrm{CD}_3, \mathrm{CF}[-\lambda], (t\wedge = m)).$$

Proof. Our previous considerations deliver the case $m > 2$, since $S_{2(m-1)}$ is not a (programmed) language of index $2m-3$, and therefore not a (programmed) language of index m; hence,

$$S_{2\cdot(m-1)} \notin \mathscr{L}_m(\mathrm{CD}_3, \mathrm{CF}[-\lambda], (t\wedge = m)).$$

In case $m = 2$, we know that

$$S_3 \notin \mathscr{L}_2(\mathrm{CD}_3, \mathrm{CF}[-\lambda], (t\wedge = 2)).$$

On the other hand, the $(\mathrm{CD}_3, \mathrm{CF}[-\lambda], (t\wedge = 2))$ grammar system

$$G = (\{S, A, B, A', B'C, C', B'', F\}, \{a, b\}, S, P_1, P_2, P_3)$$

with the following three components generates S_3, starting with S:

$$\begin{aligned} P_1 &= \{A \to aA'a, A \to aba, B \to aB'a, B \to B''\} \\ P_2 &= \{B' \to B, C \to aC'a, A \to F, B'' \to aba, C \to aba\} \\ P_3 &= \{S \to bAbBbCb, C' \to C, A' \to A, B' \to F\}. \end{aligned}$$

Finally, we have $\mathscr{L}_1(\mathrm{CD}_3, \mathrm{CF}[-\lambda], (t\wedge = 1)) = \mathscr{L}(\mathrm{LIN})$. The example $\{a^n b^n a^m b^m \mid n, m \in \mathbb{N}\} \notin \mathscr{L}(\mathrm{LIN})$ in Lemma 11 was shown to be in $\mathscr{L}(\mathrm{CD}_3, \mathrm{CF}[-\lambda], (t\wedge = 1))$. □

If we admit four components (or arbitrarily many), by a similar reasoning we can separate all corresponding classes, since $3(m-1) = 3m-3 > m$ for $m \geq 2$.

Corollary 17 *For $m, n \in \mathbb{N}$, $n \geq 4$ (or $n = \infty$), we have*

$$\mathscr{L}_m(\mathrm{CD}_n, \mathrm{CF}[-\lambda], (t\wedge = m)) \subset \mathscr{L}(\mathrm{CD}_n, \mathrm{CF}[-\lambda], (t\wedge = m)).$$

Observe that the results exhibited in the last two corollaries are quite astonishing if one keeps in mind that

$$\begin{aligned} \mathscr{L}_{\mathit{fin}}(\mathrm{P}, \mathrm{CF}) &= \bigcup_{m \in \mathbb{N}} \mathscr{L}_m(\mathrm{CD}_n, \mathrm{CF}[-\lambda], (t\wedge = m)) \\ &= \bigcup_{m \in \mathbb{N}} \mathscr{L}(\mathrm{CD}_n, \mathrm{CF}[-\lambda], (t\wedge = m)) \end{aligned}$$

for all $n \in \mathbb{N}, n > 2$.

6 Conclusions and Prospects

In this paper we have studied CD grammar systems working in the internally hybrid modes $(t\wedge = m)$ and $(t\wedge \leq m)$ together with the finite index restriction. Showing specific relations to programmed grammars of finite index, we were able to establish infinite hierarchies for CD grammar systems of finite index working in the internally hybrid modes $(t\wedge = k)$ and $(t\wedge \leq k)$ both with respect to the number of components n and the number of maximal steps k. However, many quite natural questions still remain open. For instance, Theorem 12 leaves open the strictness of several natural inclusion relations relating the parameters "numbers of components" n and "step number bound" k.

It is well-known that ET0L systems are tightly related to CD grammar systems working in the t-mode. In the literature, several step-bound restrictions have been discussed in relation with parallel systems, see [5] for an overview. Are these somehow related to the internally hybrid systems discussed in this paper (and their companions)? Or do hybrid modes lead to new (natural) derivation modes for parallel systems? In particular, the finite index restriction studied in Section 5 could be of interest in this context, although (and also because) we are not aware of a study of finite index in the context of limited parallel rewriting, which might be an interesting research question in its own.

Acknowledgements Most of the research of the first and last author was undertaken while being affiliated to Wilhelm-Schickard Institut für Informatik, Universität Tübingen, Sand 13, D-72076 Tübingen, Germany. Part of the research of the first author was supported by Deutsche Forschungsgemeinschaft, grant DFG La 618/3-1/2 "Komplexitätstheoretische Methoden für die adäquate Modellierung paralleler Berechnungen."

References

[1] H. Bordihn & M. Holzer (1999): *On a hierarchy of languages generated by cooperating distributed grammar systems*. Information Processing Letters 69(2), pp. 59–62, doi:10.1016/S0020-0190(98)00200-2.

[2] E. Csuhaj-Varjú & J. Dassow (1990): *On cooperating/distributed grammar systems*. J. Inf. Process. Cybern. EIK (formerly Elektron. Inf.verarb. Kybern. 26(1/2), pp. 49–63.

[3] E. Csuhaj-Varjù, J. Dassow, J. Kelemen & Gh. Păun (1994): *Grammar Systems. A Grammatical Approach to Distribution and Cooperation*. Gordon and Breach, London.

[4] J. Dassow & Gh. Păun (1989): *Regulated Rewriting in Formal Language Theory*. EATCS Monographs in Theoretical Computer Science 18, Springer Berlin, doi:10.1007/978-3-642-74932-2.

[5] H. Fernau (2003): *Parallel Grammars: A Phenomenology*. GRAMMARS 6, pp. 25–87, doi:10.1023/A:1024087118762.

[6] H. Fernau & R. Freund (1996): *Bounded parallelism in array grammars used for character recognition*. In P. Perner, P. Wang & A. Rosenfeld, editors: Advances in Structural and Syntactical Pattern Recognition (Proceedings of the SSPR'96), Lecture Notes in Computer Science 1121, Springer, pp. 40–49, doi:10.1007/3-540-61577-6_5.

[7] H. Fernau, R. Freund & M. Holzer (1996): *External versus Internal Hybridization for Cooperating Distributed Grammar Systems*. Technical Report TR 185–2/FR–1/96, Technische Universität Wien (Austria).

[8] H. Fernau, R. Freund & M. Holzer (2001): *Hybrid modes in cooperating distributed grammar systems: internal versus external hybridization*. Theoretical Computer Science 259(1–2), pp. 405–426, doi:10.1016/S0304-3975(00)00022-0.

[9] H. Fernau, R. Freund & M. Holzer (2003): *Hybrid modes in cooperating distributed grammar systems: combining the t-mode with the modes $\leq k$ and $= k$.* Theoretical Computer Science 299, pp. 633–662, doi:10.1016/S0304-3975(02)00541-8.

[10] H. Fernau & M. Holzer (1996): *Accepting multi-agent systems II.* Acta Cybernetica 12, pp. 361–379.

[11] H. Fernau & M. Holzer (1997): *Conditional context-free languages of finite index*. In Gh. Păun & A. Salomaa, editors: New Trends in Formal Languages, Lecture Notes in Computer Science 1218, Springer, pp. 10–26, doi:10.1007/3-540-62844-4_2.

[12] R. Meersman & G. Rozenberg (1978): *Cooperating grammar systems*. In: Proceedings of Mathematical Foundations of Computer Science MFCS'78, Lecture Notes in Computer Science 64, Springer, pp. 364–374, doi:10.1007/3-540-08921-7_84.

[13] V. Mitrana (1993): *Hybrid cooperating/distributed grammar systems*. Computers and Artificial Intelligence 12(1), pp. 83–88.

[14] Gh. Păun (1994): *On the generative capacity of hybrid CD grammar systems*. J. Inf. Process. Cybern. EIK (formerly Elektron. Inf.verarb. Kybern.) 30(4), pp. 231–244.

[15] G. Rozenberg & A. Salomaa, editors (1997): *Handbook of Formal Languages, 3 volumes*. Springer, Berlin.

[16] G. Rozenberg & D. Vermeir (1978): *On the effect of the finite index restriction on several families of grammars; Part 2: context dependent systems and grammars*. Foundations of Control Engineering 3(3), pp. 126–142.

Representations of Circular Words

László Hegedüs* Benedek Nagy* †

{hegedus.laszlo, nbenedek}@inf.unideb.hu

*Department of Computer Science,
Faculty of Informatics, University of Debrecen

†Department of Mathematics, Faculty of Arts and Sciences,
Eastern Mediterranean University, Famagusta, North Cyprus, Mersin-10, Turkey

In this article we give two different ways of representations of circular words. Representations with tuples are intended as a compact notation, while representations with trees give a way to easily process all conjugates of a word. The latter form can also be used as a graphical representation of periodic properties of finite (in some cases, infinite) words. We also define iterative representations which can be seen as an encoding utilizing the flexible properties of circular words. Every word over the two letter alphabet can be constructed starting from *ab* by applying the fractional power and the cyclic shift operators one after the other, iteratively.

1 Introduction

One of the most popular areas of research in theoretical computer science is combinatorics on words. This field deals with various properties of finite and infinite sequences or words. Being closely related to mathematics, it has connections to algebra, number theory, game theory and several others. Although it was written decades ago, the books of M. Lothaire are good reads and are recommended for researchers who want to get a deep overview of the subject [8, 9, 10]. Axel Thue contributed the first results to the field [19, 20]. Since then many applications in computer science have been discovered (e.g., in string matching, data compression, bioinformatics, etc.).

We deal with circular words (sometimes called necklaces [18] or cyclic words) that are different from linear ones and lead to some interesting new viewpoints. Similar sequences can appear in nature, for example, the DNA sequences of some bacteria has a similar form to a necklace. In the simplest sense, circular words are strongly periodic discrete functions.

Circular words are not as widely investigated as linear words. We hope that our approach and results may show that interesting facts can be obtained by analyzing these sequences. Dirk Nowotka wrote about unbordered conjugates of words in Chapter 4 of his dissertation [13]. Complementing this, we deal with bordered conjugates that have periods smaller than the length of the word. Another related article is [4], where permutations and cyclic permutations of primitive and non-primitive words were investigated. For an overview of current research about circular words, the reader can consult the following articles. Relations to Weinbaum factorizations are investigated in [3]. Several articles were written about pattern avoidance of circular words, for example, [2, 6, 17] to name a few. Other applications in mathematics, namely integer sequences [14, 15] were also considered.

The notion of weak and strong periods was introduced in [7]. One result about periodic functions is often cited in combinatorics on words, since it is clearly about periodic infinite words too. This result belongs to Fine and Wilf [5]. It can be shown by example that this statement is not true for weak periods of circular words [7]. In this paper, we investigate two kinds of representations of circular words continuing the research line of the paper [7] presented at the WORDS 2013 conference in Turku. The first one is connected to the property that every linear word has a shortest root, while the other one is related to tries (see e.g., [18]).

The structure of the paper is as follows. Section 2 defines the notation and notions used in the rest of the article. After this, in Section 3 we discuss ways of representing circular words with tuples and an algorithm to construct one of these representations. Section 4 is about representing circular words with trees (or tries) and we present some results related to Fibonacci words. At the end in Section 5 some possible directions of future research is discussed.

2 Preliminaries

The following notions and notation are used in the rest of the article. We will call a non-empty set of symbols an *alphabet* and denote it by Σ. *Words* (or *linear words*) over Σ are finite sequences of symbols of Σ. The operation of concatenation is defined by writing two words after each-other. The *empty word*, i.e., the empty sequence is denoted by ε and it is the unit element of the monoid Σ^*. We also define $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$. The *length* of the word $w \in \Sigma^*$ (denoted by $|w|$) is the length of w as a sequence, that is, the number of all the symbols in w. We will use $\mathbb{N}$ to denote the set of non-negative integers.

We say, that $v \in \Sigma^*$ is a *factor* of $w \in \Sigma^*$ if there exist words $x, y \in \Sigma^*$ such that $w = xvy$. Furthermore, if $x = \varepsilon$ (resp. $y = \varepsilon$), then v is a *prefix* (resp. *suffix*) of w. For any word w and integer $0 \leq k \leq |w|$, we denote the *length k factors* of w by $\mathscr{F}_k(w)$. For arbitrary positive integers p and q, we use $(p \bmod q)$ to denote the remainder of $\frac{p}{q}$. Let $w \in \Sigma^*$ be a word of length n, that is, $w = w_1 \dots w_n$, where $w_1, \dots, w_n \in \Sigma$. Then for any $p \in \mathbb{N}$, we have $w^{\frac{p}{n}} = w^{\lfloor \frac{p}{n} \rfloor} w'$, where $w' = w_1 \dots w_{(p \bmod n)}$. We call $w^{\frac{p}{n}}$ the *fractional power* of w. From now on we will always refer to the *i*th position of a word $w \in \Sigma^*$ as w_i. A word $w \in \Sigma^+$ is *primitive* if there is no word $v \in \Sigma^*$ such that $w = v^p$ where $p \in \mathbb{N}$, $p > 1$.

A positive integer p is a *period* of $w = w_1 \dots w_n$ if $w_i = w_{i+p}$ for all $i = 1, \dots, n-p$. As a complementary notion, word $v \in \Sigma^*$ is a *border* of $w \in \Sigma^*$ if v is a prefix and also a suffix of w. Each word $w \in \Sigma^*$ has *trivial borders* ε and w. It is clear, that word w has a border b if and only if w has period $|w| - |b|$.

Words x and y are *conjugates* if there exist words $u, v \in \Sigma^*$ such that $x = uv$ and $y = vu$. Related to this notion, we define the *shift* operation $\sigma(w)$ for all $w \in \Sigma^*$ as follows:

$$\sigma(w) = w_2 \dots w_n w_1.$$

Moreover, $\sigma^{\ell}(w) = \sigma^{\ell-1}(\sigma(w)) = w_{1+\ell} \dots w_n w_1 \dots w_{\ell}$. Also, we will use $\sigma^{-\ell}(w)$ that can also be written as $\sigma^{|w|-\ell}(w)$.

Lyndon and Schützenberger stated the following, which characterizes the relation between a word and its non-trivial borders [12].

Lemma 1 (Lyndon and Schützenberger)**.** *Let $x \in \Sigma^+$, y, $b \in \Sigma^*$ be arbitrary words. Then $xb = by$ if and only if there exist $u \in \Sigma^+$, $v \in \Sigma^*$ and $k \in \mathbb{N}$ such that $x = uv$, $y = vu$ and $b = (uv)^k u = u(vu)^k$.*

A *circular word* is obtained from a linear word $w \in \Sigma^*$ if we link its first symbol after the last one, as seen on Figure 1.

One can see from the figure that circular words do not have a beginning nor an end. Nor do the notions of suffix and prefix make sense. A circular word $w_\circ$ can be seen as the set of all conjugates of w, or all cyclic shifts of w, that is, the set

$$w_\circ = \{v \mid v \text{ is a conjugate of } w\} = \{\sigma^{\ell}(w) \mid \ell = 0, \dots, |w| - 1\}.$$

$$\begin{tikzcd}
 & & & w_1 \arrow[r] & w_2 \arrow[dr] & & & \\
w = w_1w_2w_3\dots w_{n-1}w_n & \Longrightarrow & w_n \arrow[ur] & & & w_3 \arrow[dl] & = & w_\circ \\
 & & & w_{n-1} \arrow[ul] & \cdots \arrow[l] & & &
\end{tikzcd}$$

Figure 1: Creating the circular word $w_\circ$ from the linear word w.

Note, that $w_\circ$ consists exactly of the length $|w|$ factors of ww. That is, $w_\circ = \mathscr{F}_{|w|}(ww)$. The notions of weak- and strong periods were given in [7]. We will only refer to weak periods in this paper and define them as follows.

Definition 1. The positive integer p is a *weak* (*strong*) *period* of a circular word $w_\circ$ if p is a period of at least one (all) of the conjugates $v \in w_\circ$.

3 Representations with tuples

If not stated otherwise, we assume that alphabet Σ can be arbitrary. Every word $w \in \Sigma^*$ can be represented by a power of a (possibly shorter) word $u \in \Sigma^*$ and a positive integer that is the length of w. In other words, for all $w \in \Sigma^*$, there exists a word $u \in \Sigma^*$ such that $u^{\frac{|w|}{|u|}} = w$. We will call such a u a *root* of w, while the shortest root is called the *primitive root* of w (see e.g., pages 10–11 of [18]). In this section we discuss analogous representations of circular words that take advantage of their lack of strictly specified endpoints.

Definition 2. A pair $(u,n) \in \Sigma^* \times \mathbb{N}$ is a *representation* of the circular word $w_\circ$ over Σ if $|u| \le n$, $n = |w_\circ|$ and $u^{\frac{n}{|u|}} \in w_\circ$.

Definition 3. A *minimal representation* of a circular word $w_\circ$ over Σ is a representation (u,n) of $w_\circ$, such that $|u| \le |u'|$ for any other representation (u',n) of $w_\circ$.

It is clear, that every circular word has a minimal representation, since all of them have a smallest weak period. Trivially, that not all pairs (u,n) are minimal representations of some circular word. For example, consider the representation $(baa,5)$ of the circular word $(baaba)_\circ$. This circular word also has a representation $(ab,5)$ which is in fact a minimal representation.

It is also true, that a circular word can have more than one minimal representations. For example, $(ababa,12)$, $(babaa,12)$, $(abaab,12)$ and $(baaba,12)$ are all minimal representations of the circular word $(ababaababaab)_\circ$. Note, that $(aabab,12)$ is not a minimal representation of this circular word, since it represents $(aababaababaa)_\circ$.

Clearly, if $n = k \cdot |u|$ for some $k \in \mathbb{N}$ in a minimal representation (u,n), then $(\sigma^\ell(u),n)$ is also a minimal representation of the same circular word for all $\ell = 0,\dots,|u|-1$.

Suppose, that $w = u^m u'$ for some $u \in \Sigma^*$ where u' is a non empty prefix of u and $m \in \mathbb{N} \setminus \{0\}$. Then for every $k \in \mathbb{N}$, the word $w' = wu^k$ has a cyclic shift $\sigma^{|w|}(w') = u^{k+m}u'$. Thus the circular word $w'_\circ$ has a representation $(u, |w| + k \cdot |u|)$.

Theorem 1. *Let (u,n) be a representation of $w_\circ$. Suppose, that u has border s, that is, $u = sx = ys$, and $n = 2\cdot|u| - |s|$. Then (y,n) is also a representation of $w_\circ$. Moreover, if s is the longest non-trivial border of u, then (y,n) is a minimal representation of $w_\circ$.*

Proof. Let us have a representation (u,n) of $w_\circ$ that satisfies the assumption, that is, u has border s and $n = 2\cdot|u| - |s|$. Then u is in the form $u = sx = ys$ for some $x,y \in \Sigma^*$ and $w_\circ = (uy)_\circ = (ysy)_\circ$. By Lemma 1, yys has period $|y|$, thus $w_\circ$ has weak period $|y|$ and a representation (y,n).

If s is the longest non-trivial border of u, then y is the primitive root of u, thus (y,n) is a minimal representation of $w_\circ$. □

Suppose that we have a representation $w_\circ = (u,n)$, where $u \in \Sigma^*$ and $n \in \mathbb{N}$. If $|u| \geq 2$, then u may be compressed further. In other words, we can take a minimal representation $(u', |u|)$ with an additional parameter $k \in \mathbb{N}$, such that $\sigma^k(u)$ has primitive root u'. This method of compression can be done finitely many times, until reaching a word u_0 which we will refer to as a *minimal root* of $w_\circ$. We will call these representations *iterative representations*, defined formally in Definition 4. Of course, if a minimal root of a word $w_\circ$ has only one letter, then it is in the form $(a^{|w|})_\circ$ for some $a \in \Sigma$. In this case, this letter is unique and we can refer to it as the minimal root of $w_\circ$. Thus words in these forms have trivial representations and we will no longer deal with them.

Definition 4. Let $u \in \Sigma^*$, $m \in \mathbb{N}\setminus\{0\}$ and $\ell_1,\ell_2,\ldots,\ell_{m-1},\ell_m,k_1,k_2,\ldots,k_{m-1} \in \mathbb{N}$. The $2m$-tuple

$$(u,\ell_1,k_1,\ell_2,k_2,\ldots,\ell_{m-1},k_{m-1},\ell_m)$$

is an *iterative representation* of the circular word $w_\circ = (u_{m-1}^{\frac{\ell_m}{\ell_{m-1}}})_\circ$ over the two letter alphabet $\{a,b\}$, where $u_0 = u$, $u_1 = \sigma^{k_1}(u_0^{\frac{\ell_1}{|u_0|}})$ and $u_i = \sigma^{k_i}(u_{i-1}^{\frac{\ell_i}{\ell_{i-1}}})$ for all $i = 2,\ldots,m-1$.

Example 1. Consider the circular word $w_\circ = (bababaabbabaab)_\circ$. One of its iterative representations is

$$(baa,4,0,6,4,14).$$

By using the previous definition of the words u_i, the following words are obtained during the reconstruction of the circular word: $u_0 = baa$, $u_1 = baab$, $u_2 = babaab$ and finally, $w_\circ = (babaabbabaabba)_\circ$. Note, that no shifting is required in the last step, because $w_\circ = v_\circ$ for all $v \in w_\circ$.

Of course, every circular word has an iterative representation of the form above that can be constructed with the greedy algorithm in Figure 2. Moreover, the algorithm halts if only if it has found a minimal root.

Note, that by using this algorithm, we can process the iterative representation in Example 1 further to obtain $(ab,3,1,4,0,6,4,14)$. In fact, the following can be stated about the iterative representations of circular words over the two letter alphabet $\{a,b\}$.

Theorem 2. *Let $w \in \{a,b\}^*$. If $(u,\ell_1,k_1,\ldots,\ell_{m-1},k_{m-1},|w|)$ is a minimal iterative representation of $w_\circ$, then $|u| \leq 2$.*

Proof. It follows from the fact that every word $u \in \{a,b\}$, $|u| \geq 3$ has a conjugate that has a border of length at least one, thus in this case $u_\circ$ has a representation $(v,|u|)$ such that $|v| < |u|$. □

Let $(u,\ell_1,k_1,\ldots,\ell_{m-1},k_{m-1},|w|)$ be an iterative representation of $w_\circ$. It is *optimal* if for all iterative representations $(u',\ell'_1,k'_1,\ldots,\ell'_{m'-1},k'_{m'-1},|w|)$ of $w_\circ$, $|u| \leq |u'|$ and if $|u| = |u'|$, then $m \leq m'$. In other words an optimal iterative representation of $w_\circ$ is one with the shortest possible minimal root, such that

```
construct_iterative_representation($w_\circ$)
 1. $u \leftarrow w$
 2. $v \leftarrow$ find $v$ such that $(v,|w|)$ is a minimal representation of $w_\circ$
 3. $rep \leftarrow [|w|]$                                    # rep is a vector of integers
 4. while true do
 5.   $u \leftarrow v$
 6.   $v \leftarrow$ find $v$ such that $(v,|u|)$ is a minimal representation of $u_\circ$
 7.   if $|u| = |v|$ then                         # if we have found a minimal root,
 8.     break                                     # then the algorithm breaks the loop
 9.   endif
10.   $k \leftarrow$ find $k$ such that $\sigma^{-k}(u)$ has root $v$
11.   $rep \leftarrow |u| : k : rep$               # append $|u|$ and $k$ to $rep$ from the left
12. endwhile
13. return $v : rep$
```

Figure 2: Algorithm for constructing the iterative representation of $w_\circ$.

$w_\circ$ can be reconstructed from it with the least amount of fractional power operations (regardless of the amount of shift operations required).

The algorithm may not provide an optimal solution for all inputs $w_\circ$. For example, consider the circular word $(ababaa)_\circ$. The algorithm would construct the iterative representation $(ab,3,0,4,0,6)$, while an optimal solution would be $(ab,5,0,6)$. One of the directions of future research is to look for an efficient algorithm that always finds an optimal iterative representation of any circular word $w_\circ$ (see Section 5).

Note, that we do not have to restrict ourselves to representations of circular words. If we are looking for a linear word, another shift operation has to be applied at the end of the reconstruction.

Let us now turn to another method of representation, which is not intended as an encoding, nor as a compression, but a way of representing the structure of different conjugates of a word and their relation to each-other (e.g., common prefixes).

4 Representations with trees

The tree τ is the tree of the circular word $w_\circ$ if and only if for any word $v = v_1 \dots v_n$ in $w_\circ$, there exists a path in τ between the root and a leaf node with a series of edges labeled $v_1, \dots, v_n$.

This approach is related to tries that are data structures representing associative structures. They are often used to search for suffixes or other factors of words. Quite similarly, our trees represent a set of words that are conjugates of each-other. For more information on the use of tries consult [1].

We remark, that in our figures the letters appear as nodes, but they are to be considered as labels of edges between two (unnamed) nodes. This way, the represented words can be seen more clearly. First,

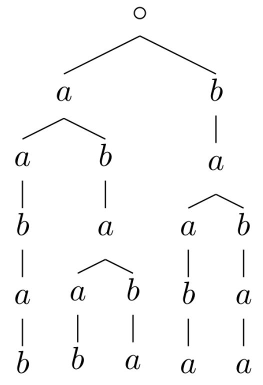


Figure 3: Tree representation of $(abaab)_\circ$.

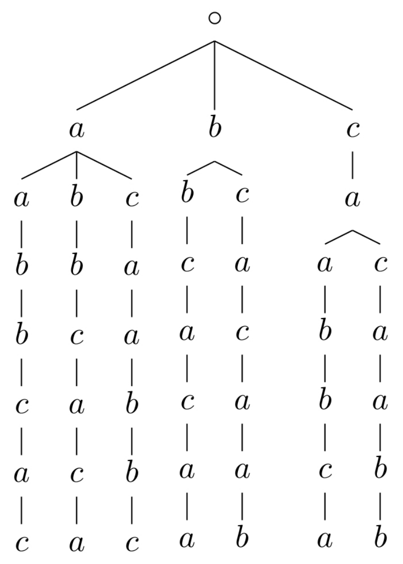


Figure 4: Tree representation of $(aabbcac)_\circ$.

consider the circular word

$$(abaab)_\circ = \{abaab, baaba, aabab, ababa, babaa\}.$$

Its tree representation is shown in Figure 3.

Now, see Figure 4 for the tree of the circular word $(aabbcac)_\circ$ (over the three letter alphabet $\{a,b,c\}$) which is the set

$$(aabbcac)_\circ = \{aabbcac, abbcaca, bbcacaa, bcacaab, cacaabb, acaabbc, caabbca\}.$$

Clearly, both trees represent finite-state automata with partially defined, deterministic transition functions. We can distinguish different levels of a tree. Vertex $\circ$ is on level zero ($\ell(\circ) = 0$) and if there is an edge $u \rightarrow v$, then $\ell(v) = \ell(u) + 1$.

We can see some branching nodes in both trees. The tree in Figure 4 has two branching nodes on level one while no two branching nodes of the tree in Figure 3 are on the same level.

Examining branching nodes is useful for analyzing trees of circular words and the words themselves. Suppose that tree τ has $u_1, \ldots, u_k$ branching nodes such that $\circ \rightarrow^a u_1$ and $u_i \rightarrow^a u_{i+1}$ for all $i = 1, \ldots, k-1$. Then there is a letter b such that a^k, $a^{k-1}b$, and thus $a^{k-2}b, \ldots, ab$, b are all factors of $w_\circ$. If the level of the leaf nodes is $k+1$, then the represented circular word must be $(a^k b)_\circ$. Similarly, if there are branching nodes $u_1, \ldots, u_m$ and $v_1, \ldots, v_k$ such that $\circ \rightarrow^a u_1 \rightarrow^a \ldots \rightarrow^a u_m$ and $\circ \rightarrow^b v_1 \rightarrow^b \ldots \rightarrow^b v_k$, and the level of the leaf nodes is $m+k$, then the tree can only represent the circular word $(a^m b^k)_\circ$. Apart from these simple cases, we can state the following about the relation of circular words and branching nodes in their trees: Let $w_\circ$ be a circular word with tree τ. There is a branching node in τ on level ℓ if and only if there are two distinct words $w', w'' \in w_\circ$, such that the longest common prefix of w' and w'' is a word of length ℓ. Moreover, if there is a branching node in the tree on level $n > 0$, then there is a branching node on level $n-1$. These nodes do not necessarily lie on the same path. To verify this, assume that tree τ contains the edges $u \rightarrow^a v$ and $u \rightarrow^b s$, where $u \neq \circ$. Then there are words $xay, xbz \in w_\circ$ such that $x, y, z \in \Sigma^*$ with $|x| > 0$, and $a, b \in \Sigma$, where Σ is an alphabet of at least two letters. Write $x = x_1, \ldots, x_m$. Clearly, both

$x_2 \dots x_m a y x_1$ and $x_2 \dots x_m b z x_1$ are in $w_\circ$, having a common prefix of length $|x|-1$. Thus there must be a node u' such that the path from $\circ$ to u' reads $x_2 \dots x_m$ and two nodes v' and s', such that $u' \to^a v'$ and $u' \to^b s'$.

Proposition 1. *Consider a circular word $w_\circ \in \{a,b\}$ with tree τ. If τ has a branching node on level $|w|-2$, then there is exactly one branching node on all levels $m = 0, \dots, |w|-2$ of τ.*

Proof. From the previous argument, it follows that all levels $k < |w|-2$ of the tree has at least one branching node. Clearly, the depth of the tree is $|w|$. Since the root node is branching, the number of possible paths (words) up to level one is two. Moreover, if level $k > 0$ has $m_k \in \mathbb{N}$ branching nodes, then the number of all possible paths up to level $k+1$ is equal to the number of all possible paths up to level k, plus m_k. Then we get that the number of possible paths on the level of the leaf nodes is $2+m_1+\ldots+m_{|w|-1}+m_{|w|} = |w|$. We have stated, $m_i > 0$ for all $i = 1, \dots, |w|-2$, thus $m_{|w|-1} = m_{|w|} = 0$ and $2+m_1+\ldots+m_{|w|-2} = |w|$. If $m_i > 1$ for any $i \geq 1$, then $m_j = 0$ for some $j \neq i$. This is impossible, since all levels under $|w|-2$ have at least one branching node, thus $m_i = 1$ for all $i = 1, \dots, |w|-2$. $\square$

Now, let us analyze an interesting class of words. Let $f_1 = b$, $f_2 = a$ and define $f_n = f_{n-1} f_{n-2}$ for all $n \geq 3$. We call f_n (where $n \geq 1$) the *nth finite Fibonacci word.* The *infinite Fibonacci word* is the limit of the sequence $f_1, f_2, \dots$

The following lemma describes a well known property of the infinite Fibonacci words.

Lemma 2 (see Séébold [16])**.** *If a word u^2 is a factor of the infinite Fibonacci word, then u is a conjugate of some finite Fibonacci word.* $\square$

Note that the tree in Figure 3 represents the circular word obtained from f_5 which is the fifth Fibonacci word. See the trees of $(f_6)_\circ$ and $(f_7)_\circ$ in Figure 5. One can observe that the structure of these trees are very similar. This is strongly related to the definition of Fibonacci words.

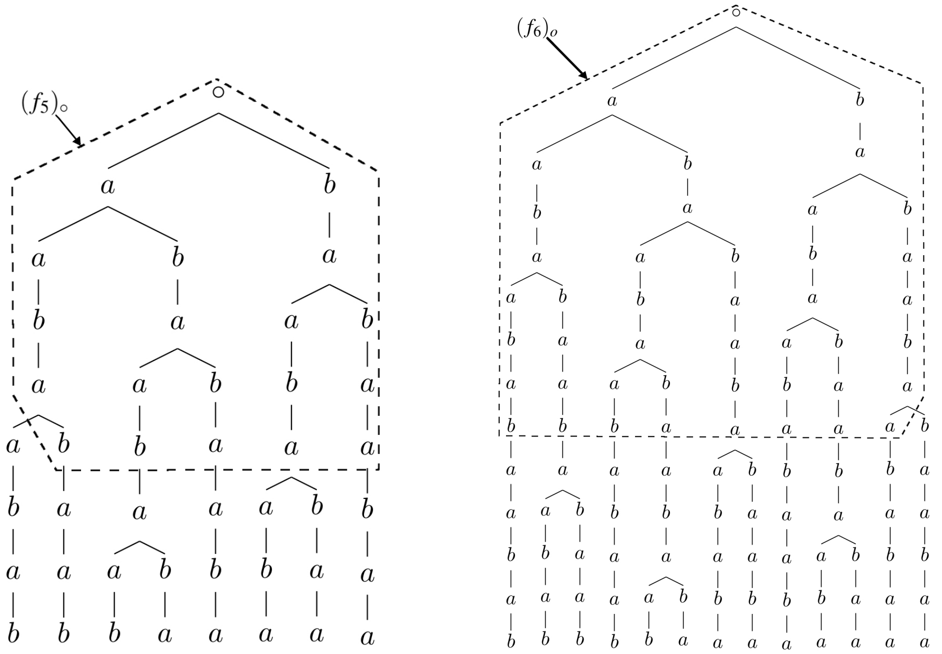


Figure 5: Trees of $(f_6)_o$ and $(f_7)_o$.

Theorem 3. *Let us denote the tree of the finite Fibonacci word f_i by φ_i for all $i \in \mathbb{N}$. Then for all $i \in \mathbb{N}$, the tree φ_i has exactly one branching node on all of its levels, except for the last two.*

Proof. Consider the tree φ_i of the circular Fibonacci word $(f_i)_\circ$ and let $\ell \in \{0,\dots,|f_i|\}$. The paths from $\circ$ to nodes on level k represent the length k factors of $(f_i)_\circ$. By the properties of Fibonacci words (or Sturmian words), we know that the number of distinct factors of length k in the infinite Fibonacci word is $k+1$. Since all of the length k words of the tree appear in the infinite Fibonacci word (because it has factor f_i^2), their number must not be more than $k+1$. On the other hand, each tree of a primitive word of length n must contain n branching nodes. Thus in φ_i all branching nodes must be on different levels. $\square$

Based on the proof, we can state the following about the trees of circular Fibonacci words.

Corollary 1. *For all $i, j \in \mathbb{N} \setminus \{0\}$, if $j > i$, then φ_i is a subtree of φ_j.*

Thus the trees of Fibonacci words are not only very similar, but they contain recurring subtrees. Notice in Figure 5, that the tree of $(f_5)_\circ$ appears in the tree of $(f_6)_\circ$ which also appears in the tree of $(f_7)_\circ$, marked by the dashed lines. Thus we can define the tree φ which belongs to the limit of the sequence of Fibonacci words, that is, the infinite Fibonacci word. Each path in the tree φ defines an infinite suffix of the infinite Fibonacci word. This is a consequence of the structure of the trees φ_i $(i = 1,2,\dots)$, since all of their words are factors of the infinite Fibonacci word and an infinite factor must be a suffix.

Let us state another interesting fact about branching nodes of trees of circular Fibonacci words.

Theorem 4. *Consider the tree φ_i for any $i \in \mathbb{N}$. Let u and u' be branching nodes of φ_i such that they lie on the same path and there are no other branching nodes between them. Then $|\ell(u) - \ell(u')|$ is a Fibonacci number.*

Proof. Assume the contrary, that is, there is a Fibonacci word f_i such that there are two branching nodes u, u' in tree φ_i that lie on the same path and do not have any other branching nodes between them, but $|\ell(u) - \ell(u')|$ is not a Fibonacci number. Then, there exists a Fibonacci word f_j with $j \geq i$ such that $(f_j)_\circ$ has square factor vv where v is the word constructed from the labels on the path between u and u'. Moreover, this will be true for all Fibonacci words $f_{j'}$ where $j' \geq j$. Thus the infinite Fibonacci word must contain the square factor vv. This contradicts Lemma 2, since v cannot be a conjugate of any Fibonacci word because its length is not a Fibonacci number. Thus our indirect assumption is false. $\square$

5 Conclusion and future directions

Combinatorics on circular words is a field that still has countless open problems and many possible research directions. We have shown some non-traditional methods of considering (representing) circular words. The following questions are still open and may lead to a better characterization of these sequences.

1. The algorithm presented in Section 3 does not always provide optimal solutions. Is there a way of deciding how to choose the best sequence of roots in the algorithm?
2. Theorem 2 is about the minimal roots of words over the two letter alphabet. What can we say about words over alphabets of more than two letters?
3. One could use the tree φ to deduce some properties of the infinite Fibonacci word.

4. Or the tree representations can be utilized to prove results about the structure of other (possibly infinite) words.
5. We believe, that Theorem 3 is true for all standard sturmian words (see e.g., [11] for their definition).

Acknowledgements

The authors would like to thank the reviewers for their valuable and useful comments. The work is supported by the TÁMOP 4.2.2/C-11/1/KONV-2012-0001 and 4.2.2/B-10/1-2010-0024 projects. The projects are implemented through the New Hungary Development Plan, co-financed by the European Social Fund and the European Regional Development Fund.

References

[1] Maxime Crochemore & Wojciech Rytter (2002): *Jewels of Stringology*. World Scientific Publishing Company, Incorporated, doi:10.1142/4838.

[2] James D. Currie & D. Sean Fitzpatrick (2002): *Circular words avoiding patterns. Proceedings of the 6th International Conference on Developments in Language Theory. LNCS 2450.*, pp. 319–325, doi:10.1007/3-540-45005-X_28.

[3] Volker Diekert, Tero Harju & Dirk Nowotka (2006): *Factorizations of cyclic words. Workshop on Words and Automata at CSR* 7.

[4] Szilárd Zsolt Fazekas & Benedek Nagy (2008): *Scattered Subword Complexity of Non-primitive Words. J. Autom. Lang. Comb.* 13(3), pp. 233–247.

[5] Nathan J. Fine & Herbert S. Wilf (1965): *Uniqueness theorems for periodic functions. Proceedings of the American Mathematical Society* 16, pp. 109–114, doi:10.1090/S0002-9939-1965-0174934-9.

[6] D. Sean Fitzpatrick (2005): *There are binary cube-free circular words of length n contained within the Thue-Morse word for all positive integers n. Ars Combinatorica* 74.

[7] László Hegedüs & Benedek Nagy (2013): *Periodicity of circular words. Local Proceedings of WORDS 2013, TUCS Lecture Notes* 20, pp. 45–56.

[8] M. Lothaire (1983): *Combinatorics on words.* Addison-Wesley.

[9] M. Lothaire (2002): *Algebraic Combinatorics on Words. Encyclopedia of Mathematics and its Applications* 90, Cambridge University Press, doi:10.1017/CBO9781107326019.

[10] M. Lothaire (2005): *Applied Combinatorics on Words. Encyclopedia of Mathematics and its Applications* 105, Cambridge University Press, doi:10.1017/CBO9781107341005.

[11] Aldo de Luca & Filippo Mignosi (1994): *Some combinatorial properties of Sturmian words. Theoretical Computer Science* 136(2), pp. 361–385, doi:10.1016/0304-3975(94)00035-H.

[12] Roger C. Lyndon & Marcel-Paul Schützenberger (1962): *The equation $a^M = b^N c^P$ in a free group. Michigan Math. J.* 9(4), pp. 289–298, doi:10.1307/mmj/1028998766.

[13] Dirk Nowotka (2004): *Periodicity and unbordered factors of words.* TUCS Dissertations No. 50.

[14] Benoît Rittaud & Laurent Vivier (2011): *Circular words and applications. Proceedings of Words 2011, Electronic Proceedings in Theoretical Computer Science* 63, pp. 31–36, doi:10.4204/EPTCS.63.6.

[15] Benoît Rittaud & Laurent Vivier (2012): *Circular words and three applications: factors of the Fibonacci word, $\mathscr{F}$-adic numbers, and the sequence 1, 5, 16, 45, 121, 320,.... Funct. Approx. Comment. Math.* 47(2), pp. 207–231, doi:10.7169/facm/2012.47.2.6.

[16] Patrice Séébold (1985): *Propriétés combinatoires des mots infinis engendrés par certains morphismes.* Thèse de doctorat, Université P. et M. Curie, Institut de Programmation.

[17] Arseny M. Shur (2010): *On ternary square-free circular words. The Electronic Journal of Combinatorics* 17.

[18] William Smyth (2003): *Computing patterns in strings.* Addison-Wesley.

[19] Axel Thue (1906): *Über unendliche Zeichenreihen. Kra. Vidensk. Selsk. Skrifter, I. Mat. Nat. Kl.* 7, pp. 1–22.

[20] Axel Thue (1912): *Über die gegenseitige Lage gleicher Teile gewisser Zeichenreihen. Kra. Vidensk. Selsk. Skrifter, I. Mat. Nat. Kl.* 46, pp. 1–67.

More Structural Characterizations of Some Subregular Language Families by Biautomata

Markus Holzer and Sebastian Jakobi
Institut für Informatik, Universität Giessen,
Arndtstr. 2, 35392 Giessen, Germany
{holzer,sebastian.jakobi}@informatik.uni-giessen.de

We study structural restrictions on biautomata such as, e.g., acyclicity, permutation-freeness, strongly permutation-freeness, and orderability, to mention a few. We compare the obtained language families with those induced by deterministic finite automata with the same property. In some cases, it is shown that there is no difference in characterization between deterministic finite automata and biautomata as for the permutation-freeness, but there are also other cases, where it makes a big difference whether one considers deterministic finite automata or biautomata. This is, for instance, the case when comparing strongly permutation-freeness, which results in the family of definite language for deterministic finite automata, while biautomata induce the family of finite and co-finite languages. The obtained results nicely fall into the known landscape on classical language families.

1 Introduction

The finite automaton is one of the first and most intensely investigated computational model in theoretical computer science, see, e.g., [15]. Its systematic study led to a rich and unified theory of regular subfamilies such as, for example, finite languages (are accepted by acyclic finite automata—here, except for non-accepting sink states, self-loops on states count as cycles), ordered languages (where the transitions of the accepting automata preserve an order on the state set), and star-free languages or non-counting languages (which can be described by regular like expressions using only union, concatenation, and complement or equivalently by permutation-free finite automata), to mention a few. Relations between several subregular language families, such as those mentioned above, are summarized in [6]. In particular, an extensive study of star-free regular languages can be found in [14]. Even nowadays the study of subregular language families from different perspectives such as, for instance, algebra, logic, descriptional, or computational complexity, is a vivid area of research.

Recently, an alternative automaton model to the deterministic finite automaton (DFA), the so called *biautomaton* (DBiA) [12] was introduced. Roughly speaking, a biautomaton consists of a *deterministic* finite control, a read-only input tape, and two reading heads, one reading the input from left to right (forward transitions), and the other head reading the input from right to left (backward transitions). Similar two-head finite automata models were introduced, e.g., in [5, 13, 17]. An input word is accepted by a biautomaton, if there is an accepting computation starting the heads on the two ends of the word meeting somewhere in an accepting state. Although the choice of reading a symbol by either head is nondeterministic, a deterministic outcome of the computation of the biautomaton is enforced by two properties: (i) The heads read input symbols independently, i.e., if one head reads a symbol and the other reads another, the resulting state does not depend on the order in which the heads read these single letters. (ii) If in a state of the finite control one head accepts a symbol, then this letter is accepted in this state by the other head as well. Later we call the former property the $\diamond$-property and the latter one the F-property. In [12] and a series of forthcoming papers [7, 8, 10, 11] it was shown that biautomata share a lot of properties with ordinary finite automata. For instance, as minimal DFAs, also minimal DBiAs are unique up to isomorphism [1, 12].

Now the question arises, which structural characterizations of subregular language families of DFAs carry over to biautomata. Let us give an example which involves partially ordered automata. A DFA with state set Q and input alphabet Σ is *partially ordered*, if there is a (partial) order $\leq$ on Q such that $q \leq \delta(q,a)$, for every $q \in Q$ and $a \in \Sigma$. In [4] it was shown that partially ordered DFAs characterize the family of $\mathscr{R}$-trivial regular languages, that is, a regular language L is $\mathscr{R}$-trivial if for its syntactic monoid M_L, the assumption $sM_L = tM_L$ implies $s = t$, for all $s,t \in M_L$. For the definition of the syntactic monoid of a regular language we refer to [1]. Adapting the definition of being partially ordered literally to DBiAs results in a characterization of the family of $\mathscr{J}$-trivial regular languages [11, 12]—originally the authors of [12] speak of acyclic biautomata instead, since loops are not considered as cycles there; we think that the term *partially ordered* is more suitable in this context. Here a regular language L is $\mathscr{J}$-trivial if for its syntactic monoid M_L, the assumption $M_LsM_L = M_LtM_L$ implies $s = t$, for all $s,t \in M_L$. Note that a language is $\mathscr{J}$-trivial regular if and only if it is piecewise testable [19]. A language $L \subseteq \Sigma^*$ is *piecewise testable* if it is a finite Boolean combination of languages of the form $\Sigma^*a_1\Sigma^*a_2\Sigma^* \ldots \Sigma^*a_n\Sigma^*$, where $a_i \in \Sigma$ for $1 \leq i \leq n$. We can also ask whether a transfer of conditions can be done the other way around from DBiAs to DFAs. This is not that obvious, since structural properties on DBiAs may involve conditions on the forward and backward transitions. For instance, in [7] it was shown that biautomata, where for every state and every input letter the forward and the backward transition go to the same state, characterize the family of commutative regular languages. A regular language $L \subseteq \Sigma^*$ is *commutative* if for all words $u,v \in \Sigma^*$ and letters $a,b \in \Sigma$ we have $uabv \in L$ if and only if $ubav \in L$. Obviously, this condition can be used also to give a structural characterization of commutative regular languages on DFAs, namely that for every state q and letters $a,b \in \Sigma$ the finite state device satisfies $\delta(\delta(q,a),b) = \delta(\delta(q,b),a)$. This is the starting point of our investigations.

We study structural properties of DFAs appropriately adapted to DBiAs, since up to our knowledge most classical properties from the literature on finite automata were not studied for DBiAs yet. Our investigation is started in Section 3 with automata which transition functions induce permutations on the state set. Originally permutation DFAs were introduced in [20]. We show that both types of finite state machines, permutation DFAs and permutation DBiAs are equally powerful. Thus, an alternative characterization of the family of p-regular languages in terms of DBiAs is obtained. Next we take a closer look on quite the opposite of permutation automata, namely on permutation-free devices—see Section 4. A special case of a permutation-free automaton is an acyclic (expect for sink states) one. It is easy to see that acyclic DFAs as well as DBiAs characterize the family of finite languages. An important subregular language family, which can be obtained from finite languages by finitely many applications of concatenation, union, and complementation with respect to the underlying alphabet, is the class of star-free languages. It obeys a variety of different characterizations [14], one of them are permutation-free DFAs. We show that permutation-free DBiAs characterize the star-free languages, too. For strongly permutation-free automata, which are automata that are permutation-free and where also the identity permutation is forbidden, we find the first significant difference of DFAs and DBiAs. While for DFAs this property characterizes the family of definite languages, DBiAs describe only finite or co-finite languages. A language $L \subseteq \Sigma^*$ is *definite* [16] if and only if $L = L_1 \cup \Sigma^*L_2$, for some finite languages L_1 and L_2. Moreover, we find a relation between strongly permutation-free automata, and automata where all states are almost-equivalent—the notion of almost-equivalence was introduced in [2]. Then in Section 5 we continue our investigation with another important subfamily of star-free languages, namely ordered languages [18]. A DFA with state set Q and input alphabet Σ is *ordered* if there is a total order $\leq$ on the state set Q such that $p \leq q$ implies $\delta(p,a) \leq \delta(q,a)$, for every $p,q \in Q$ and $a \in \Sigma$.

Property	Automata type	
	DFAs	DBiAs
permutation	p-regular	p-regular
permutation-free	star-free	star-free
ordered	ordered	FIN $\cup$ co-FIN $\subset \cdot \subset$ ORD
partially ordered	$\mathscr{R}$-trivial	$\mathscr{J}$-trivial
strongly permutation-free	definite	finite and co-finite
acyclic; self-loops are cycles	finite	finite
non-exiting	prefix-free	circumfix-free
non-returning	strict superset of suffix-free	strict subset of non-returning DFAs

Table 1: Comparison of the results on structural properties on DFAs and DBiAs and their induced language families (shading represents results obtained in this paper); here FIN refers to the family of finite languages, co-FIN to the family of co-finite languages, and ORD to the family of ordered languages.

The family of ordered languages lies strictly in-between the family of finite and the family of star-free languages. Appropriately adapting this definition to biautomata results in a language class, which we call the family of bi-ordered languages, that is a proper superset of the family of finite and co-finite languages and a strict subset of the family of ordered languages. Moreover, it is shown that there is a subtle difference whether the order condition is applied to automata in general or to minimal devices only. In the next to last section we take a closer look on non-exiting and non-returning machines. It is well known that non-exiting DFAs characterize the family of prefix-free languages, while non-returning automata are related to suffix-free languages. We show that every biautomaton which is non-exiting must also be non-returning (unless it accepts the empty language), and that non-exiting minimal DBiAs characterize the family of circumfix-free languages. For non-returning minimal DBiAs we prove that the induced language family is a strict subset of the family of non-returning minimal DFAs languages. The obtained results are summarized in Table 1. In the last section we briefly discuss our findings and give some hints on future research directions on the subject under consideration.

2 Preliminaries

A *deterministic finite automaton* (DFA) is a quintuple $A = (Q, \Sigma, \delta, q_0, F)$, where Q is the finite set of *states*, Σ 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 $\delta \colon Q \times \Sigma \to Q$ is the *transition function*. As usual, the transition function δ can be recursively extended to $\delta \colon Q \times \Sigma^* \to Q$. The *language accepted* by A is defined as $L(A) = \{\, w \in \Sigma^* \mid \delta(q_0, w) \in F \,\}$.

A *deterministic biautomaton* (DBiA) is a sixtuple $A = (Q, \Sigma, \cdot, \circ, q_0, F)$, where Q, Σ, q_0, and F are defined as for DFAs, and where $\cdot$ and $\circ$ are mappings from $Q \times \Sigma$ to Q, called the *forward* and *backward transition function*, respectively. It is common in the literature on biautomata to use an infix notation for these functions, i.e., writing $q \cdot a$ and $q \circ a$ instead of $\cdot(q, a)$ and $\circ(q, a)$. Similar as for the transition function of a DFA, the forward transition function $\cdot$ can be extended to $\cdot \colon Q \times \Sigma^* \to Q$ by $q \cdot \lambda = q$ and $q \cdot av = (q \cdot a) \cdot v$, for all states $q \in Q$, symbols $a \in \Sigma$, and words $v \in \Sigma^*$. Here λ refers to the *empty word*. The extension of the backward transition function $\circ$ to $\circ \colon Q \times \Sigma^* \to Q$ is defined as follows: $q \circ \lambda = q$ and $q \circ va = (q \circ a) \circ v$, for all states $q \in Q$, symbols $a \in \Sigma$, and words $v \in \Sigma^*$. Notice that $\circ$ consumes the input from right to left, hence the name backward transition function.

The DBiA A *accepts* a word $w \in \Sigma^*$ if there are words $u_i, v_i \in \Sigma^*$, for $1 \leq i \leq k$, such that w can be written as $w = u_1 u_2 \ldots u_k v_k \ldots v_2 v_1$, and

$$((\ldots((((q_0 \cdot u_1) \circ v_1) \cdot u_2) \circ v_2) \ldots) \cdot u_k) \circ v_k \in F.$$

The language accepted by A is $L(A) = \{\, w \in \Sigma^* \mid A \text{ accepts } w \,\}$.

The DBiA A has the $\diamond$*-property*, if $(q \cdot a) \circ b = (q \circ b) \cdot a$, for all $a, b \in \Sigma$, and $q \in Q$, and it has the F*-property*, if for all $q \in Q$ and $a \in \Sigma$ it is $q \cdot a \in F$ if and only if $q \circ a \in F$. The biautomata as introduced in [12] always had to satisfy both these properties, while in [7, 8] also biautomata that lack one or both of these properties, as well as nondeterministic biautomata were studied. Throughout the current paper, when writing of biautomata, or DBiAs, we always mean deterministic biautomata that satisfy both the $\diamond$-property, and the F-property, i.e., the model as introduced in [12]. For such biautomata the following it is known from the literature [7, 12]:

- $(q \cdot u) \circ v = (q \circ v) \cdot u$, for all states $q \in Q$ and words $u, v \in \Sigma^*$,
- $(q \cdot u) \circ vw \in F$ if and only if $(q \cdot uv) \circ w \in F$, for all states $q \in Q$ and words $u, v, w \in \Sigma^*$.

From this one can conclude that for all words $u_i, v_i \in \Sigma^*$, with $1 \leq i \leq k$, it is

$$((\ldots((((q_0 \cdot u_1) \circ v_1) \cdot u_2) \circ v_2) \ldots) \cdot u_k) \circ v_k \in F$$

if and only if

$$q_0 \cdot u_1 u_2 \ldots u_k v_k \ldots v_2 v_1 \in F.$$

Therefore, the language accepted by a DBiA A can as well be defined as $L(A) = \{\, w \in \Sigma^* \mid q_0 \cdot w \in F \,\}$.

Let A be DFA or a DBiA with state set Q. We say that a state $q \in Q$ is a *sink state* if and only if all outgoing transition (regardless whether they are forward or backward transitions) are self-loops only. Note, that in particular, one can distinguish between accepting and non-accepting sink states.

In the following we define the two DFAs contained in a DBiA, which accept the language, and the reversal of the language accepted by the biautomaton. Let $A = (Q, \Sigma, \cdot, \circ, q_0, F)$ be a DBiA. We denote by Q_{fwd} the set of all states reachable from q_0 by only using forward transitions, and denote the set of states reachable by only using backward transitions by Q_{bwd}, i.e.,

$$Q_{\text{fwd}} = \{\, q \in Q \mid \exists u \in \Sigma^* : q_0 \cdot u = q \,\} \quad \text{and} \quad Q_{\text{bwd}} = \{\, q \in Q \mid \exists v \in \Sigma^* : q_0 \circ v = q \,\}.$$

Now we define the DFA $A_{\text{fwd}} = (Q_{\text{fwd}}, \Sigma, \delta_{\text{fwd}}, q_0, F_{\text{fwd}})$, with $F_{\text{fwd}} = Q_{\text{fwd}} \cap F$, and $\delta_{\text{fwd}}(q, a) = q \cdot a$, for all states $q \in Q_{\text{fwd}}$ and symbols $a \in \Sigma$. Similarly, we define the DFA $A_{\text{bwd}} = (Q_{\text{bwd}}, \Sigma, \delta_{\text{bwd}}, q_0, F_{\text{bwd}})$, with $F_{\text{bwd}} = Q_{\text{bwd}} \cap F$, and $\delta_{\text{bwd}}(q, a) = q \circ a$, for all $q \in Q$ and $a \in \Sigma$. One readily sees that $L(A_{\text{fwd}}) = L(A)$. Moreover, since $q \circ uv = (q \circ v) \circ u$, one can also see $L(A_{\text{bwd}}) = L(A)^R$. It is shown in [9] that if A is a minimal biautomaton, then the two DFAs A_{fwd} and A_{bwd} are minimal, too.

3 Permutation Automata

First we study automata where every input induces a permutation on the state set. Such finite automata were defined in [20]. A DFA $A = (Q, \Sigma, \delta, q_0, F)$ is a *permutation DFA* if $\delta(p, a) = \delta(q, a)$ implies $p = q$, for all $p, q \in Q$ and $a \in \Sigma$. A regular language is *p-regular* if it is accepted by a permutation DFA. We give a similar definition for biautomata: a biautomaton $A = (Q, \Sigma, \cdot, \circ, q_0, F)$ is a *permutation biautomaton* if for all $p, q \in Q$ and $a \in \Sigma$ we have that $p \cdot a = q \cdot a$ implies $p = q$, and also $p \circ a = q \circ a$ implies $p = q$.

We will see that a language is p-regular if and only if it is accepted by a permutation biautomaton. Before we can show this, we describe a useful technique to construct a biautomaton from finite automata. In [12] a construction of a biautomaton from a given DFA A is described, that uses a cross-product construction of A with the power-set automaton of the reversal of A. In the following we describe how a biautomaton can be constructed from two arbitrary DFAs accepting a regular language and its reversal.

Let $L \subseteq \Sigma^*$ be a regular language, and for $i \in \{1,2\}$ let $A_i = (Q_i, \Sigma, \delta_i, q_0^{(i)}, F_i)$ be DFAs with $L(A_1) = L$, and $L(A_2) = L^R$. Further, for all states $p \in Q_1$ let u_p be some word with $\delta_1(q_0^{(1)}, u_p) = p$, and similarly for $q \in Q_2$ let v_q be a word with $\delta_2(q_0^{(2)}, v_q) = q$. Then define the automaton $B_{A_1 \times A_2} = (Q, \Sigma, \cdot, \circ, q_0, F)$ with state set $Q = Q_1 \times Q_2$, initial state $q_0 = (q_0^{(1)}, q_0^{(2)})$, accepting states $F = \{\,(p,q) \in Q \mid u_p v_q^R \in L\,\}$, and where for all $(p,q) \in Q$ and $a \in \Sigma$ we have $(p,q) \cdot a = (\,\delta_1(p,a), q\,)$, and $(p,q) \circ a = (\,p, \delta_2(q,a)\,)$. The following lemma proves the correctness of this construction.

Lemma 1 *For $i \in \{1,2\}$ let $A_i = (Q_i, \Sigma, \delta_i, q_0^{(i)}, F_i)$ be DFAs with $L(A_1) = L$, and $L(A_2) = L^R$. Then $B_{A_1 \times A_2}$ is a deterministic biautomaton, such that $L(B_{A_1 \times A_2}) = L$.*

Besides its usefulness for our result on permutation biautomata, this construction is also of relevance from a descriptional complexity point of view. Using the construction from [12] on an n-state DFA yields a biautomaton with $n \cdot 2^n$ states. In fact, a precise analysis in [10] that uses a similar construction as in [12] proves a tight bound of $n \cdot 2^n - 2(n-1)$ states for converting an n-state DFA into an equivalent biautomaton. However, this bound only takes into account the state complexity of the original language L, but not the state complexity of L^R. If the state complexity of L^R much smaller than 2^n then the bound from [10] is far off the number of states of the minimal biautomaton for L. Using our construction, we can deduce an upper bound of $n \cdot m$ for the number of states of a biautomaton for the language L, if n is the state complexity of L, and m is the state complexity of L^R.

Now we show our result on permutation automata.

Theorem 2 *A language is p-regular if and only if it is accepted by some permutation biautomaton.*

Proof: If A is a permutation biautomaton, then A_{fwd} is a permutation DFA, hence $L(A)$ is p-regular. For the reverse implication let L be some p-regular language over the alphabet Σ. Then L^R is p-regular, too [20], so there are permutation DFAs $A_i = (Q_i, \Sigma, \delta_i, q_0^{(i)}, F_i)$, for $i = 1,2$, that $L = L(A_1)$, and $L^R = L(A_2)$. Using the cross-product construction from Lemma 1, we obtain the biautomaton $B = B_{A_1 \times A_2}$. Recall that the states of B are of the form (p,q), with $p \in Q_1$ and $q \in Q_2$, and the transitions are defined such that $(p,q) \cdot a = (\delta_1(p,a), q)$, and $(p,q) \circ a = (p, \delta_2(q,a))$, for all states (p,q) and symbols $a \in \Sigma$. We will show in the following that B is a permutation automaton. Therefore let (p,q) and (p',q') be two states of B, and $a \in \Sigma$. If $(p,q) \cdot a = (p',q') \cdot a$ then $(\delta(p,a), q) = (\delta(p',a), q')$, which implies $\delta(p,a) = \delta(p',a)$ and $q = q'$. Since A_1 is a permutation DFA, we also obtain $p = p'$, hence $(p,q) = (p',q')$. With a similar reasoning, using the permutation property of A_2, we see that also $(p,q) \circ a = (p',q') \circ a$ implies $(p,q) = (p',q')$, therefore B is a permutation biautomaton. $\square$

4 Permutation-Free Automata

An important subregular language family is the class of star-free languages. A language is star-free if it can be obtained from finite languages by finitely many applications of concatenation, union, and complementation with respect to the underlying alphabet. For the class of finite languages we have the following obvious theorem, which we state without proof.

Theorem 3 *A language is finite (co-finite, respectively) if and only if its minimal biautomaton is acyclic except for non-accepting (accepting, respectively) sink states; self-loops count as cycles.* ∎

The class of star-free languages obeys a variety of different characterizations [14], one of them being the following: a regular language is star-free if its minimal DFA is permutation-free. A DFA $A = (Q,\Sigma,\delta,q_0,F)$ is *permutation-free* if there is no word $w \in \Sigma^*$ such that the mapping $q \mapsto \delta(q,w)$, for all $q \in Q$, induces a *non-trivial* permutation, i.e., a permutation different from the identity permutation, on some set $P \subseteq Q$. Now the question arises whether a similar condition for biautomata also yields a characterization of the star-free languages. We feel that the following definition of permutation-freeness is a natural extension from the corresponding definition for DFAs. We say that a biautomaton $A = (Q,\Sigma,\cdot,\circ,q_0,F)$ is *permutation-free* if there are no words $u,v \in \Sigma^*$, such that the mapping $q \mapsto (q\cdot u)\circ v$ induces a non-trivial permutation on some set of states $P \subseteq Q$. It turns out that with this definition, permutation-free biautomata indeed characterize the star-free languages. We will later discuss some other possible definitions. Before we show our result on permutation-free biautomata, we prove the following lemma which helps us to relate permutations in biautomata to permutations in DFAs.

Lemma 4 *Let Q be a finite set and $\pi_1,\pi_2\colon Q \to Q$ be two mappings satisfying $\pi_1(\pi_2(q)) = \pi_2(\pi_1(q))$ for all $q \in Q$. If there exists a subset $P \subseteq Q$ such that the mapping $\pi\colon P \to P$ defined by $\pi(p) = \pi_2(\pi_1(p))$ is a non-trivial permutation on P, then there exists a subset $P' \subseteq Q$ and an integer $d \geq 1$ such that π_1^d or π_2^d is a non-trivial permutation on P'.*

Proof: Consider the sequence of sets $\pi_1^0(P),\pi_1^1(P),\pi_1^2(P),\ldots \subseteq Q$. Since Q is a finite set, the number of different sets $\pi_1^j(P)$, for $j \geq 0$, is finite. Thus, there must be integers $m,d \geq 1$ such that the sets $\pi_1^0(P),\pi_1^1(P),\ldots\pi_1^{m+d-1}(P)$ are pairwise distinct, and $\pi_1^{m+d}(P) = \pi_1^m(P)$. Since $\pi = \pi_1\pi_2 = \pi_2\pi_1$ is a permutation on P, we obtain

$$P = \pi^{m+d}(P) = \pi_2^{m+d}(\pi_1^{m+d}(P)) = \pi_2^d(\pi_2^m(\pi_1^m(P))) = \pi_2^d(P),$$

which shows that π_2^d is a permutation on P. It follows that also π_1^d must be a permutation on P. If one of these is a non-trivial permutation we are done. Therefore assume that both π_1^d and π_2^d are the identity permutation on P. In this case it must be $d \geq 2$ because otherwise the permutation $\pi = \pi_1\pi_2$ would be trivial. Then the two sets P and $\pi_1(P)$ are different, so there is an element $q \in P$ with $\pi_1(q) \neq q$. On the other hand q must satisfy $\pi_1^d(q) = q$. Therefore the mapping π_1 is a permutation on the set $P' = \{\pi_1^0(p),\pi_1^1(p),\ldots,\pi_1^{d-1}(p)\}$, and it is non-trivial because $\pi_1^0(p) = p \neq \pi_1^1(p)$. ∎

Now we can show the following characterization of star-free languages in terms of permutation-free biautomata.

Theorem 5 *A language is star-free if and only if its* minimal *biautomaton is permutation-free.*

Proof: Clearly, if A is a minimal biautomaton that is permutation-free, then also the contained minimal DFA A_{fwd} is permutation-free, too. Therefore the language $L(A)$ is star-free.

For proving the reverse implication let $A = (Q,\Sigma,\cdot,\circ,q_0,F)$ be a minimal biautomaton that is *not* permutation-free. Then there are words $u,v \in \Sigma^*$, and a set of states $P \subseteq Q$, with $|P| \geq 2$, such that the mapping $\pi\colon Q \to Q$ defined as $\pi(q) = (q\cdot u)\circ v$ induces a non-trivial permutation on P. Notice that the $\diamond$-property of the biautomaton A implies $\pi(q) = \pi_1(\pi_2(q)) = \pi_2(\pi_1(q))$, for $\pi_1(q) = q\cdot u$, and $\pi_2(q) = q\circ v$. Therefore we can use Lemma 4, and obtain an integer $d \geq 1$ such that π_1^d or π_2^d induces a non-trivial permutation on some subset $P' \subseteq Q$. If π_1^d is non-trivial, then the word u^d induces a non-trivial permutation in the minimal DFA A_{fwd}, which in turn means that the language $L(A)$ is *not* star-free.

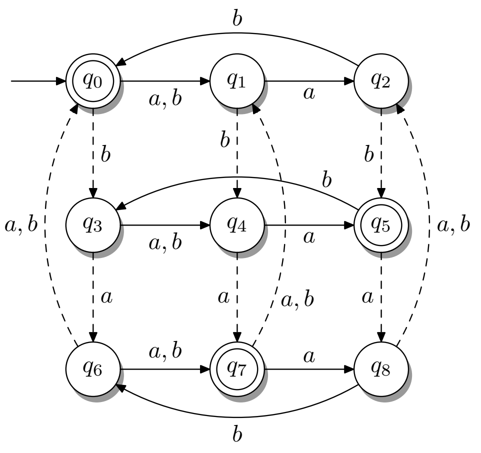


Figure 1: The permutation-free biautomaton A that has a word-cycle and a graph-cycle.

Otherwise, the mapping π_2^d is non-trivial, and the word v^d induces a non-trivial permutation on the minimal DFA A_{bwd}, which means that the language $L(A)^R$ is not star-free. Since the class of star-free languages is closed under reversal, we again conclude that the language $L(A)$ cannot be star-free in this case. □

In the above definition of permutation-free biautomata, from all states in the permutation induced by the word uv, the prefix u must be read with forward transitions and the suffix v with backward transitions. One could also think of other kinds of permutations in biautomata, and we shortly discuss two different such notions in the following. Since a permutation is composed of cycles, we describe the types of cycles. Let $P = \{p_0, p_1, \ldots, p_{k-1}\}$ be some set of states of a biautomaton A and w be some non-empty word over the input alphabet of A.

- We say that w induces a *word-cycle* on P if for $0 \leq i \leq k-1$ we have $p_{(i+1) \bmod k} = (p_i \cdot u_i) \circ v_i$, for some words u_i and v_i, with $u_i v_i = w$.
- We say that w induces a *graph-cycle* on P if $w = a_1 a_2 \ldots a_n$ and we have

$$p_{(i+1) \bmod k} = (\ldots((p_i \bullet_{i,1} a_1) \bullet_{i,2} a_2) \ldots) \bullet_{i,n} a_n,$$

  for $0 \leq i \leq k-1$, where $\bullet_{i,j} \in \{\cdot, \circ\}$, for $1 \leq j \leq n$. The intuition behind the definition of a graph-cycle is that the word w specifies the sequence of transitions (regardless whether they are forward or backward transitions) that are taken during the course of the computation.

Notice that if a biautomaton has a permutation as defined before Theorem 5, then it also has a word-cycle, and also a graph-cycle. Hence, if a language is accepted by a biautomaton that has no word-cycle or by a biautomaton that has no graph-cycle, then it is star-free. However, the converse is not true, as the following example shows.

Example 6 *Let $A = (Q, \Sigma, \cdot, \circ, q_0, F)$ be the* minimal *DBiA for the language $L = \{aab, bab\}^*$. The biautomaton A is depicted in Figure 1—solid arrows denote forward transitions by $\cdot$, and dashed arrows denote backward transitions by $\circ$. By inspecting the DFA A_{fwd}, consisting of the states q_0, q_1, q_2, and the non-accepting sink state, which is not shown, one can see that A_{fwd} is permutation-free. Therefore the language L is star-free, and also the biautomaton A must be permutation-free. However, the word ab induces a word-cycle on the states q_0, q_1, and q_6 because $q_0 \circ ab = q_6$, $(q_6 \cdot a) \circ b = q_1$, and $q_1 \cdot ab = q_0$. Further, the word ab also induces a graph-cycle on the states q_0, q_4, and q_3 because $(q_0 \cdot a) \circ b = q_4$, $(q_4 \cdot a) \cdot b = q_3$, and $(q_3 \circ a) \circ b = q_0$.*

4.1 Strongly Permutation-Free Automata

A permutation-free automaton does not contain any *non-trivial* permutation, but it may contain the identity permutation. We may also forbid the identity permutation, which leads to the following definitions. A DFA $A = (Q, \Sigma, \delta, q_0, F)$ is *strongly permutation-free* if there is no non-empty word $w \in \Sigma^+$, such that the mapping $q \mapsto \delta(q, w)$ induces a permutation on some set $P \subseteq Q$, with $|P| \geq 2$. Similarly, a biautomaton $A = (Q, \Sigma, \cdot, \circ, q_0, F)$ is *strongly permutation-free* if there are no words $u, v \in \Sigma^*$, with $uv \in \Sigma^+$, such that the mapping $q \mapsto (q \cdot u) \circ v$ induces a permutation on some set $P \subseteq Q$, with $|P| \geq 2$. In these definitions we require the words w and uv to be non-empty because the empty-word always induces the identity permutation on all sets of states. Further we only consider subsets $P \subseteq Q$ with $|P| \geq 2$ because every DFA and every biautomaton must contain a (maybe identity) permutation on a set $P \subseteq Q$ with $|P| \geq 1$, since by the pigeon hole principle there is a state which is repeatedly visited by only reading the letter a long enough.

Before we study which languages are accepted by strongly permutation-free automata, we give some further definitions which turn out to be related to strongly permutation-freeness. Let $A = (Q, \Sigma, \delta, q_0, F)$ be a DFA, and $w \in \Sigma^+$ some non-empty word. A state $q \in Q$ is called a *w-attractor* in A, if for all states $p \in Q$ there is an integer $k > 0$ such that $\delta(p, w^k) = q$. For a biautomaton $A = (Q, \Sigma, \cdot, \circ, q_0, F)$ and words $u, v \in \Sigma^*$, with $|uv| \geq 1$, we denote by $\pi_{u,v}$ the mapping $p \mapsto (p \cdot u) \circ v$. Now a state $q \in Q$ is a *(u,v)-attractor* in A if for all $p \in Q$ there is an integer $k > 0$ such that $\pi^k_{u,v}(p) = q$. Notice that due to the $\diamond$-property of A the condition $\pi^k_{u,v}(p) = q$ can also be written as $(p \cdot u^k) \circ v^k = q$. Next we recall the definition of almost-equivalence. Two languages L_1 and L_2 are *almost-equivalent* ($L_1 \sim L_2$) if their symmetric difference $L_1 \triangle L_2 = (L_1 \setminus L_2) \cup (L_2 \setminus L_1)$ is finite. This notion naturally transfers to states as follows. For a state q of some DFA or biautomaton A we denote by $L_A(q)$ the language accepted by the automaton ${}_qA$ which is obtained from A by making state q its initial state. The language $L_A(q)$ is also called the *right language* of q. Now two states p and q of a DFA or biautomaton A are *almost-equivalent* ($p \sim q$) if their *right languages* $L_A(p)$ and $L_A(q)$ are almost-equivalent. We write $p \equiv q$, if p and q are *equivalent*, i.e., if $L_A(p) = L_A(q)$. Our next theorem connects the notions of almost-equivalence, w-attractors, and strongly permutation-freeness for DFAs, and shows that these conditions can be used to characterize the class of definite languages—a language L over an alphabet Σ is *definite* if there are finite languages L_1 and L_2 over Σ such that $L = L_1 \cup \Sigma^* L_2$. Interestingly the relation between definite languages and almost-equivalence was already studied in [16], long before the notion of almost-equivalence became popular in [2]—in [16] the used form of equivalence was not called "almost-equivalence," but simply "equivalent." Moreover, the relation between definite languages and strongly permutation-free automata was independently shown in [3]—compare also with [14, Exercise 28 of Chapter 4 and Exercise 13 of Chapter 5].

Theorem 7 *Let $A = (Q, \Sigma, \delta, q_0, F)$ be some* minimal *DFA, then the following statements are equivalent:*

1. *All states in Q are pairwise almost-equivalent.*
2. *For all words $w \in \Sigma^+$, there is a w-attractor in A.*
3. *A is strongly permutation-free.*
4. *$L(A)$ is a definite language.*

Now we turn to biautomata, where we will see that the equivalences between the first three conditions of Theorem 7 also hold in the setting of biautomata. However, we will see that the language class related to these conditions is different. Before we come to this result we recall the following lemma from [9]:

Lemma 8 *Let $A = (Q,\Sigma,\cdot,\circ,q_0,F)$ and $A' = (Q',\Sigma,\cdot',\circ',q'_0,F')$ be two biautomata, and let $p \in Q$ and $q \in Q'$. Then $p \sim q$ if and only if $(p \cdot u) \circ v \sim (q \cdot' u) \circ' v$, for all words $u,v \in \Sigma^*$. Moreover, $p \sim q$ implies $(p \cdot u) \circ v \equiv (q \cdot' u) \circ' v$, for all words $u,v \in \Sigma^*$ with $|uv| \geq k = |Q \times Q'|$.*

The two automata A and A' in Lemma 8 need not be different, so this lemma can also be used for states p and q in one biautomaton A. Further, if this biautomaton A is minimal, then it does not contain a pair of different, but equivalent states. In this case the states p and q are almost-equivalent if and only if for all long enough words $uv \in \Sigma^*$ the two states $(p \cdot u) \circ v$ and $(q \cdot u) \circ v$ are the same state. We obtain the following corollary.

Corollary 9 *Let $A = (Q,\Sigma,\cdot,\circ,q_0,F)$ be a* minimal *biautomaton, and $k = |Q \times Q|$. Two states $p,q \in Q$ are almost-equivalent if and only if for all words $u,v \in \Sigma^*$, with $|uv| \geq k$, it is $(p \cdot u) \circ v = (q \cdot u) \circ v$.* □

Now we are ready for our result on strongly permutation-free biautomata.

Theorem 10 *Let $A = (Q,\Sigma,\cdot,\circ,q_0,F)$ be some* minimal *biautomaton, then the following statements are equivalent:*

1. *All states in Q are pairwise almost-equivalent.*
2. *There is a state $s \in Q$ that is a (u,v)-attractor in A, for all $u,v \in \Sigma^*$ with $|uv| \geq 1$.*
3. *A is strongly permutation-free.*
4. *$L(A)$ is a finite or co-finite language.*

Proof: Let $A = (Q,\Sigma,\cdot,\circ,q_0,F)$ be a minimal biautomaton. First assume $L(A)$ is a finite language, and let ℓ be the length of a longest word in $L(A)$. Then the right language $L_A(q)$ of every state $q \in Q$ is finite, so all states in Q are pairwise almost-equivalent. Since $L(A)$ is finite, and A is minimal, the biautomaton has a non-accepting sink state s, with $s \cdot a = s \circ a = s$ for all $a \in \Sigma$. Moreover, from any state $q \in Q$ the automaton always reaches this sink state s after reading at most ℓ symbols. Therefore, the state s is a (u,v)-attractor in A, for all words $u,v \in \Sigma^*$ with $|uv| \geq 1$. It also follows that the only permutation that is possible in A is the identity permutation on the singleton set $\{s\}$, hence A is strongly permutation-free. The case where $L(A)$ is a co-finite language is similar. The only differences are that the sink state s is an accepting state, and the integer ℓ must be the length of the longest word that is *not* in $L(A)$. This shows that statement 4 implies all other statements, and it remains to prove the other directions.

Assume that all states in Q are pairwise almost-equivalent, and let k be the integer from Corollary 9. If the length of every word in $L(A)$ is less than k then $L(A)$ is a finite language, so assume that there is a word $w \in L(A)$ with $|w| \geq k$. We show that in this case language $L(A)$ contains every word of length at least k, and thus, is co-finite. Since $|w| \geq k$, we can write w as $w = w_1w_2$, with $|w_2| = k$. Because $w \in L(A)$ we have $(q_0 \cdot w_1) \circ w_2 \in F$. Now let $u \in \Sigma^{\geq k}$, and consider the states $p = (q_0 \cdot w_1)$ and $q = (q_0 \cdot u)$. Since all states are almost-equivalent, and the word w_2 has length k, we can use Corollary 9 to obtain $(p \cdot \lambda) \circ w_2 = (q \cdot \lambda) \circ w_2$. Hence the state $q \circ w_2 = (q_0 \cdot u) \circ w_2$ is accepting. By the $\diamond$-property of A we have $(q_0 \cdot u) \circ w_2 = (q_0 \circ w_2) \cdot u$, and another application of Corollary 9 on the almost-equivalent states q_0 and $q_0 \circ w_2$ we obtain $q_0 \cdot u = (q_0 \circ w_2) \cdot u$, because $|u| \geq k$. This shows that the word u is accepted by A, hence $L(A)$ is co-finite. This shows that statements 1 and 4 are equivalent.

Next assume there is a state $s \in Q$ that is a (u,v)-attractor for all words $u,v \in \Sigma^*$ with $|uv| \geq 1$. Then A must be strongly permutation-free, which can be seen as follows. Assume that there are words $u,v \in \Sigma^*$ with $|uv| \geq 1$ such that the mapping $\pi\colon q \mapsto (q \cdot u) \circ v$ is a permutation on some set $P \subseteq Q$, with $|P| \geq 2$. Then it must be $\left|\pi^i(P)\right| = |P| \geq 2$, for all $i \geq 0$. But since s is a (u,v)-attractor, there is an integer $m \geq 0$

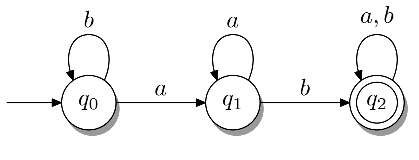


Figure 2: The minimal DFA A for the language $\Sigma^* ab \Sigma^*$ over the alphabet $\Sigma = \{a,b\}$. The states of A_1 can be ordered by $q_0 \leq q_1 \leq q_2$.

such that $(q \cdot u^m) \circ v^m = s$, for all states $q \in Q$. Then $|\pi^m(P)| = 1$, which is a contradiction, therefore A is strongly permutation-free.

Now it is sufficient to show that statement 3 implies statement 1. Therefore let A be strongly permutation-free, and assume for the sake of contradiction, that there are two states $p,q \in Q$ with $p \not\sim q$. Corollary 9 now implies that there are words $u,v \in \Sigma^*$, with $|uv| \geq |Q \times Q|$, such that $(p \cdot u) \circ v \neq (q \cdot u) \circ v$. Since the number of steps in the computations $(p \cdot u) \circ v$ and $(q \cdot u) \circ v$ is at least $|Q \times Q|$, the words u and v can be written as $u = u_1 u_2 u_3$ and $v = v_3 v_2 v_1$ such that

$$\begin{aligned} &(p \cdot u_1) \circ v_1 = p_1, && (p_1 \cdot u_2) \circ v_2 = p_1, && (p_1 \cdot u_3) \circ v_3 = (p \cdot u) \circ v, \\ &(q \cdot u_1) \circ v_1 = q_1, && (q_1 \cdot u_2) \circ v_2 = q_1, && (q_1 \cdot u_3) \circ v_3 = (q \cdot u) \circ v. \end{aligned}$$

But then the mapping $\pi \colon r \mapsto (r \cdot u_2) \circ v_2$ is a permutation (the identity permutation) on the states $\{p_1, q_1\}$. Since A is strongly permutation-free, it follows $p_1 = q_1$, and in turn $(p \cdot u) \circ v = (q \cdot u) \circ v$. This contradicts the assumption $p \not\sim q$, and concludes our proof. $\square$

5 Ordered Automata

We now study automata where one can find an order on the state set that is compatible with the transitions of the automaton. Ordered DFAs and their accepted languages were studied in [18]. A DFA $A = (Q, \Sigma, \delta, q_0, F)$ is *ordered* if there exists some total order $\leq$ on the state set Q such that $p \leq q$ implies $\delta(p,a) \leq \delta(q,a)$, for all states $p,q \in Q$ and symbols $a \in \Sigma$. Similarly, a biautomaton $A = (Q, \Sigma, \cdot, \circ, q_0, F)$ is *ordered* if there is a total order $\leq$ on Q such that $p \leq q$ implies $p \cdot a \leq q \cdot a$ as well as $p \circ a \leq q \circ a$, for all states $p,q \in Q$ and symbols $a \in \Sigma$. A regular language is *ordered* if it is accepted by some ordered DFA, and it is *bi-ordered* if it is accepted by an ordered biautomaton. Moreover, a language is *strictly ordered* if its minimal DFA is ordered, and it is *strictly bi-ordered* if its minimal biautomaton is ordered.

The next two results show that the class of bi-ordered languages is located between the class of ordered languages and the class of finite and co-finite languages.

Theorem 11 *The class of bi-ordered languages is strictly contained in the class of ordered languages.*

Proof: If L is a bi-ordered language, then it is accepted by some ordered DBiA A. Then of course the automaton A_{fwd} is an ordered DFA. Therefore any bi-ordered language is an ordered language. The strictness of this inclusion is witnessed by the language $\Sigma^* ab \Sigma^*$ over the alphabet $\Sigma = \{a,b\}$, which is accepted by the ordered DFA A from Figure 2.

Next let us argue, why no biautomaton for the language $\Sigma^* ab \Sigma^*$ can be ordered. Therefore consider some biautomaton $B = (Q, \Sigma, \cdot, \circ, q_0, F)$ with $L(B) = \Sigma^* ab \Sigma^*$. In the following we use the notation $[u.v]$, for $u,v \in \Sigma^*$, to describe the state $(q_0 \cdot u) \circ v$ of B. Of course, different word pairs $[u.v]$ and $[u'.v']$ may describe the same state. First note that the three states $[\lambda.\lambda]$, $[a.\lambda]$, and $[\lambda.b]$ must be pairwise distinct, because every one of these states leads to an accepting state on a different input string. Assume there is some order $\leq$ on the state set Q that is compatible with the transition functions of B. There are six different possibilities to order the three above mentioned states:

$$\begin{array}{lll} [\lambda.\lambda] \leq [a.\lambda] \leq [\lambda.b], & [a.\lambda] \leq [\lambda.\lambda] \leq [\lambda.b], & [\lambda.b] \leq [a.\lambda] \leq [\lambda.\lambda], \\ [\lambda.\lambda] \leq [\lambda.b] \leq [a.\lambda], & [a.\lambda] \leq [\lambda.b] \leq [\lambda.\lambda], & [\lambda.b] \leq [\lambda.\lambda] \leq [a.\lambda]. \end{array}$$

If $[\lambda.\lambda] \leq [a.\lambda] \leq [\lambda.b]$ then it must be $[\lambda.b^i] \leq [a.b^i] \leq [\lambda.b^{i+1}]$, for all $i \geq 0$. Since the number of states in Q is finite, there must be integers $j > k \geq 0$ for which $[\lambda.b^k] = [\lambda.b^j]$. It then follows that $[\lambda.b^k] = [a.b^k] = [\lambda.b^{k+1}]$. This is a contradiction because $[a.b^k]$ describes an accepting state, while $[\lambda.b^k]$ and $[\lambda.b^{k+1}]$ describe non-accepting states.

If $[\lambda.\lambda] \leq [\lambda.b] \leq [a.\lambda]$ then we obtain $[a^i.\lambda] \leq [a^i.b] \leq [a^{i+1}.\lambda]$ for all $i \geq 0$. Similar to the case above we get a contradiction: because Q is finite there is an integer $k \geq 0$ with $[a^k.\lambda] = [a^k.b] = [a^{k+1}.\lambda]$, but the state $[a^k.b]$ is accepting while the other two states are non-accepting.

Next consider the case $[a.\lambda] \leq [\lambda.\lambda] \leq [\lambda.b]$. By reading symbol a with a forward transition we obtain $[a.\lambda] \leq [a.b]$ from the second inequality, and by reading b with a backward transition, the first inequality implies $[a.b] \leq [\lambda.b]$. Note that both times $[a.b]$ describes the same state because B has the $\diamond$-property. Further, this state must be different from the three states $[a.\lambda]$, $[\lambda.\lambda]$, and $[\lambda.b]$ because it is an accepting state, and the others are not. Now there are two possibilities for the placement of state $[a.b]$ in the order:

$$[a.\lambda] \leq [a.b] \leq [\lambda.\lambda] \leq [\lambda.b] \quad \text{or} \quad [a.\lambda] \leq [\lambda.\lambda] \leq [a.b] \leq [\lambda.b].$$

The first case implies $[a^{i+1}.\lambda] \leq [a^{i+1}.b] \leq [a^i.\lambda]$, for all $i \geq 0$, which leads to the contradictory equation $[a^{k+1}.\lambda] = [a^{k+1}.b] = [a^k.\lambda]$, for some $k \geq 0$. The second case implies $[\lambda.b^i] \leq [a.b^{i+1}] \leq [\lambda.b^{i+1}]$, for all $i \geq 0$, and to the contradiction $[\lambda.b^k] = [a.b^{k+1}] = [\lambda.b^{k+1}]$, for some $k \geq 0$.

With similar argumentation, the remaining three cases $[a.\lambda] \leq [\lambda.b] \leq [\lambda.\lambda]$, $[\lambda.b] \leq [a.\lambda] \leq [\lambda.\lambda]$, and $[\lambda.b] \leq [\lambda.\lambda] \leq [a.\lambda]$ lead to contradictions—we omit the details. This shows that there is no order of the state set Q that is compatible with the transition functions of B. Therefore, the ordered language $\Sigma^* ab \Sigma^*$ is not a bi-ordered language. □

In the proof of the following result we use the lexicographic order $<_{\text{lex}}$ of words, which is defined as follows. Let Σ be an alphabet of size k and fix some order $a_1, a_2, \ldots, a_k$ of the symbols from Σ. For two words $w_1, w_2 \in \Sigma^*$ let $w_1 <_{\text{lex}} w_2$ if and only if either w_1 is a prefix of w_2, or $w_1 = u a_i w_1'$ and $w_2 = u a_j w_2'$, for some words $u, w_1', w_2' \in \Sigma^*$ and symbols $a_i, a_j \in \Sigma$, with $i < j$.

Theorem 12 *The class of finite and co-finite languages is strictly contained in the class of bi-ordered languages.*

Proof: Let L be some finite language over the alphabet Σ and let ℓ be the length of the longest word in L. We construct an ordered biautomaton for L as follows. Let $A = (Q, \Sigma, \cdot, \circ, q_0, F)$ be the biautomaton with state set $Q = \{\, (u,v) \mid u, v \in \Sigma^*, |uv| \leq \ell \,\} \cup \{s\}$, initial state $q_0 = (\lambda, \lambda)$, set of accepting states $F = \{\, (u,v) \in Q \mid uv \in L \,\}$, and where the transition functions $\cdot$ and $\circ$ are defined as follows: for all symbols $a \in \Sigma$ let $s \cdot a = s \circ a = s$, and for all states $(u,v) \in Q$ let

$$(u,v) \cdot a = \begin{cases} (ua, v) & \text{if } |uav| \leq \ell, \\ s & \text{otherwise,} \end{cases} \qquad (u,v) \circ a = \begin{cases} (u, av) & \text{if } |uav| \leq \ell, \\ s & \text{otherwise.} \end{cases}$$

One readily sees that A has both the $\diamond$-property, and the F-property. Now let us define the order $\leq$ on Q as follows. First of all let $(u,v) \leq s$, for all $(u,v) \in Q$, so the non-accepting sink state is the largest element of Q. Next, for two different states (u_1, v_1) and (u_2, v_2) let $(u_1, v_1) \leq (u_2, v_2)$ if and only if

- $|u_1v_1| < |u_2v_2|$, or
- $|u_1v_1| = |u_2v_2|$, and $|u_1| < |u_2|$, or
- $|u_1v_1| = |u_2v_2|$, $|u_1| = |u_2|$, and $u_1 <_{\text{lex}} u_2$, or
- $|u_1v_1| = |u_2v_2|$, $u_1 = u_2$, and $v_1 <_{\text{lex}} v_2$.

Notice that if none of the four cases above holds, then $(u_1,v_1) = (u_2,v_2)$. It remains to show that the transitions of A respect the order $\leq$. Since state s goes to itself on every symbol, because it is the largest element, we have $(u,v)\cdot a \leq s\cdot a$, and $(u,v)\circ a \leq s\circ a$. Next let (u_1,v_1) and (u_2,v_2) be two different states of A with $(u_1,v_1) \leq (u_2,v_2)$. Then it must be $|u_1v_1| \leq |u_2v_2| \leq \ell$. If $|u_2v_2| = \ell$, then (u_2,v_2) goes to the sink state s on both the forward, and the backward a-transition. Since s is the largest element we obtain $(u_1,v_1)\cdot a \leq (u_2,v_2)\cdot a$, and $(u_1,v_1)\circ a \leq (u_2,v_2)\circ a$. Therefore, in the following argumentation we assume that $|u_2,v_2| < \ell$, so that we have $(u_1,v_1)\cdot a = (u_1a,v_1)$ and $(u_2,v_2)\cdot a = (u_2a,v_2)$, as well as $(u_1,v_1)\circ a = (u_1,av_1)$ and $(u_2,v_2)\circ a = (u_2,av_2)$. Now we have to show $(u_1a,v_1) \leq (u_2a,v_2)$ and $(u_1,av_1) \leq (u_2,av_2)$, for which we distinguish four cases.

- If $|u_1v_1| < |u_2v_2|$ then clearly $|u_1av_1| < |u_2av_2|$, from which we conclude $(u_1a,v_1) \leq (u_2a,v_2)$, and $(u_1,av_1) \leq (u_2,av_2)$.
- If $|u_1v_1| = |u_2v_2|$, and $|u_1| < |u_2|$, then also $|u_1a| < |u_2a|$. Again, we can conclude $(u_1a,v_1) \leq (u_2a,v_2)$, and $(u_1,av_1) \leq (u_2,av_2)$.
- Next assume $|u_1v_1| = |u_2v_2|$, $|u_1| = |u_2|$, and $u_1 <_{\text{lex}} u_2$. Since u_1 and u_2 are of same length, the fact $u_1 <_{\text{lex}} u_2$ implies $u_1a <_{\text{lex}} u_2a$. Thus, we get $(u_1a,v_1) \leq (u_2a,v_2)$, and $(u_1,av_1) \leq (u_2,av_2)$.
- Finally let $|u_1v_1| = |u_2v_2|$, $|u_1| = |u_2|$, $u_1 = u_2$, and $v_1 <_{\text{lex}} v_2$. From $v_1 <_{\text{lex}} v_2$ follows $av_1 <_{\text{lex}} av_2$, so we obtain $(u_1a,v_1) \leq (u_2a,v_2)$, and $(u_1,av_1) \leq (u_2,av_2)$.

This shows that the biautomaton A is an ordered biautomaton.

In case of a co-finite language $L \subseteq \Sigma^*$ we first take its complement $\Sigma^* \setminus L$, which is finite, and apply the above given construction. Then we obtain an ordered biautomata A. Finally, exchanging accepting and non-accepting states—this is the ordinary complementation construction known for DFAs applied to DBiAs—results in an ordered biautomata for the language L.

Finally, strictness of the inclusion is witnessed by the infinite and not co-finite language $a^* + b$, which is accepted by the bi-ordered biautomaton from Figure 3. □

Notice that the language $\Sigma^*ab\Sigma^*$ from the proof of Theorem 11 is even a strictly ordered language, since its minimal DFA A from Figure 2 is ordered. As we have seen, this language is not a bi-ordered language, therefore the class of bi-ordered languages does not even contain all strictly ordered languages. On the other hand, if a language is strictly bi-ordered, i.e., if its minimal biautomaton B is ordered, then also the minimal DFA B_{fwd} is ordered. Therefore, the class of strictly bi-ordered languages is contained in the classes of strictly ordered languages. We summarize our findings in the following corollary.

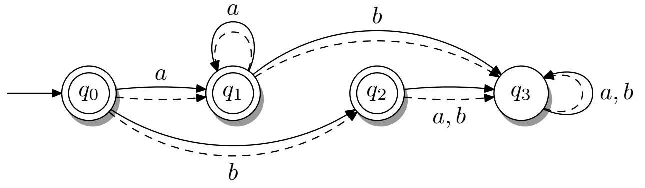


Figure 3: A bi-ordered biautomaton with order $q_0 \leq q_1 \leq q_2 \leq q_3$ for the language $a^* + b$.

Corollary 13 *The class of strictly bi-ordered languages is proper subset of the class of strictly ordered languages.* □

Concerning the relation between bi-ordered languages and strictly ordered languages, we can see that these are incomparable to each other. We have seen that the strictly ordered language $\Sigma^*ab\Sigma^*$ is not bi-ordered. On the other hand, we know that every finite language is bi-ordered. But one can see that the minimal DFA for the finite language $\{ab\}$ is not ordered—the reader is invited convince himself of this fact. A proof of a more general result, saying that a single word language is strictly ordered if and only if the word is of the form a^i for some alphabet symbol a and integer $i \geq 0$, can be found in [18].

Corollary 14 *The classes of bi-ordered languages and of strictly ordered languages are incomparable to each other.* □

Moreover, with a similar argumentation as above we obtain:

Corollary 15 *The class strictly bi-ordered languages is a proper subset of the class of bi-ordered languages.* □

6 Non-Exiting and Non-Returning Automata

In this last section we study so called non-exiting automata and non-returning automata. A biautomaton or finite automaton A is *non-exiting* if all outgoing transitions from accepting states go to a non-accepting sink state, and it is *non-returning* if the initial state does not have any ingoing transitions. We say that A is *exiting* if it is *not* non-exiting, and it is *returning* if it is *not* non-returning.

In the DFA case non-exiting automata are known to characterize the class of prefix-free languages, while non-returning automata are related to suffix-free languages. However this latter relation is not a characterization: it is true that every DFA that accepts a (non-empty) suffix-free language must be non-returning, but the reverse implication does not hold.

Concerning the situation for biautomata, we first show that every biautomaton which is non-exiting must also be non-returning (unless it accepts the empty language).

Lemma 16 *Let A be a biautomaton with $L(A) \neq \emptyset$. If A is non-exiting, then A is non-returning.*

Proof: We prove the contraposition of the lemma. Assume $A = (Q, \Sigma, \cdot, \circ, q_0, F)$ is a returning biautomaton. Then there must be words $u, v \in \Sigma^*$, with $|uv| \geq 1$, such that $(q_0 \cdot u) \circ v = q_0$. It follows that $(q_0 \cdot u^n) \circ v^n = (q_0 \circ v^n) \cdot u^n = q_0$, for all $n \geq 0$. Since the number of states in A is finite, there are integers $i, j \geq 0$ and $x, y \geq 1$ such that $q_0 \cdot u^i = q_0 \cdot u^{i+x}$ and $q_0 \circ v^j = q_0 \circ v^{j+y}$. We obtain $q_0 = (q_0 \cdot u^{i+x}) \circ v^i$ and $q_0 = (q_0 \circ v^{j+y}) \cdot u^j$. Now let $w \in L(A)$, i.e., $q_0 \cdot w \in F$. From our considerations above we get $((q_0 \cdot u^i) \circ v^i) \cdot w \in F$ and

$$((q_0 \cdot u^i) \circ v^i) \cdot w = ((q_0 \cdot u^{i+x}) \circ v^i) \cdot w = (((q_0 \cdot u^i) \circ v^i) \cdot w) \cdot u^x \in F.$$

Recall that $|uv| \geq 1$. If $|u| \geq 1$, then we see that A is exiting because the accepting state $((q_0 \cdot u^{i+x}) \circ v^i) \cdot w$ cannot go to a non-accepting sink state on every input symbol. If $|u| = 0$ then it must be $|v| \geq 1$. Now a similar argumentation gives $((q_0 \circ v^j) \cdot u^j) \cdot w \in F$ and

$$((q_0 \circ v^j) \cdot u^j) \cdot w = ((q_0 \circ v^{j+y}) \cdot u^j) \cdot w = (((q_0 \circ v^j) \cdot u^j) \cdot w) \circ v^y \in F.$$

Here the accepting state $((q_0 \circ v^j) \cdot u^j) \cdot w$ cannot go to a non-accepting sink state on every alphabet symbol, which shows that A is exiting. □

The converse of Lemma 16 is not true which can easily be seen by the minimal biautomaton for the language $\{a, aa\}$. Since the language is finite, there cannot be a cycle $q_0 = (q_0 \cdot u) \circ v$, hence the biautomaton is non-returning. However, since both states $q_0 \cdot a$ and $(q_0 \cdot a) \cdot a$ are accepting, the automaton cannot be non-exiting.

Now we study the classes of languages accepted by biautomaton that are non-exiting or non-returning. While minimal non-exiting DFAs characterize the class of prefix-free languages, we show in the following that minimal non-exiting biautomata characterize a different language class, namely the class of circumfix-free languages. A word $v \in \Sigma^*$ is a *circumfix* of a word $w \in \Sigma^*$, if $w = w_1w_2w_3$ and $v = w_1w_3$, for some words $w_1, w_2, w_3 \in \Sigma^*$. A language L is called *circumfix-free* if there are no two *different* words $w, v \in L$, such that v is a circumfix of w. Notice that prefixes and suffixes of a word are also circumfixes (where one "side" is λ). Therefore the class of circumfix-free languages is contained in both the classes of prefix-free languages and suffix-free languages.

Theorem 17 *A regular language is circumfix-free if and only if its* minimal *biautomaton is non-exiting.*

Proof: Let $A = (Q, \Sigma, \cdot, \circ, q_0, F)$ be a minimal biautomaton and $L = L(A)$. First assume that A is exiting, i.e., there is an accepting state $q \in F$ and a non-empty word $w \in \Sigma^+$ such that $q \cdot w \in F$. Since q must be reachable from the initial state of A, there are words $u, v \in \Sigma^*$ with $(q_0 \cdot u) \circ v = q$. Then both words uv and uwv belong to $L(A)$, but uv is a circumfix of w. Therefore, if $L(A)$ is circumfix-free then A must be non-exiting.

For the reverse implication notice that whenever there are two different words w and w' in $L(A)$ such that w' is a circumfix of w, then $w = w_1w_2w_3$ and $w' = w_1w_3$, with $w_1, w_3 \in \Sigma^*$ and $w_2 \in \Sigma^+$ (because $w \neq w'$). It follows $(q_0 \cdot w_1) \circ w_3 \in F$ and $((q_0 \cdot w_1) \circ w_3) \cdot w_2 \in F$, and since $w_2 \neq \lambda$ the automaton A must be exiting. Thus, if A is non-exiting then $L(A)$ must be circumfix-free. □

Now we consider languages accepted non-returning biautomata. If a minimal biautomaton A is non-returning then clearly the contained minimal DFA A_{fwd} is non-returning, too. Therefore the class of languages accepted by minimal non-returning biautomata is contained in the class of languages accepted by minimal non-returning DFAs. Moreover, this inclusion is strict because the minimal DFA for the language ab^* is non-returning, while the minimal biautomaton for that language is not (it has a backward transition loop for symbol b on its initial state). Therefore we have the following result.

Theorem 18 *The class of languages accepted by* minimal *non-returning biautomata is strictly contained in the class of languages accepted by* minimal *non-returning deterministic finite automata.* □

7 Conclusions

We continued the study of structural properties on biautomata started in [7, 11, 12]. Our focus was on the effect of classical properties of deterministic finite automata such as, e.g., permutation-freeness, strongly permutation-freeness, and orderability, on biautomata. It is shown that this approach on structurally restricting the recently introduced biautomata model was worth looking at. A comparison of the induced language families on structurally restricted deterministic automata and biautomata is given in Table 1. Future research on the subject under consideration may consist on some further properties such as, e.g., biautomata where all states are final or all are initial. In the case of ordinary deterministic finite automata the family of prefix-closed languages is obtained by the former property, while the latter gives the family of suffix-closed languages. Moreover, it would be also interesting to study, which structural properties can be successfully applied to nondeterministic biautomata, as introduced in [8].

References

[1] M. A. Arbib (1969): *Theories of Abstract Automata*. Automatic Computation, Prentice-Hall, London.

[2] A. Badr, V. Geffert & I. Shipman (2009): *Hyper-Minimizing Minimized Deterministic Finite State Automata*. RAIRO–Informatique théorique et Applications / Theoretical Informatics and Applications 43(1), pp. 69–94. doi:10.1051/ita:2007061

[3] J. Brzozowski & B. Liu (2021): *Syntactic Complexity of Finite/Cofinite, Definite, and Reverse Definite Languages*. arXiv:1203.2873v1 [cs.FL].

[4] J. A. Brzozowski & F. E. Fitch (1980): *Languages of $\mathscr{R}$-Trivial Monoids*. Journal of Computer and System Sciences 20(1), pp. 32–49. doi:10.1016/0022-0000(80)90003-3

[5] J.-M. Champarnaud, J.-P. Dubernard, H. Jeanne & L. Mignot (2013): *Two-Sided Derivatives for Regular Expressions and for Hairpin Expressions*. In A. H. Dediu, C. Martín-Vide & B. Truthe, editors: Proc. of the 7th International Conference on Language and Automata Theory and Applications, LNCS 7810, Springer, Bilbao, Spain, pp. 202–213. doi:10.1007/978-3-642-37064-9_19

[6] I. M. Havel (1969): *The theory of regular events II*. Kybernetika 6, pp. 520–544.

[7] M. Holzer & S. Jakobi (2013): *Minimization and Characterizations for Biautomata*. In S. Bensch, F. Drewes, R. Freund & F. Otto, editors: Proc. of the 5th International Workshop on Non-Classical Models of Automata and Applications, books@ocg.at 294, Österreichische Computer Gesellschaft, Umeå, Sweden, pp. 179–193.

[8] M. Holzer & S. Jakobi (2013): *Nondeterministic Biautomata and Their Descriptional Complexity*. In H. Jürgensen & R. Reis, editors: Proc. of the 15th International Workshop on Descriptional Complexity of Formal Systems, LNCS 8031, Springer, London, Ontario, Canada, pp. 112–123. doi:10.1007/978-3-642-39310-5_12

[9] M. Holzer & S. Jakobi (2014): *Minimal and Hyper-Minimal Biautomata*. IFIG Research Report 1401, Institut für Informatik, Justus-Liebig-Universität Gießen, Arndtstr. 2, D-35392 Gießen, Germany.

[10] G. Jirásková & O. Klíma (2012): *Descriptional Complexity of Biautomata*. In M. Kutrib, N. Moreira & R. Reis, editors: Proc. of the 14th International Workshop Descriptional Complexity of Formal Systems, LNCS 7386, Springer, Braga, Portugal, pp. 196–208. doi:10.1007/978-3-642-31623-4_15

[11] O. Klíma & L. Polák (2012): *Biautomata for k-Piecewise Testable Languages*. In H.-C. Yen & O. H. Ibarra, editors: Proc. of the 16th International Conference Developments in Language Theory, LNCS 7410, Springer, Taipei, Taiwan, pp. 344–355. doi:10.1007/978-3-642-31653-1_31

[12] O. Klíma & L. Polák (2012): *On Biautomata*. RAIRO–Informatique théorique et Applications / Theoretical Informatics and Applications 46(4), pp. 573–592. doi:10.1051/ita/2012014

[13] R. Loukanova (2007): *Linear Context Free Languages*. In C. B. Jones, Z. Liu & J. Woodcock, editors: Proc. of the 4th International Colloquium Theoretical Aspects of Computing, LNCS 4711, Springer, Macau, China, pp. 351–365.

[14] R. McNaughton & S. Papert (1971): *Counter-free automata*. Research monographs 65, MIT Press.

[15] M. L. Minsky (1967): *Computation: Finite and Infinite Machines*. Automatic Computation, Prentice-Hall.

[16] M. Perles, M. O. Rabin & E. Shamir (1963): *The Theory of Definite Automata*. IEEE Transactions on Electronic Computers EC-12(3), pp. 233–243. doi:10.1109/PGEC.1963.263534

[17] A. L. Rosenberg (1967): *A Machine Realization of the Linear Context-Free Languages*. Information and Control 10, pp. 175–188. doi:10.1016/S0019-9958(67)80006-8

[18] H.-J. Shyr & G. Thierrin (1974): *Ordered Automata and Associated Languages*. Tamkang Journal of Mathematics 5(1).

[19] I. Simon (1975): *Piecewise Testable Events*. In H. Brakhage, editor: Proc. of the 2nd GI Conference on Automata Theory and Formal Languages, LNCS 33, Springer, Kaiserslautern, Germany, pp. 214–222.

[20] G. Thierrin (1968): *Permutation Automata*. Mathematical Systems Theory 2(1), pp. 83–90. doi:10.1007/BF01691347

Buffered Simulation Games for Büchi Automata

Milka Hutagalung, Martin Lange and Etienne Lozes
School of Electr. Eng. and Computer Science, University of Kassel, Germany*

Simulation relations are an important tool in automata theory because they provide efficiently computable approximations to language inclusion. In recent years, extensions of ordinary simulations have been studied, for instance multi-pebble and multi-letter simulations which yield better approximations and are still polynomial-time computable.

In this paper we study the limitations of approximating language inclusion in this way: we introduce a natural extension of multi-letter simulations called buffered simulations. They are based on a simulation game in which the two players share a FIFO buffer of unbounded size. We consider two variants of these buffered games called continuous and look-ahead simulation which differ in how elements can be removed from the FIFO buffer. We show that look-ahead simulation, the simpler one, is already PSPACE-hard, i.e. computationally as hard as language inclusion itself. Continuous simulation is even EXPTIME-hard. We also provide matching upper bounds for solving these games with infinite state spaces.

1 Introduction

Nondeterministic Büchi automata (NBA) are an important formalism for the specification and verification of reactive systems. While they have originally been introduced as an auxiliary device in the quest for a decision procedure for Monadic Second-Order Logic [4] they are by now commonly used in such applications as LTL software model-checking [13, 21], or size-change termination analysis for recursive programs [24, 17]. Typical decision procedures from these domains then reduce to automata-theoretic decision problems like emptiness or inclusion for instance [30].

While emptiness for Büchi automata is NLOGSPACE-complete, deciding inclusion between two nondeterministic finite automata is already more difficult, namely PSPACE-complete [25]. This is also the complexity of inclusion for NBA. Thus, it is – given current knowledge – exponential in the size of the involved NBA regardless of whether it is solved using explicit complementation [27, 29, 23] or other means [1, 16]. One major issue of automata manipulation is therefore to keep the number of states as small as possible.

Since the early works of Dill et al [11], simulations have been intensively used in automata-based verification. Unlike the PSPACE-hard problems like inclusion, simulation between two NBA is cheap to compute. Simulations are interesting with respect to several aspects. On the one hand, they offer a sound, but incomplete, approximation of language inclusion that may be sufficient in many practical cases. On the other hand, simulations can be used for quotienting automata [5, 18, 14], for pruning transitions [1, 2], or for improving existing decision procedures on NBA like the Ramsey-based [17] or the antichain algorithm for inclusion, resp. universality checking [12].

There is a simple game-theoretic characterisation of simulation between two NBA: two players called Spoiler and Duplicator move two pebbles on the transition graph of the NBA, each of them controls one pebble. In order to decide whether or not an NBA $\mathscr{A}$ is simulated by an NBA $\mathscr{B}$, Spoiler starts with his

*The European Research Council has provided financial support under the European Community's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement no 259267.

pebble on the initial state of $\mathscr{A}$ and moves it along a transition labeled with some alphabet symbol a. Duplicator starts with her pebble on the initial state of $\mathscr{B}$ and responds with a move along a transition labeled with the same letter. This proceeds ad infinitum. There are different kinds of simulation depending on the winning conditions in these games. For instance, fair simulation models the Büchi acceptance condition and requires Duplicator to have visited infinitely often accepting states if Spoiler has done so. While it is close to the actual condition on inclusion between these two automata, quotienting automata with respect to fair simulation does not preserve the automaton's language.

It is therefore that different winning conditions like delayed simulation have been invented which require Duplicator to eventually visit an accepting states whenever Spoiler has visited one [14]. They, however, do not necessarily provide better approximations to language inclusion. Extensions of the plain simulation relation have been considered since, in particular multi-pebble [15] and multi-letter simulations [9, 22]. Both try to alleviate the gap between simulation and language inclusion which shows up in the game-theoretic characterisation as Spoiler being too strong: language inclusion would correspond to a game in which player chooses an entire run in $\mathscr{A}$ and then Duplicator produces one in $\mathscr{B}$ on the same word. In the simulation game, Spoiler reveals his run step-wise and can therefore dupe Duplicator into positions from which she cannot win anymore even though language inclusion holds.

The two extensions – multi-pebble and multi-letter simulation – use different approaches to approximate language inclusion better: multi-pebble simulation add a certain degree of imperfectness to these games by allowing Duplicator to be in several positions at the same time. Multi-letter simulation forces Spoiler to reveal more of his runs and therefore allows Duplicator to delay her choices for a few rounds and therefore benefit from additional information she gained about Spoiler's moves. The complexity of computing these extended simulations has been studied before: both are polynomial for a fixed number of pebbles, respectively a fixed look-ahead in the multi-letter games. However, nothing is known about the complexity of these simulations if the number of pebbles/letters is not fixed.

Contribution. This paper studies a natural extension of multi-letter games to unbounded look-aheads. We introduce a new family of simulation relations for Büchi automata, called buffered simulations. In a buffered simulation, Spoiler and Duplicator move two pebbles along automata transitions, but unlike in standard simulations, Spoiler and Duplicator's moves do not always alternate. Indeed, Duplicator can "skip her turn" and wait to see Spoiler's next moves before responding. Spoiler and Duplicator share a first-in first-out buffer: every time Spoiler moves along an a-labelled transition, he adds an a into the buffer, whereas every time Duplicator makes a step along a b-labelled transition, she removes a b from the buffer. Since Duplicator has more chances to defeat Spoiler than in standard simulations, buffered simulations better approximate language inclusion. They also improve multi-letter simulations, and it is thus a natural question to ask if they are polynomial time decidable and could be used in practice.

We study two notions of buffered simulation games, called continuous and look-ahead simulation games, respectively. Their rules only differ in the way that Duplicator must use the buffer: in look-ahead simulations, Duplicator is forced to flush the buffer, so that she "catches up" with Spoiler every time she decides to make a move. Thus, the buffer is flushed completely with each of Duplicator's moves. In the continuous case, Duplicator can choose to only consume a part of the buffer with every move, and it need not ever be flushed.

We show that these unbounded buffer simulation games – whilst naturally extending the "easy" multi-letter simulations – provide in a sense a limit to the efficient approximability of language inclusion: we show that look-ahead simulations are already PSPACE-hard, i.e. as difficult as language inclusion itself, while continuous simulations are even worse: they are EXPTIME-hard, i.e. presumably even more difficult than language inclusion.

We also provide matching upper bounds in order to show that these lower bounds are tight, i.e. these simulations problems are not worse than that. In particular, look-ahead simulation is therefore as difficult as language inclusion, and continuous simulation is "only" slightly more difficult. Decidability of these simulations is not obvious. In the finitary cases, it is provided by a rather straight-forward reduction to parity games but games with unbounded buffers would yield parity games of infinite size. Moreover, questions about systems with unbounded FIFO buffers are often undecidable; for instance, linear-time properties of a system of two machines and one buffer are known to be undecidable [6]. Decidability of these simulation relations may therefore be seen as surprising, and it is also not inconceivable that the decidability results for these unbounded FIFO buffer simulations may lead to developments in other areas, for instance reachability in infinite-state systems etc.

Outline. Section 2 first recalls Büchi automata and ordinary simulation relations. It then introduces continuous simulation as a simulation game extended with an unbounded buffer. Look-ahead simulation is obtained by restricting the use of the buffer in a natural way. Section 3 contains the most important results in these relations: it shows that look-ahead simulation is already as hard as language inclusion whereas continuous simulation is even harder. Section 4 shows that these bounds are tight by introducing a suitable abstraction called quotient game which yields corresponding upper bounds. Finally, Section 5 collects further interesting results on these simulation relations like topological characterisations for instance and concludes with comments on their use in automata minimisation.

2 Extended Simulation Relations

2.1 Background

Nondeterministic Büchi Automata. A non-deterministic Büchi automaton (NBA) is a tuple $\mathscr{A} = (Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states with q_0 being a designated starting state, $\delta \subseteq Q \times \Sigma \times Q$ is a transition relation, and $F \subseteq Q$ is a set of accepting states. A state $q \in Q$ is called a *dead end* when there is no $a \in \Sigma$ and $q' \in Q$ such that $(q, a, q') \in \delta$. If $w = a_1 \dots a_n$, a sequence $q_0 a_1 q_1 \dots q_n$ is called a w-path from q_0 to q_n if $(q_i, a_{i+1}, q_{i+1}) \in \delta$ for all $i \in \{0, \dots, n-1\}$. It is an *accepting* w-path if there is some $i \in \{1, \dots, n\}$ such that $q_i \in F$. We write $q_0 \xrightarrow{w} q_n$ to state that there is a w-path from q_0 to q_n, and $q_0 \overset{w}{\rightarrow\!\!\!\bullet\!\!\!\rightarrow} q_n$ to state that there is an accepting one.

A *run* of $\mathscr{A}$ on a word $w = a_1 a_2 \dots \in \Sigma^\omega$ is an infinite sequence $\rho = q_0 q_1 \dots$ such that $(q_i, a_{i+1}, q_{i+1}) \in \delta$ for all $i \geq 0$. The run is *accepting* if there is some $q \in F$ such that $q = q_i$ for infinitely many i. The language of $\mathscr{A}$ is the set $L(\mathscr{A})$ of infinite words for which there exists an accepting run.

Fair Simulation.

Fair simulation [19] is an extension of standard simulation to Büchi automata. The easiest way of defining fair simulation is by means of a game between two players called *Spoiler* and *Duplicator*. Let us fix two NBA $\mathscr{A} = (Q, \Sigma, \delta, q_I, F)$ and $\mathscr{B} = (Q', \Sigma, \delta', q'_I, F')$. Spoiler and Duplicator are each given a pebble that is initially placed on $q_0 := q_I$ for Spoiler and $q'_0 := q'_I$ for Duplicator. Then, on each round $i \geq 1$,

1. Spoiler chooses a letter $a_i \in \Sigma$ and a transition $(q_{i-1}, a_i, q_i) \in \delta$, and moves his pebble to q_i;
2. Duplicator responds by choosing a transition $(q'_{i-1}, a_i, q'_i) \in \delta'$ and moves his pebble to q'_i.

Either the play terminates because one player reaches a dead end, and then the opponent wins the play. Or the game produces two infinite runs $\rho = q_0 a_1 q_1, \ldots$ and $\rho' = q'_0 a_1 q'_1 \ldots$, in which case Duplicator is declared the winner of the play if ρ is not accepting or ρ' is accepting. Otherwise Spoiler wins this play.

We say that $\mathscr{A}$ is *fairly simulated* by $\mathscr{B}$, written $\mathscr{A} \sqsubseteq^{\mathsf{f}} \mathscr{B}$, if Duplicator has a winning strategy for this game. Clearly, $\mathscr{A} \sqsubseteq^{\mathsf{f}} \mathscr{B}$ implies $L(\mathscr{A}) \subseteq L(\mathscr{B})$, but the converse does not hold in general.

Remark 2.1. Notice that standard simulation, as defined for labelled transition systems, is a special case of fair simulation. Indeed, for a given labelled transition system (Q, Σ, δ), and a given state q, we can define the NBA $\mathscr{A}(q)$ with $q_I := q$ as the initial state, and $F := Q$ as the set of accepting states. Then q' simulates q in the standard sense (without taking care of fairness) if and only if $\mathscr{A}(q) \sqsubseteq^{\mathsf{f}} \mathscr{A}(q')$. We write $q \sqsubseteq q'$ when q' simulates q in the standard sense.

2.2 Continuous Simulation

Continuous simulations are defined by games in which Duplicator is allowed to see in advance some finite but *unbounded* number of Spoiler's moves. This naturally extends recent work on extensions of fair simulation called *multi-letter* or *look-ahead simulations* in which Duplicator is allowed to see a number of Spoiler's moves that is *bounded* by a constant [22, 9].

Let $\mathscr{A} = (Q, \Sigma, \delta, q_I, F)$ and $\mathscr{B} = (Q', \Sigma, \delta', q'_I, F')$ be two NBA. In the continuous fair simulation game, Spoiler and Duplicator now share a FIFO buffer β and move two pebbles through the automata's state spaces. The positions of the pebbles form a word w and two runs ρ and ρ', obtained by successively extending sequences ρ_i and ρ'_i in each round i with zero or more states. At the beginning we have $\rho_0 := q_I$ and $\rho'_0 := q'_I$, i.e. Spoiler's pebble is on q_I and Duplicator's pebble is on q'_I. Initially, both word and buffer are empty, i.e. we have $w_0 := \varepsilon$ and $\beta_0 := \varepsilon$.

For the m-th round, with $m \geq 1$ suppose that $w_{m-1} = a_1, \ldots, a_{m-1}$, $\rho_{m-1} = q_0, \ldots, q_{m-1}$, $\rho'_{m-1} = q'_0, \ldots, q_{m'}$ and β_{m-1} have been created already. Duplicator's run in $\mathscr{B}$ is shorter than Spoiler's run, i.e. $m' \leq m$. Furthermore, the buffer β contains the suffix $a_{m'+1}, \ldots, a_m$ of w_{m-1} that Duplicator has not mimicked yet. The m-th round then proceeds as follows.

1. Spoiler chooses a letter $a_m \in \Sigma$ and a transition $(q_{m-1}, a_m, q_m) \in \delta$ and moves the pebble to q_m, i.e. we get $w_m := w_{m-1} a_m$ and $\rho_m := \rho_{m-1} q_{m+1}$. The letter a_m is added to the buffer, i.e. $\beta' := \beta, a_m$.
2. Suppose we now have $\beta' = b_1, \ldots, b_k$. Duplicator picks some r with $0 \leq r \leq k$ as well as states $q'_{m'}, \ldots, q'_{m'+r-1}$ such that $(q_{m'+i-1}, b_i, q_{m'+i}) \in \delta'$ for all $i = 1, \ldots, r$. Then we get $\rho'_m := \rho'_{m-1}, q_{m'+1}, \ldots, q_{m'+r}$. The letters get flushed from the buffer, i.e. $\beta_i := b_{r+1}, \ldots, b_k$.

   Note that we have $\rho'_m = \rho'_{m-1}$ if Duplicator chooses $r = 0$. In this case we also say that she *skips her turn*.

A play of this game defines a finite or infinite run $\rho = q_0, q_1, \ldots$ for Spoiler (finite if Spoiler reaches a dead end), and a finite or infinite run $\rho' = q'_0, q'_1, \ldots$ for Duplicator (finite if Duplicator eventually always skips her turn) on the finite or infinite word $w = a_1 a_2 \ldots$.

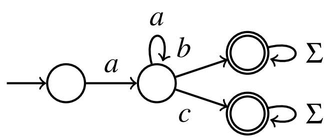
Duplicator is declared the winner of the play if

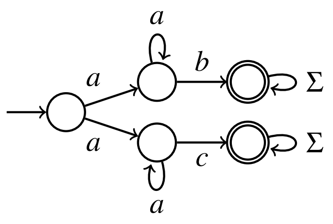
- ρ is finite, or
- ρ is infinite (and w is necessarily infinite as well) and
  - ρ is not an accepting run on w, or
  - ρ' is infinite and an accepting run on w.

In all other cases, Spoiler wins the play.

We say that $\mathscr{B}$ *continuously fairly simulates* $\mathscr{A}$, written $\mathscr{A} \sqsubseteq^{\mathsf{f}}_{\mathsf{co}} \mathscr{B}$, if Duplicator has a winning strategy for the continuous fair simulation game on $\mathscr{A}$ and $\mathscr{B}$. We also consider the (unfair) continuous simulation $\sqsubseteq_{\mathsf{co}}$ for pairs of LTS states by considering an LTS with a distinguished state as an NBA where all states are accepting.

Example 2.2. Consider the following two NBA $\mathscr{A}$ (left) and $\mathscr{B}$ (right) over the alphabet $\Sigma = \{a,b,c\}$.





Clearly, we have $L(\mathscr{A}) \subseteq L(\mathscr{B})$.

Duplicator has a winning strategy for the continuous fair simulation game on this pair of automata: she skips her turns until Spoiler follows either the b- or the c-transition. However, if we ignore the accepting states and consider these automata as a transition system, then Spoiler has a winning strategy for the continuous simulation: he iterates the a-loop, and then either Duplicator waits forever and loses the play, or she makes a move and it is then easy for Spoiler to defeat her.

This example also shows that continuous fair simulation strictly extends multi-letter fair simulation which can be seen as the restriction of the former to a bounded buffer. I.e. in these games, Duplicator can only benefit from a fixed look-ahead of at most k letters for some k. It is not hard to see that Spoiler wins the game with a bounded buffer of length k for any k on these two automata: he simply takes k turns on the a-loop in $\mathscr{A}$ which forces Duplicator to choose a transition out of the initial state in $\mathscr{B}$. After doing so, Spoiler can choose the b- or c-transition that is not present for Duplicator anymore and make her get stuck.

2.3 Look-Ahead Simulations

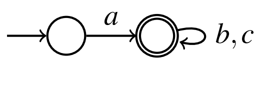
We now consider a variant of the continuous simulation games called *look-ahead* simulation games (the terminology follows [9]). Look-ahead simulation games proceed exactly like the continuous ones, except that now Duplicator has only two possibilities: either she skips her turn, or she flushes the entire buffer. Formally, the definition of the game only differs from the one of Section 2.2 in that the number r of letters removed by Duplicator in a round is either 0 or the size $|\beta|$ of the current buffer β, whereas continuous simulation allowed any $r \in \{0,\ldots,|\beta|\}$.

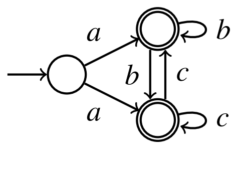
We write $\mathscr{A} \sqsubseteq^{\mathsf{f}}_{\mathsf{la}} \mathscr{B}$ if Duplicator has a winning strategy for the look-ahead fair simulation on the two automata $\mathscr{A},\mathscr{B}$. Similarly, we define the look-ahead fair simulation for LTS, $\sqsubseteq_{\mathsf{la}}$.

Example 2.3. Consider again $\mathscr{A}$ and $\mathscr{B}$ as in Example 2.2. It holds that $\mathscr{A} \sqsubseteq^{\mathsf{f}}_{\mathsf{la}} \mathscr{B}$, because Duplicator can flush the buffer once she has seen the first b or c.

Clearly, look-ahead simulation implies continuous simulation but the converse does not hold.

Example 2.4. Consider the following two NBA $\mathscr{A}$ (left) and $\mathscr{B}$ (right) over the alphabet $\Sigma = \{a,b,c\}$.





Duplicator wins the continuous fair simulation: a winning strategy for Duplicator is to skip her first turn, and then to remove one letter at a time during the rest of the play. Thus, after each round, the buffer always contains exactly one element.

On the other hand, Spoiler wins the look-ahead simulation, because the first time Duplicator flushes the buffer, she has committed to a choice between the two right states and thus makes a prediction about the next letter that Spoiler will play.

Remark 2.5. Multi-pebble simulations [15] are another notion of simulation in which duplicator is given more than just one pebble, which she can move, duplicate, and drop during the game. If the number of such pebbles is not bounded, multi-pebble simulations better approximate language inclusion than continuous and look-ahead simulation; in particular, the look-ahead simulation game corresponds to the multi-pebble simulation game in which duplicator is required to drop all but one pebble infinitely often.

3 Lower Bounds: The Complexity of Buffered Simulations

The difficulty of deciding continuous and look-ahead simulation is shown by reduction from suitable tiling problems.

Definition 3.1. A tiling system is a tuple $\mathscr{T} = (T,H,V,t_I,t_F)$, where T is a set of tiles, $H,V \subseteq T \times T$ are the horizontal and vertical compatibility relations, $t_I,t_F \in T$ are the initial and final tiles.

Let n,m be two natural numbers. A tiling with n columns and m rows according to $\mathscr{T}$ is a function $t : \{1,\dots,n\} \times \{1,\dots,m\} \to T$; the tiling is valid if (1) $t_{1,1} = t_I$ and $t_{n,m} = t_F$, (2) for all $i = 1,\dots,n-1$ and all $j = 1,\dots,m$ we have $(t_{i,j},t_{i+1,j}) \in H$, (3) for all $i = 1,\dots,n$, for all $j = 1,\dots,m-1$ we have $(t_{i,j},t_{i,j+1}) \in V$.

The problem of deciding whether there exists a valid tiling with n columns and 2^n rows, for a given n in unary and a tiling system $\mathscr{T}$, is known to be PSPACE-hard [3].[0] Clearly, the problem to decide whether there is no such tiling is equally PSPACE-hard. We reduce the complement of the tiling problem to look-ahead buffered simulation.

Theorem 3.2. *Deciding* $\sqsubseteq_{\mathsf{la}}$ *(resp.* $\sqsubseteq^{\mathsf{f}}_{\mathsf{la}}$*) is PSPACE-hard.*

Proof. Given a tiling system $\mathscr{T} = (T,H,V,t_I,t_F)$ and an $n \in \mathbb{N}$, we consider the alphabet $\Sigma := (T \times \{0,1\}) \cup \{\$,\#\}$. We define the two automata $\mathscr{A}$, $\mathscr{B}$ as depicted on Figure 1, where all states are accepting. The sizes of $\mathscr{A}$, $\mathscr{B}$ are polynomial in $|T|+n$. Let us consider first the automaton $\mathscr{A}$. A word accepted by $\mathscr{A}$ is composed of blocks of n tiles separated by the \$ symbol, such that each block is tagged with the binary representation of a number in $\{0,\dots,2^n-1\}$. We take as a convention that the first bit is the least significant one. Either the word contains finitely many blocks, in which case, the word ends with the symbol # repeated infinitely often, or it contains infinitely many blocks. Moreover, the first block is tagged with 0, and the last one, if it exists, is tagged with 2^n-1 and it is the only one that may be tagged with 2^n-1.

Consider now the automaton $\mathscr{B}$. From state q_0, the automaton accepts a word if the two first blocks are not tagged with consecutive numbers. From the state q_i, the automaton accepts a word if either it starts with a tile that is not horizontally compatible with t_i, or if after n symbols it contains a tile that is not vertically compatible with t_i.

[0]The requirement on the final tile for instance is not needed for PSPACE-hardness but this variant of the tiling problem is most convenient for the reductions presented here.

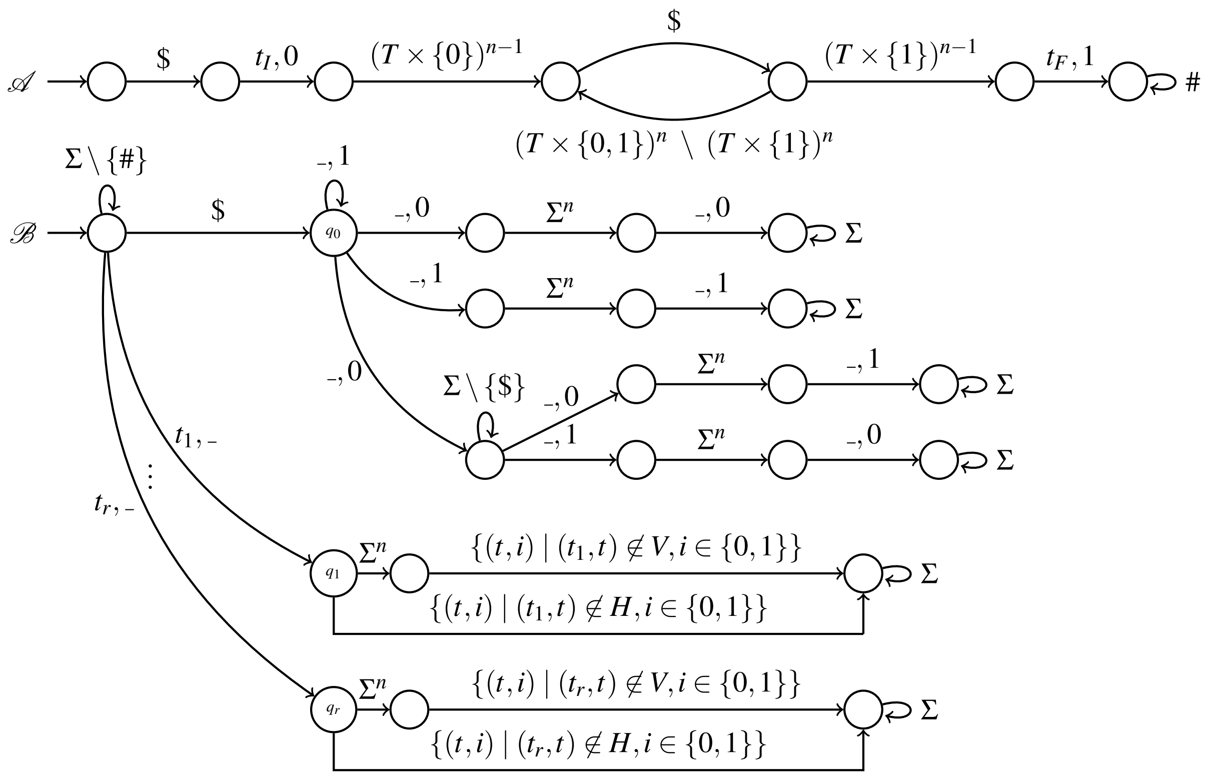

$\mathscr{A}$
$\$$
$t_I,0$
$(T\times\{0\})^{n-1}$
$\$$
$(T\times\{0,1\})^n \setminus (T\times\{1\})^n$
$(T\times\{1\})^{n-1}$
$t_F,1$
$\#$
$\mathscr{B}$
$\Sigma\setminus\{\#\}$
$-,1$
q_0
$-,0$
Σ^n
$-,1$
$\Sigma\setminus\{\$\}$
Σ
$t_1,-$
$t_r,-$
q_1
q_r
$\{(t,i)\mid(t_1,t)\notin V, i\in\{0,1\}\}$
$\{(t,i)\mid(t_1,t)\notin H, i\in\{0,1\}\}$
$\{(t,i)\mid(t_r,t)\notin V, i\in\{0,1\}\}$
$\{(t,i)\mid(t_r,t)\notin H, i\in\{0,1\}\}$

Figure 1: Automata $\mathscr{A}$ and $\mathscr{B}$ used in the proof of Theorem 3.2.

The claim is that $\mathscr{A} \sqsubseteq_{\mathsf{la}} \mathscr{B}$ (resp. $\mathscr{A} \sqsubseteq^{\mathsf{f}}_{\mathsf{la}} \mathscr{B}$) if and only if there is no valid $n\times 2^n$ tiling. Assume first that a valid tiling exists. Then Spoiler wins if he plays the word that contains in the i-th block the i-th row of the tiling tagged with the binary representation of i. Note that Duplicator cannot loop forever in the initial state because she cannot read the # symbol. Conversely, assume there is no valid tiling. Then Duplicator wins if she waits until she has seen at most 2^n+1 blocks: either two blocks are not tagged with consecutive numbers, or Spoiler played exactly 2^n blocks but these do not code a valid tiling. In the former, Duplicator then accepts by moving to q_0 at the beginning of the first ill-tagged block, and in the later, she wins by moving to q_i after having read a tile t_i whose horizontal or vertical successor does not match. □

In order to establish an even higher lower bound for the continuous game we consider an EXPTIME-hard game-theoretic variant of the tiling problem on some tiling system $\mathscr{T}$. The game is played by two players: *Starter* and *Completer*. The task for Completer is to produce a valid tiling, whereas Starter's goal is to make it impossible. On every round $i\geq 1$,

1. Starter selects the tile $t_{1,i}$ starting the i-th row; if $i=1$, then $t_{1,i}=t_I$, otherwise $(t_{1,i-1},t_{1,i})\in V$.
2. Completer selects the tiles $t_{2,i},\dots t_{n,i}$ completing the i-th row; $(t_{1,i},t_{2,i}),\dots,(t_{n-1,i},t_{n,i})\in H$, and $(t_{2,i-1},t_{2,i}),\dots,(t_{n,i-1},t_{n,i})\in V$.

If one of the players gets stuck, the opponent wins. Otherwise Completer wins iff there are i,j such that $t_{i,j}=t_F$. The problem of deciding whether there exists a winning strategy for Starter in this tiling game is known to be EXPTIME-hard [7, 3]. Equally, deciding whether there is no winning strategy for

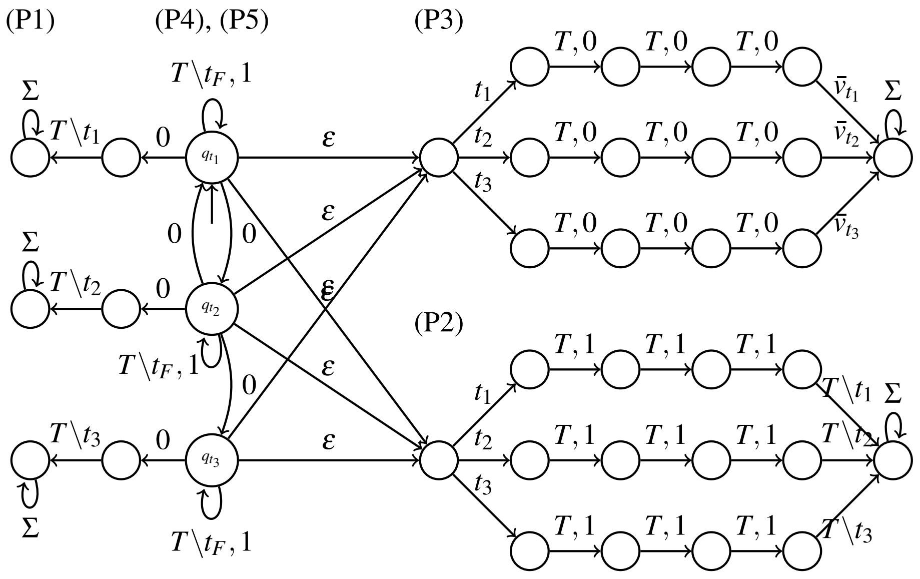


Figure 2: The NBA $\mathscr{B}$ from the construction in the proof of Thm. 3.3 for $m = 3$ and the tiling system $\mathscr{T} = (T,H,V,t_1,t_3)$ where $T = \{t_1,t_2,t_3\}, H = \{(t_1,t_1),(t_1,t_3),(t_3,t_3)\}$, and $V = \{(t_1,t_2),(t_2,t_1),(t_2,t_3)\}$. $\bar{v}_t$ denotes the set of tiles that are not vertically compatible with t.

him is EXPTIME-hard and – since the games are easily seen to be determined – so is the problem of deciding whether or not Completer has a winning strategy. This distinction is important because, as in the previous construction, we will reduce the complement of the tiling game problem to continuous buffered simulation. In other words, we present a reduction from one game to another in which the players' roles are inverted. Thus, Starter in the tiling game corresponds to Duplicator in the simulation game, and so do Completer and Spoiler.

Theorem 3.3. *Deciding $\sqsubseteq_{\mathsf{co}}$ (resp. $\sqsubseteq^{\mathsf{f}}_{\mathsf{co}}$) is EXPTIME-hard.*

Proof. Given a tiling system $\mathscr{T} = (T,H,V,t_0,t_F)$, we construct two NBA $\mathscr{A}$, $\mathscr{B}$ of polynomial size, that only contain accepting states, such that there is a winning strategy for Starter in the tiling game if and only if there is a winning strategy for Duplicator in the continuous simulation game ($\mathscr{A} \sqsubseteq_{\mathsf{co}} \mathscr{B}$, resp $\mathscr{A} \sqsubseteq^{\mathsf{f}}_{\mathsf{co}} \mathscr{B}$).

We consider the alphabet $T \uplus \{0,1\}$. Spoiler's automaton $\mathscr{A}$ is defined such that an infinite word w is accepted by $\mathscr{A}$ if and only if it is of the form $b_0w_0b_1w_1b_2w_2\ldots$, where for all $i \geq 0$, $b_i \in \{0,1\}$, $w_i \in T^n$, and two consecutive tiles in w_i are in the horizontal relation.

Duplicator's automaton does several things. It forces Spoiler to repeat the previous row when bit 1 occurs, i.e. if Spoiler plays w_i1w_{i+1}, then $w_i = w_{i+1}$. Duplicator also forces Spoiler to provide a vertically matching row when bit 0 occurs, i.e. if Spoiler plays w_i0w_{i+1}, then w_i and w_{i+1} must be vertically compatible consecutive rows. However, Duplicator does more: she always forces Spoiler to start the row with a given tile t; this tile is determined by the state q_t in which Duplicator currently is. Informally, the states q_t of Duplicator's automaton $\mathscr{B}$ are such that (1) q_{t_I} is the initial state of $\mathscr{B}$, and (2) if one starts reading from q_t, the following holds:

(P1) for an infinite word starting with $0t'\ldots$, with $t \neq t'$, one can pick an accepting run that does not depend on the infinite suffix;

(P2) for an infinite word starting with $bv1v' \dots$, $b \in \{0,1\}$, $v, v' \in T^n$, and $v \neq v'$, one can pick an accepting run that does not depend on the infinite suffix;

(P3) for an infinite word starting with $bv0v' \dots$, $b \in \{0,1\}$, $v, v' \in T^n$, if there is $i \in \{1, \dots, n\}$ such that the i-th letters of v and v' are not vertically compatible, then one can pick an accepting run that does not depend on the infinite suffix;

(P4) if $v \in T^n$ does not contain t_F, then $q_t \xrightarrow{1v} q_t$;

(P5) if $v \in T^n$ does not contain t_F, then $q_t \xrightarrow{0v} q_{t'}$ for all t' such that $(t, t') \in V$.

We illustrate the construction of $\mathscr{B}$ in Figure 2.

The main component is formed by the states q_{t_i} for $t_i \in T$. Each q_{t_i} is connected to another component that can detect a vertical mismatch (P3) and a non-proper repetition (P2). Each state q_{t_i} is also connected to a component that can detect when Spoiler does not respect Duplicator's choice of the first tile (P1). Each state q_{t_i} has a self-loop by reading $T \backslash t_F$ or 1 (P4) to consume the buffer and form an accepting run if one of Spoiler's mistakes is detected. Moreover, the automaton $\mathscr{B}$ encodes vertical compatibility for Duplicator's choice of the first tile by having edges $(q_{t_i}, 0, q_{t_{i'}}) \in \delta$ if and only if $(t_i, t_{i'}) \in V$ (P5).

We first show that if Completer has a winning strategy in the tiling game, then Spoiler has a winning strategy in the continuous fair simulation game on $\mathscr{A}$ and $\mathscr{B}$. Spoiler plays as follows: first, he moves along $0v_1$, where v_1 is the first row of the tiling. Then he iterates $1v_1$ for a while. This forces Duplicator to eventually remove $0v_1$ from the buffer, and commit to choosing some q_t, due to (P4) and (P5). Spoiler then considers the second row v_2 that Completer would answer if Starter would put t at the beginning of the second row. Spoiler picks this row v_2, and plays $0v_2$, followed by iterations of $1v_2$, and repeats the same principle.

Now we show that if Starter has a winning strategy then Duplicator has a winning strategy. Duplicator first waits for the $2n+2$ first letters of Spoiler. Because of (P1–P3), Spoiler has nothing better to do than to play $0v1v$ for some v encoding a valid first row of a tiling. Duplicator considers the tile t that would be played by Starter in the second row if Completer played v on the first row. Duplicator then removes $0v$ and ends in the state q_t. From there, she waits again for $n+1$ letters, so that the buffer now contains $1vbv'$ for some $b \in \{0,1\}$. Repeating the same process if $b = 1$, she can force Spoiler to eventually play $0v'$ where v' encodes a row vertically compatible with v and starting with t. Iterating this principle results in a play won by Duplicator, since either Completer never uses the final tile or Spoiler's move can always be mimicked by Duplicator due to (P4) and (P5) or, when Completer gets stuck on some row, Spoiler is forced to play a word with a vertical mismatch, and Duplicator wins by accepting the rest of the word. □

One may wonder why the EXPTIME-hardness proof for continuous simulation does not need the machinery of the binary counter as used in the PSPACE-hardness proof for look-ahead simulation. The reason is the following. In the look-ahead game Duplicator always has to flush the buffer entirely. Thus, she has to wait for the entire row-by-row tiling to be produced by Spoiler before she can point out a mistake. Thus, her best strategy is to wait for as long as possible but this would make her lose ultimately. The integrated counter forces Spoiler to get closer and closer to the moment when he has to play the final tile, and Duplicator can therefore relax and wait for that moment before she flushes the entire buffer. In the continuous game, Duplicator's ability to consume parts of the buffer is enough to force Spoiler to not delay the production of a proper tiling forever.

4 Upper Bounds: Quotient Games

We now show that the bounds of the previous section are tight by establishing the decidability of buffered simulations with corresponding complexity bounds. For this, we define a “quotient game” that has a finite state space, and show that it is equivalent to the buffered simulation game.

Continuous Quotient Game. The quotient game is based on the congruence relation associated with the Ramsey-based algorithm for complementation. We briefly recall its definition. Let us fix two Büchi automata $\mathscr{A} = (Q,\Sigma,\delta,q_I,F)$ and $\mathscr{B} = (Q,\Sigma,\delta,q'_I,F)$ – for simplicity we assume they share the same state space and only differ in their initial state. We introduce the function $f_w : Q^2 \to \{0,1,2\}$ defined as

$$f_w(q,q') = \begin{cases} 0 & \text{if } q \xrightarrow{w}_{🔔} q' \\ 1 & \text{if } q \overset{w}{\not\longrightarrow} q' \\ 2 & \text{otherwise} \end{cases}$$

We say that two finite words $w,w' \in \Sigma^*$ are equivalent, $w \sim w'$, if $f_w = f_{w'}$. Observe that $\sim$ is an equivalence relation, a congruence for word concatenation, and that the number $|\Sigma^*/{\sim}|$ of equivalence classes is bounded by $3^{|Q|^2}$. We write $[w]$ to denote the equivalence class of w with respect to $\sim$. We say a class $[w]$ is idempotent if $[ww] = [w]$.

Definition 4.1. The *continuous quotient game* is played between players Refuter and Prover[1] as follows. Initially, Refuter's pebble is on $q_0 := q_I$ and Prover's pebble is on $q'_0 := q'_I$. The players use an abstraction by equivalence classes of a buffer that, initially, contains $[\varepsilon]$. On each round $i \geq 1$:

1. Refuter chooses two equivalence classes $[w_1]$, $[w_2]$ and a state q_i, such that $q_{i-1} \xrightarrow{w_1} q_i \xrightarrow{w_2}_{🔔} q_i$ and $[w_2]$ is idempotent
2. Prover chooses q'_i such that $q'_{i-1} \xrightarrow{\beta w_1 w_2} q'_i \xrightarrow{w_2}_{🔔} q'_i$. The value β of the abstract buffer is set to $[w_2]$ for the next turn.

Prover wins the play if Refuter gets stuck or the play is infinitely long.

Proposition 4.2. *Whether Prover has a winning strategy for the continuous quotient game is decidable in EXPTIME.*

Proof. Observe first that the arena of the quotient game is finite and can be computed in exponential time. Indeed, a configuration of a quotient game is either a tuple $(q,q',[w])$ for Refuter's turn or a tuple $(q,q',[b],[w],[w'])$ for Prover's turn. The arena of the quotient game is thus finite and its size is bounded by $2|Q|^2 \cdot |\Sigma^*/{\sim}|^2 = 2|Q|^2 \cdot 3^{2|Q|^2} = 2^{\mathscr{O}(|Q|^2 \cdot \log |Q|)}$. The finite monoid $\Sigma^*/{\sim}$ can be computed in exponential time: starting from the set $\{[a] \mid a \in \Sigma\}$, compose any two classes until a fixpoint is reached. Composition of two equivalence classes given as functions of type $Q^2 \to \{0,1,2\}$ is not hard to compute [10].

Observe now that the quotient game is a reachability game from Refuter's point of view (he wins if he reaches a configuration in which Prover gets stuck), so once the arena is computed, one can decide the winner of the game in time polynomial in the size of the arena, which is exponential in $|Q|$. $\square$

We show that quotient games characterise the relation $\sqsubseteq^{\mathsf{f}}_{\mathsf{co}}$.

[1]We use different player names on purpose to make an easy distinction between the original simulation game and the quotient game.

Lemma 4.3. *$\mathscr{A} \sqsubseteq^{\mathsf{f}}_{\mathsf{co}} \mathscr{B}$ only if Prover has a winning strategy for the continuous quotient game.*

Proof. Assume that Refuter has a winning strategy for the continuous quotient game. We want to show that then Spoiler has a winning strategy for the continuous fair simulation game. We actually consider a variant of the continuous fair simulation game in which Spoiler may add more than one letter in a round, and Duplicator only removes one letter in a round. Clearly, Spoiler has a winning strategy for this variant if and only if he has a winning strategy for the continuous fair simulation game as defined in Section 2.2. Spoiler's strategy basically follows the one of Refuter. In the first round, Spoiler adds into the buffer some representatives w_1, w_2 of the equivalence classes played by Refuter. Spoiler then adds w_2 into the buffer on every round until the answer of Duplicator can be identified as a Prover's move in the quotient game, i.e. if Duplicator does not get stuck, she will eventually produce a trace of the form $q'_0 \xrightarrow{w_1 w_2^*} q'_1 \xrightarrow{w_2^+} q'_1$, since there are only finitely many states in the automaton. Then Spoiler considers the state q'_1 in which Duplicator is and looks at what Refuter would play if Prover would have picked q'_1. Iterating this principle, Spoiler mimics Refuter's winning strategy: eventually, since Prover gets stuck on some round i, Duplicator will get stuck when trying to mimick $w_1 w_2^+ \dots w_i^+$, and then Spoiler wins by continuously adding w_i into the buffer for the rest of the play. □

A key argument in the proof of the converse direction is the following lemma which is easily proved using Ramsey's Theorem [26].

Lemma 4.4. *Let $q_0, q_1, \dots$ be an infinite accepting run on $a_1 a_2 \dots$. Then there are i, j, k with $i < j < k$ such that $q_i = q_j = q_k$ is accepting and $a_{i+1} \dots a_j \sim a_{j+1} \dots a_k \sim a_{i+1} \dots a_k$.*

Lemma 4.5. *$\mathscr{A} \sqsubseteq^{\mathsf{f}}_{\mathsf{co}} \mathscr{B}$ if Prover has a winning strategy for the continuous quotient game.*

Proof. When the continuous simulation game starts, Duplicator just skips his turn for a while. Then Spoiler starts providing an infinite accepting run $q_0 a_0 q_1 a_1 \dots$ – if he does not, Duplicator waits forever and wins the play. At some point, Lemma 4.4 applies: the buffer contains $w_1 w_2 w'_2$ with $[w_2] = [w'_2]$ being idempotent, and Spoiler is in a state q that admits a $[w_2]$-loop. Then Duplicator considers the state q' in which Prover would move if Refuter played $[w_1], [w_2], q$ in the first round. She removes $w_1 w_2$ from the buffer and moves to this state q'. Duplicator proceeds identically in the next rounds, and either Spoiler eventually gets stuck or he follows a non-accepting run or the play is infinite. □

Lemmas 4.3 and 4.5 together with Prop. 4.2 yield an upper bound on the complexity of deciding continuous simulation. Together with the lower bound from Theorem 3.3 we get a complete characterisation of the complexity of continuous fair simulation.

Corollary 4.6. *Continuous fair simulation is EXPTIME-complete.*

Look-Ahead Quotient Game. In order to establish the decidability of look-ahead simulations, we introduce a look-ahead quotient game. The game essentially differs from the continuous quotient game in that it does not use a buffer.

Definition 4.7. The look-ahead quotient game is played between Refuter and Prover. Initially, Refuter's pebble is on $q_0 := q_I$, Prover's pebble is on $q'_0 := q'_I$, and the buffer β contains the equivalence class $[\varepsilon]$. On each round $i \geq 1$:

1. Refuter chooses two equivalence classes $[w_1]$, $[w_2]$ and a state q_i, such that $q_{i-1} \xrightarrow{w_1} q_i \xrightarrow{w_2} q_i$ and $[w_2]$ is idempotent.

2. Prover chooses q'_i such that there is a $q'_{i-1} \xrightarrow{w_1} q'_i \overset{w_2}{\dashrightarrow} q'_i$.

Prover wins the play if Refuter gets stuck or if the play is infinitely long.

Following the same kind of arguments we used for the continuous quotient game, the result below can be established.

Proposition 4.8. *$\mathscr{A} \sqsubseteq^{\mathsf{f}}_{\mathsf{la}} \mathscr{B}$ if and only if Prover has a winning strategy for the look-ahead quotient game.*

The size of the arena of a look-ahead quotient game is again exponential in the size of the automata; but there are only $|Q|^2$ positions for Refuter, so look-ahead quotient games can be solved slightly better than continuous ones.

Proposition 4.9. *Whether Prover has a winning strategy for the look-ahead quotient game can be decided in PSPACE.*

Proof. Consider the following non-deterministic algorithm that guesses the set W of all pairs (q_0, q'_0) of initial configurations of the game such that Duplicator has a winning strategy. For all (q_0, q'_0) in W, the following can then be checked in polynomial space: for all $[w_1], [w_2]$, and q_1 that could be played by Spoiler, there is q'_1 that can be played by Duplicator such that (q_1, q'_1) is in W. Inclusion in PSPACE then follows from Savitch's Theorem [28]. □

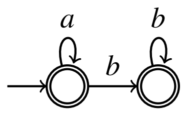
Corollary 4.10. *Look-ahead fair simulation is PSPACE-complete.*

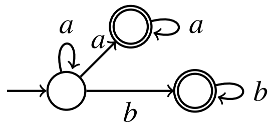
5 Properties of Buffered Simulations

In this section we investigate some fundamental properties of buffered simulations starting with a comparison to language inclusion. Remember that the main motivation for studying simulations is the approximation thereof.

Continuous Simulation vs. Language Inclusion. Continuous simulation is strictly smaller than language inclusion. It is not hard to see that continuous simulation implies language inclusion, so we focus on strictness.

The following example shows a case where language inclusion holds, indeed $L(\mathscr{A}) = L(\mathscr{B})$, but $\mathscr{A} \not\sqsubseteq^{\mathsf{f}}_{\mathsf{co}} \mathscr{B}$ since Spoiler can win the game by always producing a, whereas Duplicator has to keep the pebble on the initial state of $\mathscr{B}$ to be ready for a possible b.





Topological Characterisation. Consider a run of an NBA on some word $w = a_1a_2 \ldots \in \Sigma^\omega$ to be an infinite sequence $q_0, a_1, q_1, \ldots$ with the usual properties, i.e. the word is actually listed in the run itself. We write $\mathsf{Runs}(\mathscr{A})$ for the set of runs of $\mathscr{A}$ in this respect, and $\mathsf{ARuns}(\mathscr{A})$ for the set of accepting runs.

Given a set Δ, the set Δ^ω is equipped with a standard structure of a metric space. The distance $d(x,y)$ between two infinite sequences $x_0x_1x_2\ldots$ and $y_0y_1y_2\ldots$ is the real $\frac{1}{2^i}$, where i is the first index for which $x_i \neq y_i$. Intuitively, two words are "significantly close" if they share a "significantly long" prefix. The sets $\mathsf{Runs}(\mathscr{A})$ and $\mathsf{ARuns}(\mathscr{A})$ are subsets of $(Q \cup \Sigma)^\omega$; $\mathsf{Runs}(\mathscr{A})$ has the particularity of being a closed subset, and it is thus a compact space, whereas $\mathsf{ARuns}(\mathscr{A})$ is not.

We call a function $f : \mathsf{ARuns}(\mathscr{A}) \to \mathsf{ARuns}(\mathscr{B})$ *word preserving* if for all $\rho \in \mathsf{ARuns}(\mathscr{A})$, $f(\rho)$ and ρ are labelled with the same word. It can be seen that $L(\mathscr{A}) \subseteq L(\mathscr{B})$ holds if and only if there is a word preserving function $f : \mathsf{ARuns}(\mathscr{A}) \to \mathsf{ARuns}(\mathscr{B})$.

Proposition 5.1. *Let $\mathscr{A}, \mathscr{B}$ be two NBA. The following holds: $\mathscr{A} \sqsubseteq^{\mathsf{f}}_{\mathsf{co}} \mathscr{B}$ if and only if there is a continuous word preserving function $f : \mathsf{ARuns}(\mathscr{A}) \to \mathsf{ARuns}(\mathscr{B})$.*

Proposition 5.1 has some interesting consequences. First, it shows again that $\mathscr{A} \sqsubseteq^{\mathsf{f}}_{\mathsf{co}} \mathscr{B}$ implies $L(\mathscr{A}) \subseteq L(\mathscr{B})$, and explain the difference between the two in terms of continuity. Second, it shows that $\sqsubseteq_{\mathsf{co}}$ and $\sqsubseteq^{\mathsf{f}}_{\mathsf{co}}$ are transitive relations, since the composition of two continuous functions is continuous. Another application of Proposition 5.1 is that $\sqsubseteq_{\mathsf{co}}$ (but not $\sqsubseteq^{\mathsf{f}}_{\mathsf{co}}$) is decidable in 2-EXPTIME using a result of Holtmann et al. [20]. This is of course not optimal as seen in the previous section.

Remark 5.2. It might be asked whether look-ahead simulation has a topological characterisation similar to this one. The answer is negative: if it had (a reasonable) one, it would entail that look-ahead simulation is a transitive relation. However, Mayr and Clemente [9] gave examples of automata that show that look-ahead simulation is not transitive in general.

Buffered Simulations in Automata Minimisation. An important application of simulation relations in automata theory is automata minimisation. A preoder R over the set of states of an automaton $\mathscr{A}$ defines two new automata: its quotient $\mathscr{A}/R$, and its pruning $\mathsf{prune}(\mathscr{A}, R)$, c.f. Clemente's PhD thesis [8] for a formal definition of these notions. Intuitively, the quotient automaton is defined by merging states that are equivalent with respect to the preorder R, whereas pruning is obtained by removing a transition $q \xrightarrow{a} q_1$ if it is "subsumed" by a transition $q \xrightarrow{a} q_2$, where $q_1 \, R \, q_2$.

A preoder R is then said to be good for quotienting (GFQ) if $L(\mathscr{A}/R) = L(\mathscr{A})$, and good for pruning (GFP) if $L(\mathsf{prune}(\mathscr{A}, R)) = L(\mathscr{A})$. It can be checked that GFQ and GFP are antitone properties: if $R \supseteq R'$ and R is GFQ (resp. GFP), then so does R'.

Fair simulation is neither GFQ nor GFP; as a consequence, fair continuous and fair look-ahead simulations, which contain fair simulation, are not GFQ and GFP either. Simulation preorders that are used for automata minimisation rely on less permissive winning conditions than fairness. The *delayed* winning condition asserts that every round in which Spoiler visits an accepting state is (not necessarily immediately) succeeded by some round in which Duplicator also visits an accepting state. The *direct* winning condition imposes that, if Spoiler visits an accepting state in a given round, then in the same round Duplicator should visit an accepting state. Delayed simulation is known to be GFQ but not GFP, whereas direct simulation is known to be both GFP and GFQ. Since a play of a continuous/look-ahead simulation game yields a play of the standard simulation game, there is a natural buffered counterpart of delayed and direct simulation, obtained by changing the winning conditions accordingly.

Proposition 5.3. *Delayed continuous and delayed look-ahead simulation is GFQ but not GFP, and direct continuous as well as direct look-ahead simulation is GFP and GFQ.*

The proof is a rather straightforward consequence of similar results for multi-pebble simulations [15], and from the fact that these multi-pebble simulations subsume continuous simulations (provided the number of pebbles is larger than the number of states of duplicator's automaton).

Recall that bounded buffered simulation relations are polynomial time computable [22] and can be used to significantly improve language inclusion tests for NBA using automata minimisation [9]. We already showed that fair, unbounded, buffered simulation is not polynomial time computable, and thus cannot be used for improving language inclusion tests. We now extend this result to the delayed and direct buffered simulations.

Theorem 5.4. *The delayed (resp. direct) continuous simulation is EXPTIME hard, and the delayed (resp. direct) look-ahead simulation is PSPACE hard.*

This follows from a simple observation: the automata that were used in the hardness proofs had all states accepting, and in this case, fair, delayed and direct simulation coincide.

References

[1] P. A. Abdulla, Y.-F. Chen, L. Clemente, L. Holík, C.-D. Hong, R. Mayr & T. Vojnar (2010): *Simulation Subsumption in Ramsey-Based Büchi Automata Universality and Inclusion Testing*. In: *Proc. 22nd Int. Conf. on Computer-Aided Verification, CAV'10*, *LNCS* 6174, Springer, pp. 132–147, doi:10.1007/978-3-642-14295-6_14.

[2] P. Aziz Abdulla, Y.-F. Chen, L. Clemente, L. Holík, C.-D. Hong, R. Mayr & T. Vojnar (2011): *Advanced Ramsey-Based Büchi Automata Inclusion Testing*. In: *Proc. 22nd Int. Conf. on Concurrency Theory, CONCUR'11*, *LNCS* 6901, Springer, pp. 187–202, doi:10.1007/978-3-642-23217-6_13.

[3] Peter Van Emde Boas (1997): *The Convenience of Tilings*. In: *In Complexity, Logic, and Recursion Theory*, Marcel Dekker Inc, pp. 331–363, doi:10.1.1.38.763.

[4] J. R. Büchi (1962): *On a Decision Method in Restricted Second Order Arithmetic*. In: *Proc. Congress on Logic, Method, and Philosophy of Science*, Stanford University Press, Stanford, CA, USA, pp. 1–12, doi:10.1007/978-1-4613-8928-6_23.

[5] D. Bustan & O. Grumberg (2003): *Simulation-based minimization*. *ACM Trans. Comput. Logic* 4(2), pp. 181–206, doi:10.1145/635499.635502.

[6] G. Cécé & A. Finkel (2005): *Verification of programs with half-duplex communication*. *Inf. Comput.* 202(2), pp. 166–190, doi:10.1016/j.ic.2005.05.006.

[7] B. S. Chlebus (1986): *Domino-Tiling Games*. *Journal of Computer and System Sciences* 32, pp. 374–392, doi:10.1016/0022-0000(86)90036-X.

[8] L. Clemente (2012): *Generalized Simulation Relations with Applications in Automata Theory*. Ph.D. thesis, University of Edinburgh.

[9] Lorenzo Clemente & Richard Mayr (2013): *Advanced automata minimization*. In: *Proc. 40th Symp. on Principles of Programming Languages, POPL'13*, ACM, pp. 63–74, doi:10.1145/2429069.2429079.

[10] C. Dax, M. Hofmann & M. Lange (2006): *A proof system for the linear time μ-calculus*. In: *Proc. 26th Conf. on Foundations of Software Technology and Theoretical Computer Science, FSTTCS'06*, *LNCS* 4337, Springer, pp. 274–285, doi:10.1007/11944836_26.

[11] D. L. Dill, A. J. Hu & H. Wong-Toi (1991): *Checking for Language Inclusion Using Simulation Preorders*. In: *Proc. 3rd Int. Workshop on Computer-Aided Verification, CAV'91*, *LNCS* 575, Springer, pp. 255–265, doi:10.1007/3-540-55179-4_25.

[12] L. Doyen & J.-F. Raskin (2009): *Antichains for the Automata-Based Approach to Model-Checking*. *Logical Methods in Computer Science* 5(1), doi:10.2168/LMCS-5(1:5)2009.

[13] K. Etessami & G. J. Holzmann (2000): *Optimizing Büchi Automata*. In: *Proc. 11th Int. Conf. on Concurrency Theory, CONCUR'00*, *LNCS* 1877, Springer, pp. 153–167, doi:10.1007/3-540-44618-4_13.

[14] K. Etessami, T. Wilke & R. A. Schuller (2001): *Fair Simulation Relations, Parity Games, and State Space Reduction for Büchi Automata*. In: *Proc. 28th Int. Coll. on Algorithms, Languages and Programming, ICALP'01*, *LNCS* 2076, Springer, pp. 694–707, doi:10.1137/S0097539703420675.

[15] Kousha Etessami (2002): *A Hierarchy of Polynomial-Time Computable Simulations for Automata*. In Lubo Brim, Mojmr Ketnsk, Antonn Kuera & Petr Janar, editors: *CONCUR 2002 Concurrency Theory*, *Lecture Notes in Computer Science* 2421, Springer Berlin Heidelberg, pp. 131–144, doi:10.1007/3-540-45694-5_10.

[16] S. Fogarty & M. Y. Vardi (2010): *Efficient Büchi Universality Checking*. In: *Proc. 16th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'10*, *LNCS* 6015, Springer, pp. 205–220, doi:10.1007/978-3-642-12002-2_17.

[17] S. Fogarty & M. Y. Vardi (2012): *Büchi Complementation and Size-Change Termination*. *Logical Methods in Computer Science* 8(1), doi:10.2168/LMCS-8(1:13)2012.

[18] S. Gurumurthy, R. Bloem & F. Somenzi (2002): *Fair simulation minimization*. In: *Proc. 14th Int. Conf. on Computer-Aided Verification, CAV'02*, *LNCS* 2404, Springer, pp. 610–624, doi:10.1007/3-540-45657-0_51.

[19] T. A. Henzinger, O. Kupferman & S. K. Rajamani (2002): *Fair Simulation*. *Inf. Comput.* 173(1), pp. 64–81, doi:10.1006/inco.2001.3085.

[20] M. Holtmann, L. Kaiser & W. Thomas (2012): *Degrees of Lookahead in Regular Infinite Games*. *Logical Methods in Computer Science* 8(3), doi:10.2168/LMCS-8(3:24)2012.

[21] Gerard J. Holzmann (2004): *The SPIN Model Checker - primer and reference manual*. Addison-Wesley.

[22] M. Hutagalung, M. Lange & É. Lozes (2013): *Revealing vs. Concealing: More Simulation Games for Büchi Inclusion*. In: *Proc. 7th Int. Conf. on Language and Automata Theory and Applications, LATA'13*, LNCS, Springer, pp. 347–358, doi:10.1007/978-3-642-37064-9_31.

[23] O. Kupferman & M. Y. Vardi (2001): *Weak Alternating Automata Are Not That Weak*. *ACM Trans. on Comput. Logic* 2(3), pp. 408–429, doi:10.1145/377978.377993.

[24] C. S. Lee, N. D. Jones & A. M. Ben-Amram (2001): *The size-change principle for program termination*. In: *Proc. 28th Symp. on Principles of Programming Languages, POPL'01*, ACM, pp. 81–92, doi:10.1145/360204.360210.

[25] A. R. Meyer & L. J. Stockmeyer (1973): *Word problems requiring exponential time*. In: *Proc. 5th Symp. on Theory of Computing, STOC'73*, ACM, New York, pp. 1–9, doi:10.1145/800125.804029.

[26] F. P. Ramsey (1930): *On a problem in formal logic*. *Proc. London Math. Soc. (3)* 30, pp. 264–286, doi:10.1007/978-0-8176-4842-8_1.

[27] S. Safra (1988): *On the complexity of ω-automata*. In: *Proc. 29th Symp. on Foundations of Computer Science, FOCS'88*, IEEE, pp. 319–327, doi:10.1109/SFCS.1988.21948.

[28] W. J. Savitch (1970): *Relationships between nondeterministic and deterministic tape complexities*. *Journal of Computer and System Sciences* 4, pp. 177–192, doi:10.1016/S0022-0000(70)80006-X.

[29] W. Thomas (1999): *Complementation of Büchi automata revisited*. In J. Karhumäki et al., editor: *Jewels are Forever, Contributions on Theoretical Computer Science in Honor of Arto Salomaa*, Springer, pp. 109–122, doi:10.1007/978-3-642-60207-8_10.

[30] M. Y. Vardi (1996): *An Automata-Theoretic Approach to Linear Temporal Logic*, pp. 238–266. *LNCS* 1043, Springer, New York, NY, USA, doi:10.1007/3-540-60915-6_6.

Synchronizing weighted automata

Szabolcs Iván
University of Szeged, Hungary
szabivan@inf.u-szeged.hu

We introduce two generalizations of synchronizability to automata with transitions weighted in an arbitrary semiring $\mathbf{K} = (K, +, \cdot, 0, 1)$. (or equivalently, to finite sets of matrices in $\mathbf{K}^{n\times n}$.) Let us call a matrix A location-synchronizing if there exists a column in A consisting of nonzero entries such that all the other columns of A are filled by zeros. If additionally all the entries of this designated column are the same, we call A synchronizing. Note that these notions coincide for stochastic matrices and also in the Boolean semiring. A set $\mathscr{M}$ of matrices in $K^{n\times n}$ is called (location-)synchronizing if $\mathscr{M}$ generates a matrix subsemigroup containing a (location-)synchronizing matrix. The **K**-(location-)synchronizability problem is the following: given a finite set $\mathscr{M}$ of $n \times n$ matrices with entries in **K**, is it (location-)synchronizing? Both problems are PSPACE-hard for any nontrivial semiring. We give sufficient conditions for the semiring **K** when the problems are PSPACE-complete and show several undecidability results as well, e.g. synchronizability is undecidable if 1 has infinite order in $(K, +, 0)$ or when the free semigroup on two generators can be embedded into $(K, \cdot, 1)$.

1 Introduction

The synchronization (directing, reseting) problem of classical, deterministic automata is a well-studied topic with a vast literature (see e.g. [16] for a survey). An automaton $\mathscr{A}$ is *synchronizable* if some word u induces a constant function on its state set, in which case u is a synchronizing word of $\mathscr{A}$. Deciding whether an automaton is synchronizable can be done in polynomial time and it is also known that for synchronizable automata, a synchronizing word of length $\mathscr{O}(n^3)$ exists, where n denotes the number of its states. (The famous Černý conjecture from the sixties states that this bound is $(n-1)^2$.)

The notion of synchronizability has been extended e.g. (in three different ways) to nondeterministic automata in [9], to stochastic automata in [10] and more recently in another way in [2], to integer-weighted transitions in [1]. To our knowledge, only ad-hoc notions have been defined so far, each for a particular underlying semiring. We note that in [1] the notion has also been extended to timed automata as well.

In this paper we introduce several extensions of synchronizability to automata with transitions weighted in an arbitrary semiring $\mathbf{K} = (K, +, \cdot, 0, 1)$. For states p, q and word u, let $(pu)_q \in K$ denote the sum of the weights of all u-labeled paths from p to q, with the weight of a path being the product of the weights of its edges, as usual. Following the nomenclature of [1], we call the automaton $\mathscr{A}$ *location-synchronizable* if $\exists q, u$: $\forall p, r \quad (pu)_r \neq 0$ iff $r = q$ and *synchronizable* if $\exists q, u, k \neq 0$: $\forall p, r \quad (pu)_q = k$ and $(pu)_r = 0$ for each $r \neq q$.

As an equivalent formulation, let us call a matrix $A \in \mathbf{K}^{n\times n}$ *location synchronizing* if it contains a column entirely filled with nonzero values, and all its other entries are zero. If in addition all the nonzero values are the same, we call A *synchronizing*. Then, an instance of the synchronizability problems is a finite set $\mathscr{A} = \{A_i : 1 \le i \le k\}$ of matrices, each in $\mathbf{K}^{n\times n}$. The family $\mathscr{A}$ is called (location) synchronizable if it generates a (location) synchronizing matrix. The question is to decide whether the instance is (location) synchronizing.

Note that these notions coincide for stochastic automata and also in the Boolean semiring. For unconstrained automata, both problems are **PSPACE**-hard for any nontrivial semiring, and in any semiring, the length of the shortest directing word can be exponential. We give sufficient conditions for the semiring **K** when the problems are in **PSPACE** (and hence are **PSPACE**-complete) and show several undecidability results as well.

2 Notation

A *semiring* is an algebraic structure $\mathbf{K} = (K, +, \cdot, 0, 1)$ where $(K, +, 0)$ is a commutative monoid with identity 0, $(K, \cdot, 1)$ is a monoid with identity 1, 0 is an annihilator for $\cdot$ and $\cdot$ distributes over $+$, i.e. $0a = a0 = 0$, $(a+b)c = ac+bc$ and $a(b+c) = ab+ac$ for each $a,b,c \in K$. (When the context is clear, we usually omit the $\cdot$ sign.) The case when $|K| = 1$ is that of the trivial semiring; when $|K| > 1$, the semiring is nontrivial. Three semirings used in this paper are the *Boolean semiring* $\mathbf{B} = (\{0,1\}, \vee, \wedge, 0, 1)$ and the semirings **N** and **Z** of the natural numbers $\{0,1,2,\dots\}$ and the integers $\{0,\pm 1,\pm 2,\dots\}$ with the standard addition and product. Among these, only **Z** is a ring since the other two have no additive inverses. A semiring **K** is zero-sum-free if $a+b=0$ implies $a=b=0$; is zero-divisor-free if $ab=0$ implies $a=0$ or $b=0$; is positive if it is both zero-sum-free and zero-divisor-free; is locally finite if for any finite $K_0 \subseteq K$, the least subsemiring of K containing K_0 (which is also called the subsemiring of K generated by K_0) is finite.

An *alphabet* is a finite nonempty set, usually denoted A in this paper. When n is an integer, $[n]$ stands for the set $\{1,\dots,n\}$. For a set X, $P(X)$ denotes its power set $\{Y : Y \subseteq X\}$. For any alphabet A, the semiring of *languages* over A is $(P(A^*), \cup, \cdot, \emptyset, \{\varepsilon\})$ where product is concatenation of languages, $KL = \{uv : u \in K, v \in L\}$ and ε stands for the empty word.

When **K** is a semiring and $n > 0$ is an integer, then the set $\mathbf{K}^{n\times n}$ of $n\times n$ matrices with entries in **K** also forms a semiring with pointwise addition $(A+B)_{i,j} = A_{i,j} + B_{i,j}$ (for clarity, $A_{i,j}$ stands for the entry in the ith row and jth column) and the usual matrix product $(AB)_{i,j} = \sum_{k\in[n]} A_{i,k}B_{k,j}$. The zero element is the null matrix $\mathscr{O}_{i,j} = 0$ and the one element is the identity matrix $I_{i,j} = \begin{cases} 1, \text{ if } i=j \\ 0, \text{ otherwise} \end{cases}$ in $\mathbf{K}^{n\times n}$.

In this article we only take products of matrices, no sums and thus use the notion $\langle \mathscr{M} \rangle$ when $\mathscr{M} \subseteq \mathbf{K}^{n\times n}$ is a set of matrices for the least sub*monoid* of the monoid $(\mathbf{K}^{n\times n}, \cdot, I_n)$ containing $\mathscr{M}$. That is, $\langle \mathscr{M} \rangle$ contains all products of the form $M_1M_2\dots M_k$ with $k \ge 0$ and $M_i \in \mathscr{M}$ for each $i \in [k]$.

For a semiring **K**, alphabet A and integer $n > 0$, an *n-state* **K**-*weighted A-automaton* is a system $M = (\alpha, (M_a)_{a\in\Sigma}, \beta)$ where $\alpha, \beta \in \mathbf{K}^n$ are the *initial* and *final* vectors, respectively and for each $a \in A$, $M_a \in \mathbf{K}^{n\times n}$ is a *transition matrix*. The mapping $a \mapsto M_a$ extends in a unique way to a homomorphism $A^* \to \mathbf{K}^{n\times n}$, $w \mapsto M_w$ with $M_{a_1\dots a_k} = M_{a_1}\dots M_{a_k}$. The automaton M above associates to each word w a weight $M(w) = \alpha M_w \beta \in K$, where α is considered as a $1\times n$ row vector and β as an $n\times 1$ column vector. We usually do not specify the number n of states explicitly and omit **K** and A when the weight structure and/or the alphabet is clear from the context.

3 Synchronizability in various semirings

Classical nondeterministic automata (with multiple initial states but no ε-transitions) can be seen as automata with weights in the Boolean semiring. For any semiring **K**, a **K**-automaton $M = (\alpha, (M_a)_{a\in A}, \beta)$ is

- *partial* if there is at most one nonzero entry in each row of each transition matrix, and α has exactly one nonzero entry,
- *deterministic* if it is partial and there is exactly one nonzero entry in each row of each matrix M_a.

A classical deterministic automaton $M = (\alpha, (M_a)_{a\in A}, \beta)$ is called *synchronizable* (directable, resetable etc) if there exists a word w (called a synchronizing word of M) such that M_w has exactly one column that is filled with 1's and all the other entries of M_w are zero. (Traditionally, this property is formalized as w inducing a constant map on the state set.)

As an example, the 4-state automaton $M = (\alpha, (M_a)_{a\in\{0,1\}}, \beta)$ with arbitrary α and β and with transition matrices

$$M_0 = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix}, M_1 = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

is synchronizable since for the word 100010001, the transition matrix is

$$(M_1(M_0)^3)^2 M_1 = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}.$$

The notion celebrates its 50th anniversary this year – a very popular and intensively studied conjecture in the area is that of Černý stating if an n-state classical deterministic automaton is synchronizable, then it admits a synchronizing word of length at most $(n-1)^2$. We remark here that it is decidable in polynomial time (it's actually in **NL**) whether an input classical, deterministic automaton is synchronizable.

Synchronizability has been extended to nondeterminisic automata in [9] in three different ways. Here we highlight the one entitled "D3-directability" there: a **B**-automaton $M = (\alpha, (M_a)_{a\in A}, \beta)$ (that is, a classical nondeterministic automaton) is called D3-directable if there exists a word w such that M_w has exactly one column that is filled with 1's and all the other entries of M_w are zero. It is known (see e.g. [9]) that in general, the shortest synchronizing word of a synchronizable n-state **B**-automaton can have length $\Omega(2^n)$ with $O(2^n)$ being an upper bound [4]. For partial **B**-automata, the best known bounds are $\Omega(\sqrt[3]{3}^n)$ and $O(n^2\sqrt[3]{4}^n)$, see [11, 4].

In the next section of the paper we will frequently use the following results of [12]:

Theorem 1. *Deciding whether an input* **B**-*automaton is synchronizable is complete for* **PSPACE**. *The problem remains* **PSPACE**-*complete when restricted to partial* **B**-*automata.*

For the probabilistic semiring, in which case the weight structure is that of the nonnegative reals with the standard addition and product, and the input automata's transition matrices are restricted to be stochastic, the notion has been also generalized by several authors:

- In [10], M is synchronizable if there exists a word w such that all the rows of M_w are identical.
- In [2], M is synchronizable if there exists a single infinite word w such that for any $\varepsilon > 0$, there exists an integer K_ε such that for each finite prefix u of w having length at least K_ε, in M_u there is a column in which each entry is at least $1-\varepsilon$.

The problem of checking synchronizability is undecidable in the former setting and **PSPACE**-complete in the latter setting.

Most of these generalizations require (an arbitrary precise approximation of) a column consisting of ones and zeros everywhere else in some matrix of the form M_w. In fact, under these conditions it is a simple consequence of the structure of the semiring and the constraint on the automata that if in a row of a transition matrix M_w there is exactly one nonzero element, then it has to be 1. (The Boolean semiring has only two elements, while in the probability semiring the stochasticity of the matrices guarantee that the row sum is preserved and is one.)

The authors of [1] worked in the semiring $\mathbf{Z}$, with a different semantics notion, though: according to the notions of the present paper they worked in the semiring $P_f(\mathbf{Z})$, where the elements are finite sets of integers, with union as addition and complex sum $X+Y = \{x+y : x \in X, y \in Y\}$ being product. There two different notions of synchronizability are introduced: a matrix M is *location synchronizing* if there exists a column in which each entry is nonzero, while all the other entries of the matrix are zeroes (recall that in this semiring $\emptyset$ plays as zero) and is *synchronizing* if additionally the nonzero entries all coincide and map every possible starting vector α to some fixed vector (which is simply not possible in this semiring since this would require the presence of an $\mathscr{L}$-trivial element of the semiring). An automaton M is location synchronizable if there exists a word w such that M_w is location synchronizing. Regarding the complexity issues, location synchronizability is **PSPACE**-complete (which is due to the fact that $P_f(\mathbf{Z})$ is positive, cf. Proposition 4) and synchronizability is trivially false.

In this paper we extend the notion of synchronizability in spirit similar to [1], covering most of the generalizations above (the exception being the case of the probabilistic semiring, which seems to require a notion of metric).

Definition 1. Given a semiring $\mathbf{K}$ and a matrix $M \in \mathbf{K}^{n\times n}$, we say that M is

- *location synchronizing* if there exists a (unique) integer $i \in [n]$ such that $M_{j,k} \neq 0$ iff $k = i$;
- *synchronizing* if it is location synchronizing and additionally, $M_{j,i} = M_{1,i}$ for each $j \in [n]$ for the above index i.

A finite set $M_1, \dots, M_k \in \mathbf{K}^{n\times n}$ of matrices is *(location) synchronizable* if they generate a (location) synchronizing matrix, i.e. when $M_{i_1}M_{i_2}\dots M_{i_t}$ is (location) synchronizing for some $i_1, \dots, i_t \in [k]$, $t > 0$.

A $\mathbf{K}$-automaton is (location) synchronizable if so is its set of transition matrices.

We formulate the $\mathbf{K}$-(location) synchronizing problem (**K-Sync** and **K-LocSync** for short) as follows: given a finite set $\mathscr{M} = \{M_1, \dots, M_k\}$ of matrices in $\mathbf{K}^{n\times n}$ for some $n > 0$, decide whether $\mathscr{M}$ is (location) synchronizable?

(Clearly, this is equivalent to having a single $\mathbf{K}$-automaton as input.)

4 Results on complexity of the two problems

Given a semiring $\mathbf{K}$, call a matrix $M \in \mathbf{K}^{n\times n}$ a partial $0/1$-matrix if in each row there is at most one nonzero entry, which can have only a value of 1 if present, formally for each i there exists at most one j with $M_{i,j} \neq 0$ in which case $M_{i,j} = 1$ has to hold. Observe that the product of two partial $0/1$-matrices is still a partial $0/1$-matrix, being the same in any semiring. Moreover, a partial $0/1$-matrix is synchronizing iff it is location synchronizing. Thus the following are equivalent for any set $\mathscr{M} \subseteq \mathbf{K}^{n\times n}$ of partial $0/1$-matrices:

1. $\mathscr{M}$ is synchronizable;
2. $\mathscr{M}$ is location synchronizable;
3. $\mathscr{M}$, viewed as a set of partial $0/1$-matrices over $\mathbf{B}$, is synchronizable.

Since by Theorem 1 the last condition is **PSPACE**-hard to check, we immediately get the following:

Proposition 1. *For any nontrivial semiring* $\mathbf{K}$, *both* **K-Sync** *and* **K-LocSync** *are* **PSPACE**-*hard.*

4.1 Decidable subcases

First we make several (rather straightforward) observations on decidable subcases, generally involving finiteness conditions.

Of course if $\mathbf{K}$ is finite, we get **PSPACE**-completeness:

Proposition 2. *For any finite semiring* $\mathbf{K}$ *both problems are in* **PSPACE**, *thus are* **PSPACE**-*complete.*

Proof. Given an instance $\mathscr{M} = \{M_1, \ldots, M_k\}$ of the problem, we store a current matrix $C \in \mathbf{K}^{n\times n}$ initialized by the unit matrix I_n of $\mathbf{K}^{n\times n}$. In an endless loop, we nondeterministically choose an index $i \in [k]$ and let $C := CA_i$. After each step we check whether C is (location) synchronizing. If so, we report acceptance, otherwise continue the iteration.

If $\mathbf{K}$ is finite, storing an entry of C takes constant space, so storing C takes $O(n^2)$ memory, as well as computation of the product matrix. In total, we have an **NPSPACE** algorithm which is **PSPACE** by Savitch's theorem [13]. $\square$

Proposition 3. *For any locally finite semiring* $\mathbf{K}$, *both* **K-Sync** *and* **K-LocSync** *are decidable, provided that addition and product of* $\mathbf{K}$ *are computable.*

Proof. Recall that a semiring $\mathbf{K}$ is locally finite if any finite subset of K generates a finite subsemiring of $\mathbf{K}$.

Now given an instance $\mathscr{M} = \{M_1, \ldots, M_k\}$ of the problem, let $X = \{M_{i\,j,t} : i \in [k], j,t \in [n]\} \subseteq K$ stand for the finite set of the entries occurring in any of the matrices. Then clearly, $\langle \mathscr{M} \rangle \subseteq \mathbf{X}^{n\times n}$ where $\mathbf{X}$ is the subsemiring of $\mathbf{K}$ generated by X. Since $\mathbf{K}$ is finitely generated, this implies $\langle \mathscr{M} \rangle$ is finite as well, hence there exists an integer t such that $\langle \mathscr{M} \rangle = \mathscr{M}^{\leq t} = \{M_{i_1}M_{i_2}\ldots M_{i_d} : d \leq t, i_1, \ldots, i_d \in [k]\}$ which can be chosen to be the least integer t with $\mathscr{M}^{\leq t} = \mathscr{M}^{\leq t+1}$. Hence by computing the sets $\mathscr{M}^{\leq t}$ for $t = 0, 1, 2, \ldots$

and reporting acceptance when a witness is found and rejecting the input when $\mathscr{M}^{\le t} = \mathscr{M}^{\le t+1}$ gets satisfied without finding a witness we decide the respective problem.

(Note that computability of addition and product is needed for the effective computation of the sets above.) $\square$

Proposition 4. *For any positive semiring* **K**, **K-LocSync** *is in* **PSPACE**.

Proof. For any positive semiring **K** the mapping $\sigma : \mathbf{K} \to \mathbf{B}$ which maps 0 to 0 and all other elements of K to 1, is a semiring morphism. Hence σ can be extended pointwise to a semiring morphism $\sigma : \mathbf{K}^{n\times n} \to \mathbf{B}^{n\times n}$, with $(\sigma(A))_{i,j} = \sigma(A_{i,j})$. Then, a matrix $A \in \mathbf{K}^{n\times n}$ is *location* synchronizing if and only if $\sigma(A)$ is (location) synchronizing. Hence **K-LocSync** can be reduced to **B-Sync** via the polytime reduction $\{A_1,\dots,A_k\} \mapsto \{\sigma(A_1),\dots,\sigma(A_k)\}$, which is solvable in **PSPACE**, hence so is **K-LocSync**. $\square$

Remark 1. One can use the above semiring morphism to decide any such property of matrices which cares only on the positions of zeroes (i.e. when M satisfies the property if and only if so does $\sigma(M)$). Examples of such properties are *mortality* (whether the all-zero matrix is generated), and the *zero-in-the-upper-left-corner* (whether a matrix with a zero in the upper-left corner is generated). Thus both properties are in **PSPACE** for positive semirings (and are in fact undecidable for the semiring **Z**, which is *not* zero-sum-free).

Synchronizability, on the other hand, as well as the "equal entries problem" asking whether a matrix is generated having the same entry at two specified positions, is not such a property. The latter is well-known to be undecidable in **N** while the former is shown to be undecidable in Theorem 2.

4.2 Undecidable subcases

Now we turn our attention to undecidability results.

A well-known undecidable problem is the *Fixed Post Correspondence Problem*, or FPCP for short: given a finite set $\{(u_1,v_1),\dots,(u_k,v_k)\}$ of pairs of nonempty words over a binary alphabet, does there exist a nonempty index sequence $i_1,\dots,i_t$, each i_j in $[k]$, $t > 0$ with $i_t = 1$ (i.e. we fix the *last* used tile) such that $u_{i_1}u_{i_2}\dots u_{i_t} = v_{i_1}v_{i_2}\dots v_{i_t}$? The problem is already undecidable for the fixed constant $k = 7$ (also, it's known to be decidable for $k = 2$, see [8] and has an unknown decidability status for $3 \le k \le 6$).

Proposition 5. *For any semiring* **K** *such that the semigroup* $(\{a,b\}^*,\cdot)$ *embeds into the multiplicative monoid* $(K,\cdot,1)$ *of* **K**, *the* **K-Sync** *problem is undecidable, even for two-state deterministic WFA with an alphabet size of* 8 *(i.e. for eight* 2×2 *matrices when the question is viewed as a problem for matrices).*

Proof. In order to ease notation, suppose $(\{a,b\}^*,\cdot)$ is a subsemigroup of $(K,\cdot,1)$. For words $u,v \in \{a,b\}^+$, let us define the matrices $A(u,v) = \begin{pmatrix} u & 0 \\ 0 & v \end{pmatrix}$ and $B(u,v) = \begin{pmatrix} u & 0 \\ v & 0 \end{pmatrix}$. Then a direct computation shows that

$$A(u_1,v_1)A(u_2,v_2) = A(u_1u_2,v_1v_2),$$
$$B(u_1,v_1)A(u_2,v_2) = B(u_1u_2,v_1v_2),$$
$$B(u_1,v_1)A(u_2,v_2) = B(u_1,v_1)B(u_2,v_2) = B(u_1u_2,v_1u_2).$$

Also, matrices $A(u,v)$ are not synchronizing while matrices $B(u,v)$ are synchronizing iff $u=v$. Moreover, a product $B(u_1,v_1)X$ is synchronizing for $X \in \langle \cup_{u,v\in\{a,b\}^+}\{A(u,v),B(u,v)\}\rangle$ iff $u_1 = v_1$. Thus we can derive that a product of the form $X_1(u_1,v_2)X_2(u_2,v_2)\dots X_k(u_k,v_k)$ with each X_i being either A or B and $u_i,v_i \in \{0,1\}^+$ is synchronizing iff there exists some $t \in [k]$ such that $X_t = B$, $X_{t'} = A$ for each $t' < t$ and $u_1\dots u_t = v_1\dots v_t$ holds.

Hence, a reduction from FPCP to **K-Sync** is given by the transformation

$$\{(u_i,v_i) : i \in [k]\} \quad \mapsto \quad \{A(u_i,v_i) : i \in [k]\} \cup \{B(u_1,v_1)\}.$$

Since FPCP is undecidable, so is **K-Sync**. □

Note that $(\Sigma^*,\cup,\cdot,\emptyset,\{\varepsilon\})$ is positive, so its location synchronization problem is decidable in polynomial space, while when $|\Sigma| > 1$, its synchronization problem becomes undecidable.

Now we give a polynomial-time reduction from the **K**-mortality problem to both of the **K**-synchronization and the **K**-location synchronization problem. The **K**-mortality problem is actively studied for the case $\mathbf{K} = \mathbf{Z}$:

Definition 2. For a fixed semiring **K**, the **K**-mortality problem is the following: given a finite set $\mathscr{M} = \{M_1,\dots,M_k\}$ of matrices in $\mathbf{K}^{n\times n}$ for some $n > 0$, does $\langle\mathscr{M}\rangle$ contain the null matrix $\mathscr{O}_n$?

Proposition 6. *For any semiring K, the* **K***-mortality problem reduces to both of* **K-Sync** *and* **K-LocSync***. Thus, in particular, when* **K***-mortality problem is undecidable, so are both synchronizability problems.*

Proof. Let $\mathscr{M} = \{M_1,\dots,M_k\}$ be an instance of the **K**-mortality problem. We define the matrices $A_i = \begin{pmatrix} 1 & \mathbf{0} \\ \mathbf{0} & M_i \end{pmatrix}$, i.e. adding an all-zero top row and an all-zero first row to each M_i, $i \in [k]$ and fill the upper-left corner by 1. Also, we define $A_0 = \begin{pmatrix} 1 & \mathbf{0} \\ \mathbf{1} & I_n \end{pmatrix}$. We claim that the following are equivalent:

1. $\mathscr{O}_n \in \langle\mathscr{M}\rangle$;
2. $\mathscr{A} = \{A_i : 0 \le i \le k\}$ is synchronizable;
3. $\mathscr{A}$ is location synchronizable.

Observe that each member of $\mathscr{A}$ is block-lower triangular with 1 in the upper left corner, hence for any product $A = A_{i_1}A_{i_2}\dots A_{i_t}$ we have $A = \begin{pmatrix} 1 & 0 \\ X & M_{i_1}M_{i_2}\dots M_{i_t} \end{pmatrix}$ for some column vector X. Note that in order to ease notation we define M_0 as the unit matrix I_n and set $\mathscr{M} = \{M_0,\dots,M_k\}$ – since I_n is not synchronizing and is the unit element of $\mathbf{K}^{n\times n}$, this neither affects mortality (of $\mathscr{M}$) nor synchronizability (of $\mathscr{A}$).

Thus in particular the first column of any matrix $A \in \langle\mathscr{M}\rangle$ contains a nonzero entry, hence A is (location) synchronizing only if $M_{i_1}M_{i_2}\dots M_{i_t} = \mathscr{O}_n$, in which case $\mathscr{M}$ is indeed a positive instance of the **K**-mortality problem, showing iii)→ i). For i)→ii), let $A_{i_1}\dots A_{i_t} = \mathscr{O}_n$, $t > 0$, $i_j \in [k]$. Then $M := M_{i_1}\dots M_{i_t} = \begin{pmatrix} 1 & \mathbf{0} \\ \mathbf{0} & \mathscr{O}_n \end{pmatrix}$, thus $A_0M = \begin{pmatrix} 1 & \mathbf{0} \\ \mathbf{1} & \mathscr{O}_n \end{pmatrix}$ is a synchronizing matrix. Finally, ii)→iii) is clear for any $\mathscr{A}$. □

In particular, since mortality is undecidable in **Z**, so are **Z-Sync**and **Z-LocSync**.

Our most involved result on undecidability is the following one:

Theorem 2. **N-Sync** *is undecidable. Thus if* **N** *embeds into* **K** *(i.e. when* 1 *has infinite order in* $(K,+,0)$*), then so is* **K-Sync***.*

Proof. We give a polynomial-time reduction from the FPCP problem to **N-Sync**. This time we use the variant of FPCP in which the *first* tile is fixed to (u_1,v_1). Let $\{(u_i,v_i) : i \in [k]\}$ be an instance of the FPCP, $u_i,v_i \in \{0,1\}^+$. For a nonempty word $u \in \{0,1\}^+$ let $\mathrm{int}(u)$ be its value when considered as a *ternary* number, i.e. $\mathrm{int}(a_{n-1}\dots a_0) = \sum_{0\le i<n} a_i 3^i$. Also, we define for each word u a matrix $M(u) = \begin{pmatrix} 3^{|u|} & 0 \\ \mathrm{int}(u) & 1 \end{pmatrix}$. Then, since $\mathrm{int}(uv) = 3^{|v|}\mathrm{int}(u)+\mathrm{int}(v)$, we get that $M(u)M(v) = M(uv)$ and since the mapping $u \mapsto M(u)$ is also injective, it is an embedding of the semigroup $(\{0,1\}^+,\cdot)$ into $\mathbf{N}^{2\times 2}$.

We define the following matrices A_i, $i \in [k]$, B and C, all in $\mathbf{N}^{6\times 6}$:

$$A_i = \begin{pmatrix} M(u_i) & 0 & 0 \\ 0 & M(u_i) & 0 \\ 0 & 0 & M(v_i) \end{pmatrix},$$

$$B = \begin{pmatrix} \mathrm{int}(u_1) & 1 & \mathrm{int}(u_1) & 1 & 0 & 0 \\ \mathrm{int}(u_1) & 1 & \mathrm{int}(u_1) & 1 & 0 & 0 \\ 0 & 0 & \mathrm{int}(u_1) & 1 & 0 & 0 \\ 0 & 0 & \mathrm{int}(u_1) & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & \mathrm{int}(v_1) & 1 \\ 0 & 0 & 0 & 0 & \mathrm{int}(v_1) & 1 \end{pmatrix},$$

$$C = \begin{pmatrix} 0&0&0&0&0&0 \\ 0&0&0&0&0&0 \\ 1&0&0&0&0&0 \\ 0&0&0&0&0&0 \\ 1&0&0&0&0&0 \\ 0&0&0&0&0&0 \end{pmatrix},$$

that is, C has exactly two nonzero entries, namely $C_{3,1} = C_{5,1} = 1$.

Then for any sequence $i_2,\dots,i_t$, $t \ge 1$ we have

$$A_{i_2}\dots A_{i_t} = \begin{pmatrix} M(u) & 0 & 0 \\ 0 & M(u) & 0 \\ 0 & 0 & M(v) \end{pmatrix}$$

with $u = u_{i_2}\dots u_{i_t}$ and $v = v_{i_2}\dots v_{i_t}$ and also

$$BA_{i_2}\dots A_{i_t} = \begin{pmatrix} \mathrm{int}(u_1u) & \mathrm{int}(u) & \mathrm{int}(u_1u) & \mathrm{int}(u) & 0 & 0 \\ \mathrm{int}(u_1u) & \mathrm{int}(u) & \mathrm{int}(u_1u) & \mathrm{int}(u) & 0 & 0 \\ 0 & 0 & \mathrm{int}(u_1u) & \mathrm{int}(u) & 0 & 0 \\ 0 & 0 & \mathrm{int}(u_1u) & \mathrm{int}(u) & 0 & 0 \\ 0 & 0 & 0 & 0 & \mathrm{int}(v_1v) & \mathrm{int}(v) \\ 0 & 0 & 0 & 0 & \mathrm{int}(v_1v) & \mathrm{int}(v) \end{pmatrix},$$

and thus

$$BA_{i_2}\dots A_{i_t}C = \begin{pmatrix} \mathrm{int}(u_1u) & 0 & 0 & 0 & 0 & 0 \\ \mathrm{int}(u_1u) & 0 & 0 & 0 & 0 & 0 \\ \mathrm{int}(u_1u) & 0 & 0 & 0 & 0 & 0 \\ \mathrm{int}(u_1u) & 0 & 0 & 0 & 0 & 0 \\ \mathrm{int}(v_1v) & 0 & 0 & 0 & 0 & 0 \\ \mathrm{int}(v_1v) & 0 & 0 & 0 & 0 & 0 \end{pmatrix},$$

which is synchronizing if and only if $u_1u_{i_2}\dots u_{i_t} = v_1v_{i_2}\dots v_{i_t}$, hence if $\{(u_i,v_i) : i \in [k]\}$ is a positive instance of FPCP, then $\mathscr{M} = \{A_i : i \in [k]\} \cup \{B,C\}$ is synchronizable.

For the other direction, suppose $\mathscr{M}$ is synchronizable. We already argued that any member A of $\langle\{A_i : i \in [k]\}\rangle$ has the form $\begin{pmatrix} M(u) & 0 & 0 \\ 0 & M(u) & 0 \\ 0 & 0 & M(v) \end{pmatrix}$ for words u,v with $u = u_{i_1}u_{i_2}\dots u_{i_t}$ and $v = v_{i_1}v_{i_2}\dots v_{i_t}$ for some $i_j \in [k]$, $t \geq 0$. These matrices are clearly not (location) synchronizing.

Considering the matrix C, we have the following claims:

Claim A. For any matrix X we have $XC = \begin{pmatrix} c_1 & \\ c_2 & \\ \vdots & \mathbf{0} \\ c_6 & \end{pmatrix}$ for some $c_1,\dots,c_6 \in \mathbf{N}$.

Claim B. If XCY is synchronizing for some matrices X and Y, then so is XC.

Indeed, XC is the matrix whose first column is the sum of the third and the fifth column of X, and whose other entries are all zero. Also, if $XC = \begin{pmatrix} c_1 & \\ c_2 & \\ \vdots & \mathbf{0} \\ c_6 & \end{pmatrix}$ then $XCY = \begin{pmatrix} c_1r_1 \\ c_2r_1 \\ \vdots \\ c_6r_1 \end{pmatrix}$ where r_1 is the first row of Y. If XCY is synchronizing, this implies $c_ir_1 = c_jr_1 \neq 0$ for each $i,j \in [6]$, hence $c_i = c_j$ and XC is synchronizing as well.

Thus, by ii) above we get that if $\mathscr{M}$ is synchronizable, then there is a synchronizing matrix of the form XC with $X \in \langle\{A_i : i \in [k]\} \cup \{B\}\rangle$.

Inspecting members of $\langle\{A_i : i \in [k]\} \cup \{B\}\rangle$ we get the following claim:

Claim C. Let $\mathscr{A}$ stand for the matrix semigroup $\langle\{A_i : i \in [k]\}\rangle$. Then for any $n \geq 0$, any member of $\mathscr{A}(B\mathscr{A})^n$ has the form $\begin{pmatrix} X & nX & 0 \\ 0 & X & 0 \\ 0 & 0 & Y \end{pmatrix}$ for some matrices $X,Y \in \mathbf{N}^{2\times 2}$.

Indeed, for the base case $n = 0$ we have matrices of the form $\begin{pmatrix} M(u) & 0 & 0 \\ 0 & M(u) & 0 \\ 0 & 0 & M(v) \end{pmatrix}$ satisfying the condition. Suppose the claim holds for n and consider a matrix $M \in \mathscr{A}(B\mathscr{A})^{n+1} = \mathscr{A}(B\mathscr{A})^nB\mathscr{A}$. By the induction hypothesis, $M = M_0BA$ with $M_0 = \begin{pmatrix} X & nX & 0 \\ 0 & X & 0 \\ 0 & 0 & Y \end{pmatrix}$, and $A = \begin{pmatrix} M(u) & 0 & 0 \\ 0 & M(u) & 0 \\ 0 & 0 & M(v) \end{pmatrix}$ for some $X, Y \in \mathbf{N}^{2\times 2}$ and words u, v. We can also write U_1 for $\begin{pmatrix} \mathrm{int}(u_1) & 1 \\ \mathrm{int}(u_1) & 1 \end{pmatrix}$ and V_1 for $\begin{pmatrix} \mathrm{int}(v_1) & 1 \\ \mathrm{int}(v_1) & 1 \end{pmatrix}$. Calculating the product we get

$$\begin{aligned} M = M_0BA &= \begin{pmatrix} X & nX & 0 \\ 0 & X & 0 \\ 0 & 0 & Y \end{pmatrix} \begin{pmatrix} U_1 & U_1 & 0 \\ 0 & U_1 & 0 \\ 0 & 0 & V_1 \end{pmatrix} \begin{pmatrix} M(u) & 0 & 0 \\ 0 & M(u) & 0 \\ 0 & 0 & M(v) \end{pmatrix} \\ &= \begin{pmatrix} XU_1M(u) & (n+1)XU_1M(u) & 0 \\ 0 & XU_1M(u) & 0 \\ 0 & 0 & YV_1M(v) \end{pmatrix}, \end{aligned}$$

showing the claim.

Thus, since $\langle \{A_i : i \in [k]\} \cup \{B\} \rangle = \bigcup_{n\geq 0} \mathscr{A}(B\mathscr{A})^n$, we get by Claim B that if $\mathscr{M}$ is synchronizable, then there is a synchronizing matrix of the form $\begin{pmatrix} X & nX & 0 \\ 0 & X & 0 \\ 0 & 0 & Y \end{pmatrix} C$. Writing $X = \begin{pmatrix} x_1 & x_2 \\ x_3 & x_4 \end{pmatrix}$ and $Y = \begin{pmatrix} y_1 & y_2 \\ y_3 & y_4 \end{pmatrix}$ we get that this product is further equal to $\begin{pmatrix} nx_1 & \\ nx_3 & \\ x_1 & \mathbf{0} \\ x_3 & \\ y_1 & \\ y_3 & \end{pmatrix}$ which is synchronizing if and only if $n = 1$ and $x_1 = x_3 = y_1 = y_3 \neq 0$. By $n = 1$ we get that if $\mathscr{M}$ is synchronizable, then there is a synchronizing matrix of the form

$$X = A_{j_1}A_{j_2}\dots A_{j_\ell}BA_{i_2}A_{i_3}\dots A_{i_t}C,$$

with $\ell \geq 0$, $t \geq 1$, $j_r, i_r \in [k]$. Writing $u = u_1u_{i_2}\dots u_{i_t}$, $v = v_1v_{i_2}\dots v_{i_t}$, $u' = u_{j_1}\dots u_{j_\ell}$ and $v' = v_{j_1}\dots v_{j_\ell}$ we can write

$$\begin{aligned}
X &= A_{j_1}A_{j_2}\dots A_{j_\ell}BA_{i_2}A_{i_3}\dots A_{i_t}C \\
&= \begin{pmatrix} M(u') & 0 & 0 \\ 0 & M(u') & 0 \\ 0 & 0 & M(v') \end{pmatrix}
\begin{pmatrix}
\mathrm{int}(u_1u) & 1 & \mathrm{int}(u_1u) & 1 & 0 & 0 \\
\mathrm{int}(u_1u) & 1 & \mathrm{int}(u_1u) & 1 & 0 & 0 \\
0 & 0 & \mathrm{int}(u_1u) & 1 & 0 & 0 \\
0 & 0 & \mathrm{int}(u_1u) & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & \mathrm{int}(v_1v) & 1 \\
0 & 0 & 0 & 0 & \mathrm{int}(v_1v) & 1
\end{pmatrix} C \\
&= \begin{pmatrix}
3^{|u'|}\mathrm{int}(u_1u) & 3^{|u'|} & 3^{|u'|}\mathrm{int}(u_1u) & 3^{|u'|} & 0 & 0 \\
(\mathrm{int}(u')+1)\cdot\mathrm{int}(u_1u) & \mathrm{int}(u')+1 & (\mathrm{int}(u')+1)\cdot\mathrm{int}(u_1u) & \mathrm{int}(u')+1 & 0 & 0 \\
0 & 0 & 3^{|u'|}\mathrm{int}(u_1u) & 3^{|u'|} & 0 & 0 \\
0 & 0 & (\mathrm{int}(u')+1)\cdot\mathrm{int}(u_1u) & \mathrm{int}(u')+1 & 0 & 0 \\
0 & 0 & 0 & 0 & 3^{|v'|}\mathrm{int}(v_1v) & 3^{|v'|} \\
0 & 0 & 0 & 0 & (|v'|+1)\cdot\mathrm{int}(v_1v) & \mathrm{int}(v')
\end{pmatrix} C \\
&= \begin{pmatrix}
3^{|u'|}\mathrm{int}(u_1u) & \\
(\mathrm{int}(u')+1)\cdot\mathrm{int}(u_1u) & \\
3^{|u'|}\mathrm{int}(u_1u) & \mathbf{0} \\
(\mathrm{int}(u')+1)\cdot\mathrm{int}(u_1u) & \\
3^{|v'|}\mathrm{int}(v_1v) & \\
(\mathrm{int}(v')+1)\cdot\mathrm{int}(v_1v) &
\end{pmatrix}
\end{aligned}$$

which is synchronizing only if $3^{|u'|} = \mathrm{int}(u')+1$ and $3^{|v'|} = \mathrm{int}(v')+1$, that is, $u' = v' = \varepsilon$ implying $\ell = 0$.

Hence if $\mathscr{M}$ is synchronizable then there exists a synchronizing product of the form $BA_{i_2}A_{i_3}\dots A_{i_t}C$, which in turn implies $u_1u_{i_2}\dots u_{i_t} = v_1v_{i_2}\dots v_{i_t}$, thus in that case $\{(u_i,v_i) : i \in [k]\}$ is indeed a positive instance of the FPCP problem. $\square$

We note that the idea of encoding of a PCP variant within matrix semirings is not new, see e.g. [7, 15, 3]. For example, $\mathbf{Z}$-mortality can be shown to be undecidable for 3×3 integral matrices via a similar embedding $(u,v) \mapsto M(u,v) = \begin{pmatrix} 4^{|u|} & 0 & 0 \\ 0 & 4^{|v|} & 0 \\ \mathrm{int}(u) & \mathrm{int}(v) & 1 \end{pmatrix}$ as in the proof of Theorem 2, with $\mathrm{int}(u)$ being the base-4 value of u. This mapping is also an injective monoid homomorphism. Then, defining $B = \begin{pmatrix} 0 & 0 & 0 \\ -1 & 0 & -1 \\ 0 & 0 & 0 \end{pmatrix}$ which satisfies $B^2 = B$ and $BM(u,v)B = (4^{|u|} + \mathrm{int}(u) - \mathrm{int}(v))B$ we get a similar construction (cf. [5]), also suitable for showing the undecidability of the zero-in-the-upper-left-corner problem. However, the lack of substraction (in general, zero-sum-freeness of $\mathbf{N}$) prevents us to apply this method. Also, defining matrices of the form $TM(u,v)T^{-1}$ for a suitable T (as in [6], see also [14]) is again out of question since in $\mathbf{N}^{k\times k}$, only permutation matrices are invertible. The most closest approach is that of the equal entries problem: in the proof we also showed undecidability of the problem whether $\mathscr{A}$ generates a matrix having equal entries in the top-left corner and in entry $(5,5)$. Actually, the embedding $(u,v) \mapsto \begin{pmatrix} M(u) & 0 \\ 0 & M(v) \end{pmatrix}$ shows the same for 4×4 matrices. However, we were unable to modify the construction for 4×4 matrices to *shift* the values $\mathrm{int}(u)$ and $\mathrm{int}(v)$ into, say, the first column and at the same time, *overwrite* the values $3^{|u|}$ and $3^{|v|}$ by $\mathrm{int}(u)$ and $\mathrm{int}(v)$, respectively. (Adding them or something similar did not seem to work, either.) That's why we had to use 6×6 matrices – it is quite plausible that the encoding is not the most compact possible and the dimension can be further lowered.

5 Conclusion, future directions

We generalized the notion of synchronizability to automata with transitions weighted in an arbitrary semiring in two ways: one of them, location synchronizability requires the existence of a word u and a state q such that starting from any state p, q and only q has a nonzero weight after u is being read; synchronizability additionally requires that this nonzero weight is the same for all states p. In this paper we studied the *complexity* of checking these properties, parametrized by the underlying semiring.

Our results can be summarised as follows:

- Both problems are **PSPACE**-hard for any nontrivial semiring.
- For finite semirings, they are **PSPACE**-complete.
- For positive semirings, location synchronizability is **PSPACE**-complete.
- For locally finite semirings they are decidable (provided that the addition and product operations of the semiring are computable).
- The mortality problem reduces to both problems in any semiring. Thus for semirings having an undecidable mortality problem, both variants of synchronization are undecidable. (This is the case for **Z**.)
- If $(\{0,1\}^+,\cdot,\varepsilon)$ embeds into the multiplicative structure of **K**, then synchronizability is undecidable for **K**, even for deterministic automata.
- Synchronizability is undecidable for any semiring where 1 has infinite order in the additive semigroup. (This is the case for **N**. Note that for **N**, location synchronizability is in **PSPACE**.)

We do not have any decidability results for **K**-synchronizability when the semiring **K** is not locally finite, the element 1 has a finite order in the additive structure, and $\{0,1\}^+$ does not embed into the multiplicative semigroup. Also, it is not clear whether synchronizability can be reduced to location synchronizability in general – since in **N**, location synchronizability is decidable but synchronizability is undecidable, so in general, synchronizability cannot be Turing-reduced to location synchronizability. It is also an interesting question whether **N**-synchronizability of 5-state automata is decidable or not – we conjecture that it is still undecidable and one can use a slightly more compact encoding of FPCP. Also, to cover the existing generalizations of synchronizability for the case of the probabilistic semiring, we could study semirings that are equipped with a metric – our current investigations can be seen as the case of this perspective where the metric is the dicrete unit-distance metric.

Acknowledgement

The research was supported by the European Union and the State of Hungary, co-financed by the European Social Fund in the framework of TÁMOP-4.2.4.A/ 2-11/1-2012-0001 “National Excellence Program”.

The author is thankful to the anonymous referees for their suggestions and detailed comments.

References

[1] Laurent Doyen, Line Juhl, Kim G. Larsen, Nicolas Markey & Mahsa Shirmohammadi (2013): *Synchronizing Words for Timed and Weighted Automata*. Research Report LSV-13-15, Laboratoire Spécification et Vérification, ENS Cachan, France. Available at `http://www.lsv.ens-cachan.fr/Publis/RAPPORTS_LSV/PDF/rr-lsv-2013-15.pdf`. 26 pages.

[2] Laurent Doyen, Thierry Massart & Mahsa Shirmohammadi (2011): *Infinite Synchronizing Words for Probabilistic Automata*. In Filip Murlak & Piotr Sankowski, editors: *Mathematical Foundations of Computer Science 2011*, *Lecture Notes in Computer Science* 6907, Springer Berlin Heidelberg, pp. 278–289, doi:`10.1007/978-3-642-22993-0_27`.

[3] Stephane Gaubert & Ricardo Katz (2006): *Reachability Problems for Products of Matrices in Semirings*. *IJAC* 16(3), pp. 603–627. Available at `http://dblp.uni-trier.de/db/journals/ijac/ijac16.html#GaubertK06`.

[4] Zsolt Gazdag, Szabolcs Iván & Judit Nagy-György (2009): *Improved Upper Bounds on Synchronizing Nondeterministic Automata*. *Inf. Process. Lett.* 109(17), pp. 986–990, doi:`10.1016/j.ipl.2009.05.007`.

[5] Vesa Halava (1997): *Decidable and Undecidable Problems in Matrix Theory*.

[6] Vesa Halava & Tero Harju (2001): *Mortality in Matrix Semigroups*. *AMER. MATH. MONTHLY* 2001, p. 653.

[7] Vesa Halava & Mika Hirvensalo (2007): *Improved matrix pair undecidability results*. *Acta Informatica* 44(3-4), pp. 191–205, doi:`10.1007/s00236-007-0047-y`.

[8] Vesa Halava, Mika Hirvensalo & Ronald de Wolf (2001): *Marked $\{PCP\}$ is decidable*. *Theoretical Computer Science* 255(12), pp. 193 – 204, doi:`10.1016/S0304-3975(99)00163-2`.

[9] Balázs Imreh & Magnus Steinby (1999): *Directable Nondeterministic Automata*. *Acta Cybern.* 14(1), pp. 105–115. Available at `http://dblp.uni-trier.de/db/journals/actaC/actaC14.html#ImrehS99`.

[10] D.J. Kfoury (1970): *Synchronizing Sequences for Probabilistic Automata*. *Stud. Appl. Math.* 49, pp. 101–103.

[11] P. V. Martyugin (2010): *A lower bound for the length of the shortest carefully synchronizing words*. *Russian Mathematics* 54, pp. 46–54, doi:`10.3103/S1066369X10010056`.

[12] P.V. Martyugin (2010): *Complexity of Problems Concerning Carefully Synchronizing Words for PFA and Directing Words for NFA*. In Farid Ablayev & ErnstW. Mayr, editors: *Computer Science Theory and Applications*, *Lecture Notes in Computer Science* 6072, Springer Berlin Heidelberg, pp. 288–302, doi:`10.1007/978-3-642-13182-0_27`.

[13] Christos H. Papadimitriou (1994): *Computational complexity*. Addison-Wesley.

[14] M. Paterson (1970): *Unsolvability in 3×3 matrices*. *Studies in Applied Mathematics* 49, pp. 105–107.

[15] Igor Potapov (2004): *From Post Systems to the Reachability Problems for Matrix Semigroups and Multicounter Automata*. In: *Developments in Language Theory, LNCS 3340*, pp. 345–356, doi:`10.1007/978-3-540-30550-7_29`.

[16] Mikhail V. Volkov (2008): *Language and Automata Theory and Applications*. chapter Synchronizing Automata and the Černý Conjecture, Springer-Verlag, Berlin, Heidelberg, pp. 11–27, doi:`10.1007/978-3-540-88282-4_4`.

Hyper-Minimization for Deterministic Weighted Tree Automata

Andreas Maletti*
Universität Leipzig, Institute of Computer Science
Augustusplatz 10–11, 04109 Leipzig, Germany
maletti@informatik.uni-leipzig.de

Daniel Quernheim*
Universität Stuttgart, Institute for Natural Language Processing
Pfaffenwaldring 5b, 70569 Stuttgart, Germany
daniel@ims.uni-stuttgart.de

Hyper-minimization is a state reduction technique that allows a finite change in the semantics. The theory for hyper-minimization of deterministic weighted tree automata is provided. The presence of weights slightly complicates the situation in comparison to the unweighted case. In addition, the first hyper-minimization algorithm for deterministic weighted tree automata, weighted over commutative semifields, is provided together with some implementation remarks that enable an efficient implementation. In fact, the same run-time $\mathscr{O}(m \log n)$ as in the unweighted case is obtained, where m is the size of the deterministic weighted tree automaton and n is its number of states.

1 Introduction

Deterministic finite-state tree automata (DTA) [13, 14] are one of the oldest, simplest, but most useful devices in computer science representing structure. They have wide-spread applications in linguistic analysis and parsing [27] because they naturally can represent derivation trees of a context-free grammar. Due to the size of the natural language lexicons and processes like state-splitting, we often obtain huge DTA consisting of several million states. Fortunately, each DTA allows us to efficiently compute a unique (up to isomorphism) equivalent minimal DTA, which is an operation that most tree automata toolkits naturally implement. The asymptotically most efficient minimization algorithms are based on [22, 18], which in turn are based on the corresponding procedures for deterministic string automata [20, 16, 30]. In general, all those procedures compute the equivalent states and merge them in time $\mathscr{O}(m \log n)$, where n is the number of states of the input DTA and m is its size.

Hyper-minimization [3] is a state reduction technique that can reduce beyond the classical minimal device because it allows a finite change in the semantics (or a finite number of errors). It was already successfully applied to a variety of devices such as deterministic finite-state automata [12, 19], deterministic tree automata [21] as well as deterministic weighted automata [24]. With recent progress in the area of minimization for weighted deterministic tree automata [25], which provides the basis for this contribution, we revisit hyper-minimization for weighted deterministic tree automata. The asymptotically fastest hyper-minimization algorithms [12, 19] for DFA compute the "almost-equivalence" relation and merge states with finite left language, called preamble states, according to it in time $\mathscr{O}(m \log n)$, where m is the size of the input device and n is the number of its states. Naturally, this complexity is the goal for our investigation as well. Variations such as cover automata minimization [8], which has been explored before hyper-minimization due to its usefulness in compressing finite languages, or k-minimization [12] restrict the length of the error strings instead of their number, but can also be achieved within the stated time-bound.

*Both authors were financially supported by the German Research Foundation (DFG) grant MA / 4959 / 1–1.

As in [24] our weight structures will be commutative semifields, which are commutative semirings [17, 15] with multiplicative inverses. As before, we will restrict our attention to deterministic automata. Actually, the mentioned applications of DTA often use the weighted version to compute a quantitative answer (i.e., the numerically best-scoring parse, etc). We already know that weighted deterministic tree automata (DWTA) [4, 11] over semifields can be efficiently minimized [25], although the minimal equivalent DWTA is no longer unique due to the ability to "push" weights [26, 10, 25]. The asymptotically fastest minimization algorithm [25] nevertheless still runs in time $\mathscr{O}(m\log n)$. To the authors' knowledge, [25] is currently the only published algorithm achieving this complexity for DWTA. Essentially, it normalizes the input DWTA by "pushing" weights, which yields that, in the process, the signatures of equivalent states become equivalent, so that a classical unweighted minimization can then perform the computation of the equivalence and the merges. To this end, it is important that the signature ignores states that can only recognize finitely many contexts, which are called co-preamble states, to avoid computing a wrong "pushing" weight.

We focus on an almost-equivalence notion that allows the recognized weighted tree languages to differ (in weight) for finitely many trees. Thus, we join the results on unweighted hyper-minimization for DTA [21] and weighted hyper-minimization for WDFA [24]. Our algorithms (see Algorithms 1 and 2) contain features of both of their predecessors and are asymptotically as efficient as them because they also run in time $\mathscr{O}(m\log n)$. As in [28], albeit in a slightly different format, we use standardized signatures to avoid the explicit pushing of weights that was successful in [25]. This adjustment allows us to mold our weighted hyper-minimization algorithm into the structure of the unweighted algorithm [19].

2 Preliminaries

We use $\mathbb{N}$ to denote the set of all nonnegative integers (including 0). For every integer $n \in \mathbb{N}$, we use the set $[n] = \{i \in \mathbb{N} \mid 1 \leq i \leq n\}$. Given two sets S and T, their *symmetric difference* $S \ominus T$ is given by $S \ominus T = (S - T) \cup (T - S)$. An alphabet Σ is simply a finite set of symbols, and a *ranked alphabet* (Σ, rk) consists of an alphabet Σ and a ranking $\mathrm{rk}\colon \Sigma \to \mathbb{N}$. We let $\Sigma_n = \{\sigma \in \Sigma \mid \mathrm{rk}(\sigma) = n\}$ be the set of symbols of rank n for every $n \in \mathbb{N}$. We often represent the ranked alphabet (Σ, rk) by Σ alone and assume that the ranking 'rk' is implicit. Given a set T and a ranked alphabet Σ, we let

$$\Sigma(T) = \{\sigma(t_1, \ldots, t_n) \mid n \in \mathbb{N}, \sigma \in \Sigma_n, t_1, \ldots, t_n \in T\} \enspace .$$

The set $T_\Sigma(Q)$ of *Σ-trees indexed by a set Q* is the smallest set T such that $Q \cup \Sigma(T) \subseteq T$. We write T_Σ for $T_\Sigma(\emptyset)$. Given a tree $t \in T_\Sigma(Q)$, its positions $\mathrm{pos}(t) \subseteq \mathbb{N}^*$ are inductively defined by $\mathrm{pos}(q) = \{\varepsilon\}$ for each $q \in Q$ and $\mathrm{pos}(\sigma(t_1, \ldots, t_n)) = \{\varepsilon\} \cup \{iw \mid i \in [n], w \in \mathrm{pos}(t_i)\}$ for all $n \in \mathbb{N}$, $\sigma \in \Sigma_n$, and $t_1, \ldots, t_n \in T_\Sigma(Q)$. For each position $w \in \mathrm{pos}(t)$, we write $t(w)$ for the label of t at position w and $t|_w$ for the subtree of t rooted at w. Formally,

$$q(\varepsilon) = q \qquad \big(\sigma(t_1, \ldots, t_n)\big)(w) = \begin{cases} \sigma & \text{if } w = \varepsilon \\ t_i(v) & \text{if } w = iv \text{ with } i \in [n], v \in \mathbb{N}^* \end{cases}$$

$$q|_\varepsilon = q \qquad \sigma(t_1, \ldots, t_n)|_w = \begin{cases} \sigma(t_1, \ldots, t_n) & \text{if } w = \varepsilon \\ t_i|_v & \text{if } w = iv \text{ with } i \in [n], v \in \mathbb{N}^* \end{cases}$$

for all $q \in Q$, $n \in \mathbb{N}$, $\sigma \in \Sigma_n$, and $t_1, \ldots, t_n \in T_\Sigma(Q)$. The height $\mathrm{ht}(t)$ of a tree $t \in T_\Sigma(Q)$ is simply $\mathrm{ht}(t) = \max\,\{|w| \mid w \in \mathrm{pos}(t)\}$.

We reserve the use of the special symbol $\square$ of rank 0. A tree $t \in T_{\Sigma \cup \{\square\}}(Q)$ is a Σ-*context indexed by* Q if the symbol $\square$ occurs exactly once in t. The set of all Σ-contexts indexed by Q is denoted by $C_\Sigma(Q)$. As before, we write C_Σ for $C_\Sigma(\emptyset)$. For each $c \in C_\Sigma(Q)$ and $t \in T_\Sigma(Q)$, the substitution $c[t]$ denotes the tree obtained from c by replacing $\square$ by t. Similarly, we use the substitution $c[c']$ with another context $c' \in C_\Sigma(Q)$, in which case we obtain yet another context.

We take all weights from a *commutative semifield* $\langle \mathbb{S}, +, \cdot, 0, 1 \rangle$,[1] which is an algebraic structure consisting of a commutative monoid $\langle \mathbb{S}, +, 0 \rangle$ and a commutative group $\langle \mathbb{S} - \{0\}, \cdot, 1 \rangle$ such that

- $s \cdot 0 = 0$ for all $s \in \mathbb{S}$, and
- $s \cdot (s_1 + s_2) = (s \cdot s_1) + (s \cdot s_2)$ for all $s, s_1, s_2 \in \mathbb{S}$.

Roughly speaking, commutative semifields are commutative semirings [17, 15] with multiplicative inverses. Many practically relevant weight structures are commutative semifields. Examples include

- the real numbers $\langle \mathbb{R}, +, \cdot, 0, 1 \rangle$,
- the tropical semifield $\langle \mathbb{R} \cup \{\infty\}, \min, +, \infty, 0 \rangle$,
- the probabilistic semifield $\langle [0,1], \max, \cdot, 0, 1 \rangle$ with $[0,1] = \{r \in \mathbb{R} \mid 0 \leq r \leq 1\}$, and
- the BOOLEAN semifield $\mathbb{B} = \langle \{0,1\}, \max, \min, 0, 1 \rangle$.

For the rest of the paper, let $\langle \mathbb{S}, +, \cdot, 0, 1 \rangle$ be a commutative semifield (with $0 \neq 1$), and let $\underline{\mathbb{S}} = \mathbb{S} - \{0\}$. For every $s \in \underline{\mathbb{S}}$ we write s^{-1} for the inverse of s; i.e., $s \cdot s^{-1} = 1$. For better readability, we will sometimes write $\frac{s_1}{s_2}$ instead of $s_1 \cdot s_2^{-1}$. The following notions implicitly use the commutative semifield $\mathbb{S}$. A *weighted tree language* is simply a mapping $\varphi \colon T_\Sigma(Q) \to \mathbb{S}$. Its *support* $\operatorname{supp}(\varphi) \subseteq T_\Sigma(Q)$ is $\operatorname{supp}(\varphi) = \varphi^{-1}(\underline{\mathbb{S}})$; i.e., the support contains exactly those trees that are evaluated to non-zero by φ. Given $s \in \mathbb{S}$, we let $(s \cdot \varphi) \colon T_\Sigma(Q) \to \mathbb{S}$ be the weighted tree language such that $(s \cdot \varphi)(t) = s \cdot \varphi(t)$ for every $t \in T_\Sigma(Q)$.

A deterministic weighted tree automaton (DWTA) [4, 23, 6, 11] is a tuple $\mathscr{A} = (Q, \Sigma, \delta, \mathrm{wt}, F)$ with

- a finite set Q of states,
- a ranked alphabet Σ of input symbols such that $\Sigma \cap Q = \emptyset$,
- a transition mapping $\delta \colon \Sigma(Q) \to Q$,[2]
- a transition weight assignment $\mathrm{wt} \colon \Sigma(Q) \to \underline{\mathbb{S}}$, and
- a set $F \subseteq Q$ of final states.

The transition and transition weight mappings 'δ' and 'wt' naturally extend to mappings $\hat{\delta} \colon T_\Sigma(Q) \to Q$ and $\hat{\mathrm{wt}} \colon T_\Sigma(Q) \to \underline{\mathbb{S}}$ by

$$\hat{\delta}(q) = q \qquad \hat{\delta}(\sigma(t_1, \ldots, t_n)) = \delta(\sigma(\hat{\delta}(t_1), \ldots, \hat{\delta}(t_n)))$$
$$\hat{\mathrm{wt}}(q) = 1 \qquad \hat{\mathrm{wt}}(\sigma(t_1, \ldots, t_n)) = \mathrm{wt}(\sigma(\hat{\delta}(t_1), \ldots, \hat{\delta}(t_n))) \cdot \prod_{i \in [n]} \hat{\mathrm{wt}}(t_i)$$

for every $q \in Q$, $n \in \mathbb{N}$, $\sigma \in \Sigma_n$, and $t_1, \ldots, t_n \in T_\Sigma(Q)$. Since $\hat{\delta}(t) = \delta(t)$ and $\hat{\mathrm{wt}}(t) = \mathrm{wt}(t)$ for all $t \in \Sigma(Q)$, we can safely omit the hat and simply write δ and wt for $\hat{\delta}$ and $\hat{\mathrm{wt}}$, respectively. The DWTA $\mathscr{A}$ recognizes the weighted tree language $[\![\mathscr{A}]\!] \colon T_\Sigma \to \mathbb{S}$ such that

$$[\![\mathscr{A}]\!](t) = \begin{cases} \mathrm{wt}(t) & \text{if } \delta(t) \in F \\ 0 & \text{otherwise} \end{cases}$$

for all $t \in T_\Sigma$. Two DWTA $\mathscr{A}$ and $\mathscr{B}$ are equivalent if $[\![\mathscr{A}]\!] = [\![\mathscr{B}]\!]$; i.e., their recognized weighted tree languages coincide. A DWTA over the BOOLEAN semifield $\mathbb{B}$ is also called DTA [13, 14] and written

[1]We generally require $0 \neq 1$, and in fact, the additive monoid is rather irrelevant for our purposes.

[2]Note that our DWTA are always total. We additionally disallow transition weight 0. If a transition is undesired, then its transition target can be set to a sink state, which we commonly denote by $\perp$. Finally, the restriction to final states instead of final weights does not cause a difference in expressive power in our setting [6, Lemma 6.1.4].

(Q, Σ, δ, F) since the component 'wt' is uniquely determined. Moreover, we identify each BOOLEAN weighted tree language $\varphi\colon T_\Sigma(Q) \to \{0,1\}$ with its support. Finally, the set C_δ of shallow transition contexts is

$$C_\delta = \{\sigma(q_1, \dots, q_{i-1}, \square, q_{i+1}, \dots, q_n) \mid n \in \mathbb{N}, i \in [n], \sigma \in \Sigma_n, q_1, \dots, q_n \in Q\} \enspace ,$$

which we assume to be totally ordered by some arbitrary order $\leq$.

For minimization, the weighted (extended) context language of a state is relevant. For every $q \in Q$ the context-semantics $[\![q]\!]_{\mathscr{A}}\colon C_\Sigma(Q) \to \mathbb{S}$ of q is defined for every $c \in C_\Sigma(Q)$ by

$$[\![q]\!]_{\mathscr{A}}(c) = \begin{cases} \mathrm{wt}(c[q]) & \text{if } \delta(c[q]) \in F \\ 0 & \text{otherwise.} \end{cases}$$

Intuitively, $[\![q]\!]_{\mathscr{A}}$ is the weighted (extended) language recognized by $\mathscr{A}$ starting in state q. Two states $q, q' \in Q$ are equivalent [5], written $q \equiv q'$, if there exists $s \in \underline{\mathbb{S}}$ such that $[\![q]\!]_{\mathscr{A}}(c) = s \cdot [\![q']\!]_{\mathscr{A}}(c)$ for all $c \in C_\Sigma$. An equivalence relation $\cong \subseteq Q \times Q$ is a congruence relation (for the DWTA $\mathscr{A}$) if for all $n \in \mathbb{N}$, $\sigma \in \Sigma_n$, and $q_1 \cong q'_1, \dots, q_n \cong q'_n$ we have $\delta(\sigma(q_1, \dots, q_n)) \cong \delta(\sigma(q'_1, \dots, q'_n))$. It is known [5] that $\equiv$ is a congruence relation. The DWTA $\mathscr{A}$ is minimal if there is no equivalent DWTA with strictly fewer states. We can compute a minimal DWTA efficiently using a variant of HOPCROFT's algorithm [20, 18] that computes $\equiv$ and runs in time $\mathscr{O}(m \log n)$, where $m = |\Sigma(Q)|$ is the size of $\mathscr{A}$ and $n = |Q|$.

3 A characterization of hyper-minimality

Hyper-minimization [3] is a form of lossy compression that allows any finite number of errors. It has been investigated in [1, 19, 12] for deterministic finite-state automata and in [21] for deterministic tree automata. Finally, hyper-minimization was already generalized to weighted deterministic finite-state automata in [24], from which we borrow much of the general approach. In the following, let $\mathscr{A} = (Q, \Sigma, \delta, \mathrm{wt}, F)$ and $\mathscr{B} = (P, \Sigma, \mu, \mathrm{wt}', G)$ be DWTA over the commutative semifield $\langle \mathbb{S}, +, \cdot, 0, 1 \rangle$ with $0 \neq 1$.

We start with the basic definition of when two weighted tree languages are almost-equivalent. We decided to use the same approach as in [24], so we require that the weighted tree languages, seen as functions, must coincide on almost all trees. Note that this restriction is not simply the same as requiring that the weighted tree languages have almost-equal (i.e., finite-difference) supports. It fact, our definition yields that the supports are almost-equal, but that is not sufficient. In addition, we immediately allow a scaling factor in many of our basic definitions since those are already required in classical minimization [5] to obtain the most general statements. Naturally, a scaling factor is not allowed for the almost-equivalence of DWTA since these are indeed supposed assign a different weight to only finitely many trees.

Definition 1. *Two weighted tree languages $\varphi_1, \varphi_2\colon T_\Sigma(Q) \to \mathbb{S}$ are* almost-equivalent*, written $\varphi_1 \approx \varphi_2$, if there exists $s \in \underline{\mathbb{S}}$ such that $\varphi_1(t) = s \cdot \varphi_2(t)$ for almost all $t \in T_\Sigma(Q)$.*[3] *We write $\varphi_1 \approx \varphi_2 \;\; (s)$ to indicate the factor s. The* DWTA *$\mathscr{A}$ and $\mathscr{B}$ are almost-equivalent if $[\![\mathscr{A}]\!] \approx [\![\mathscr{B}]\!] \;\; (1)$. Finally, the states $q \in Q$ and $p \in P$ are almost-equivalent if there exists $s \in \underline{\mathbb{S}}$ such that $[\![q]\!]_{\mathscr{A}}(c) = s \cdot [\![p]\!]_{\mathscr{B}}(c)$ for almost all $c \in C_\Sigma$.*

We start with some basic properties of $\approx$, which is shown to be an equivalence relation both on the weighted tree languages as well as on the states of a single DWTA. In addition, we demonstrate that the

[3]"Almost all" means all but a finite number, as usual.

latter version is even a congruence relation. This shows that once we are in almost-equivalent states, the same impetus causes the different devices to switch to other almost-equivalent states.

Lemma 2. *Almost-equivalence is an equivalence relation such that $\delta(c[q]) \approx \mu(c[p])$ for all $c \in C_\Sigma$, $q \in Q$, and $p \in P$ with $q \approx p$.*

Proof. Trivially, $\approx$ is reflexive and symmetric (because we have multiplicative inverses for all elements of $\underline{\$}$). Let $\varphi_1, \varphi_2, \varphi_3 \colon T_\Sigma(Q) \to \$$ be weighted tree languages such that $\varphi_1 \approx \varphi_2\ \ (s)$ and $\varphi_2 \approx \varphi_3\ \ (s')$ for some $s, s' \in \underline{\$}$. Then there exist finite sets $L, L' \subseteq T_\Sigma(Q)$ such that $\varphi_1(t) = s \cdot \varphi_2(t)$ and $\varphi_2(t') = s' \cdot \varphi_3(t')$ for all $t \in T_\Sigma(Q) - L$ and $t' \in T_\Sigma(Q) - L'$. Consequently, $\varphi_1(t'') = s \cdot s' \cdot \varphi_3(t'')$ for all $t'' \in T_\Sigma(Q) - (L \cup L')$, which proves $\varphi_1 \approx \varphi_3\ \ (s \cdot s')$ and thus transitivity. Hence, $\approx$ is an equivalence relation. The same arguments can be used for $\approx$ on DWTA[4] and states. For the second property, induction allows us to easily prove [6] that

$$[\![q]\!]_{\mathscr{A}}(c'[c]) = \mathrm{wt}(c[q]) \cdot [\![\delta(c[q])]\!]_{\mathscr{A}}(c') \qquad \text{and} \qquad [\![p]\!]_{\mathscr{B}}(c_2[c_1]) = \mathrm{wt}'(c_1[p]) \cdot [\![\mu(c_1[p])]\!]_{\mathscr{B}}(c_2) \quad (\dagger)$$

for all $c, c' \in C_\Sigma(Q)$ and $c_1, c_2 \in C_\Sigma(P)$. Since $q \approx p\ \ (s)$, there exists a finite set $C \subseteq C_\Sigma$ such that $[\![q]\!]_{\mathscr{A}}(c'') = s \cdot [\![p]\!]_{\mathscr{B}}(c'')$ for all $c'' \in C_\Sigma - C$. Consequently,

$$[\![\delta(c[q])]\!]_{\mathscr{A}}(c') = \frac{[\![q]\!]_{\mathscr{A}}(c'[c])}{\mathrm{wt}(c[q])} = s \cdot \frac{[\![p]\!]_{\mathscr{B}}(c'[c])}{\mathrm{wt}(c[q])} = s \cdot \frac{\mathrm{wt}'(c[p])}{\mathrm{wt}(c[q])} \cdot [\![\mu(c[p])]\!]_{\mathscr{B}}(c')$$

for all $c' \in C_\Sigma$ such that $c'[c] \notin C$, which proves that $\delta(c[q]) \approx \mu(c[p])$. □

Next, we show that almost-equivalent states of the same DWTA even coincide (up to the factor s) on almost all extended contexts, which are contexts in which states may occur.

Lemma 3. *Let $\mathscr{A}$ be minimal and $q \approx q'\ \ (s)$ for some $s \in \underline{\$}$ and $q, q' \in Q$. Then*

$$[\![q]\!]_{\mathscr{A}}(c) = s \cdot [\![q']\!]_{\mathscr{A}}(c) \qquad (\ddagger)$$

for almost all $c \in C_\Sigma(Q)$.

Proof. By definition of $q \approx q'\ \ (s)$, there exists a finite set $C \subseteq C_\Sigma$ such that $[\![q]\!]_{\mathscr{A}}(c) = s \cdot [\![q']\!]_{\mathscr{A}}(c)$ for all $c \in C_\Sigma - C$. Let $h \geq \max\ \{\mathrm{ht}(c) \mid c \in C\}$ be an upper bound for the height of those finitely many contexts. Clearly, there are only finitely many contexts of C_Σ that have height at most h. Now, let $c \in C_\Sigma(Q)$ be an extended context such that $\mathrm{ht}(c) > h$, and let $W = \{w \in \mathrm{pos}(c) \mid c(w) \in Q\}$ be the positions that are labeled with states. For each state $q \in Q$, select $t_q \in \delta^{-1}(q) \cap T_\Sigma$ a tree (without occurrences of states) that is processed in q. Clearly, such a tree exists for each state because $\mathscr{A}$ is minimal. Let c' be the context obtained from c by replacing each state occurrence of q by t_q. Obviously, $\mathrm{ht}(c') \geq \mathrm{ht}(c) > h$ because we replace only leaves. Consequently, $c' \in C_\Sigma - C$. Using a variant [6] of (†) we obtain

$$[\![q]\!]_{\mathscr{A}}(c) \cdot \prod_{w \in W} \mathrm{wt}(t_{c(w)}) = [\![q]\!]_{\mathscr{A}}(c') = s \cdot [\![q']\!]_{\mathscr{A}}(c') = s \cdot [\![q']\!]_{\mathscr{A}}(c) \cdot \prod_{w \in W} \mathrm{wt}(t_{c(w)}) \ ,$$

where the second equality is due to the fact that $c' \in C_\Sigma - C$. Comparing the left-hand and right-hand side and cancelling the additional terms, which is allowed in a commutative semifield, we obtain $[\![q]\!]_{\mathscr{A}}(c) = s \cdot [\![q']\!]_{\mathscr{A}}(c)$ for all $c \in C_\Sigma(Q)$ with $\mathrm{ht}(c) > h$, and thus for almost all $c \in C_\Sigma(Q)$ as required. □

[4]Note that $1^{-1} = 1$ and $1 \cdot 1 = 1$, so the restriction to factor 1 in the definition of the almost-equivalence of DWTA is not problematic.

As in all the other scenarios, the goal of hyper-minimization given device $\mathscr{A}$ is to construct an almost-equivalent device $\mathscr{B}$ such that no device is smaller than $\mathscr{B}$ and almost-equivalent to $\mathscr{A}$. In our setting, the devices are DWTA over the ranked alphabet Σ and the commutative semifield $\mathbb{S}$. Since almost-equivalence is an equivalence relation by Lemma 2, we can replace the requirement "almost-equivalent to $\mathscr{A}$" by "almost-equivalent to $\mathscr{B}$" and call a DWTA $\mathscr{B}$ *hyper-minimal* if no (strictly) smaller DWTA is almost-equivalent to it. Then hyper-minimization equates to the computation of a hyper-minimal DWTA $\mathscr{B}$ that is almost-equivalent to $\mathscr{A}$. Let us first investigate hyper-minimality, which was characterized in [3] for the BOOLEAN semifield using the additional notion of a *preamble* state.

Definition 4 (see [3, Definition 2.11]). *A state $q \in Q$ is a* preamble state *if $\delta^{-1}(q) \cap T_\Sigma$ is finite. Otherwise, it is a* kernel state.

In other words, a state is a preamble state if and only if it accepts finitely many trees (without occurrences of states). This notion is essentially unweighted, so the discussion in [21] applies. In particular, we can compute the set of kernel states in time $\mathscr{O}(m)$ with $m = |\Sigma(Q)|$ being the size of the DWTA $\mathscr{A}$.

Recall that a DWTA (without unreachable states; i.e., $\delta^{-1}(q) \cap T_\Sigma \neq \emptyset$ for every $q \in Q$) is minimal if and only if it does not have a pair of different, but equivalent states [7, 5]. The "only-if" part of this statement is shown by merging two equivalent states to obtain a smaller, but equivalent DWTA. Let us define a merge that additionally applies a weight s to the rerouted transitions.

Definition 5. *Let $q, q' \in Q$ and $s \in \underline{\mathbb{S}}$ with $q \neq q'$. The s-*weighted merge *of q into q' is the* DWTA $\mathrm{merge}_{\mathscr{A}}(q \xrightarrow{s} q') = (Q - \{q\}, \Sigma, \delta', \mathrm{wt}', F - \{q\})$ *such that for all $t \in \Sigma(Q - \{q\})$*

$$\delta'(t) = \begin{cases} q' & \text{if } \delta(t) = q \\ \delta(t) & \text{otherwise} \end{cases} \qquad\qquad \mathrm{wt}'(t) = \begin{cases} s \cdot \mathrm{wt}(t) & \text{if } \delta(t) = q \\ \mathrm{wt}(t) & \text{otherwise.} \end{cases}$$

In our approach to weighted hyper-minimization, we also merge, but we need to take care of the factors, so we use the weighted merges just introduced. The next lemma hints at the correct use of weighted merges.

Lemma 6. *Let $q, q' \in Q$ be different states, of which q is a preamble state, and $s \in \underline{\mathbb{S}}$ be such that $q \approx q'$ (s). Then $\mathrm{merge}_{\mathscr{A}}(q \xrightarrow{s} q')$ is almost-equivalent to $\mathscr{A}$.*

Proof. Since $q \approx q'$ (s), there exists a finite set $C \subseteq C_\Sigma$ such that $[\![q]\!]_{\mathscr{A}}(c) = s \cdot [\![q']\!]_{\mathscr{A}}(c)$ for all $c \in C_\Sigma - C$. Let $h \geq \max\,\{\mathrm{ht}(c) \mid c \in C\}$ be an upper bound on the height of the contexts of C. Moreover, let $h' \geq \max\,\{\mathrm{ht}(t) \mid t \in \delta^{-1}(q) \cap T_\Sigma\}$ be an upper bound for the height of the trees of $\delta^{-1}(q) \cap T_\Sigma$, which is a finite set since q is a preamble state. Finally, let $z > h + h'$. Now we return to the main claim. Let $\mathscr{B} = \mathrm{merge}_{\mathscr{A}}(q \xrightarrow{s} q')$ and consider an arbitrary tree $t \in T_\Sigma$ whose height is at least z. Clearly, showing that $\mathscr{B}(t) = \mathscr{A}(t)$ for all trees t with $\mathrm{ht}(t) \geq z$ proves that $\mathscr{B}$ and $\mathscr{A}$ are almost-equivalent.[5] Let $W = \{w \in \mathrm{pos}(t) \mid \delta(t|_w) = q\}$ be the set of positions of the subtrees that are recognized in state q. Now $\mathrm{wt}'(t|_w) = s \cdot \mathrm{wt}(t|_w)$ for all $w \in W$ because clearly the subtrees $t|_w$ only use states different from q except at the root, where $\mathscr{A}$ switches to q and $\mathscr{B}$ switches to q' with the additional weight s. Note that q cannot occur anywhere else inside those subtrees because this would create a loop which is impossible for a preamble state. Let $W = \{w_1, \dots, w_m\}$ with $w_1 \sqsubset \cdots \sqsubset w_m$, in which $\sqsubseteq$ is the lexicographic order on $\mathbb{N}^*$. Let $c_1 \in C_\Sigma$ be the context obtained by removing the subtree at w_1 from t. Note that c_1 is taller

[5] There are only finitely many ranked trees up to a certain height and recall that almost-equivalence does not permit a scaling factor for DWTA.

than h (i.e., $\mathrm{ht}(c_1) > h$) and thus $c_1 \in C_\Sigma - C$ because the height of t is larger than $h + h'$ and the height of $t|_{w_1}$ is at most h'. Consequently, using a variant [6] of (†) we obtain

$$\mathscr{A}(t) = \mathscr{A}(c_1[t|_{w_1}]) \overset{\dagger}{=} \mathrm{wt}(t|_{w_1}) \cdot [\![q]\!]_{\mathscr{A}}(c_1) = \frac{\mathrm{wt}'(t|_{w_1})}{s} \cdot s \cdot [\![q']\!]_{\mathscr{A}}(c_1) = \mathrm{wt}'(t|_{w_1}) \cdot [\![q']\!]_{\mathscr{A}}(c_1)$$
$$= \mathrm{wt}'(t|_{w_1}) \cdot \begin{cases} \mathrm{wt}(c_1[q']) & \text{if } \delta(c_1[q']) \in F \\ 0 & \text{otherwise.} \end{cases}$$

Let c_2 be the context obtained from $c_1[q']$ by replacing the subtree at w_2 by $\Box$. Also $c_2 \notin C$.

$$= \mathrm{wt}'(t|_{w_1}) \cdot \begin{cases} \mathrm{wt}(c_2[t|_{w_2}]) & \text{if } \delta(c_2[t|_{w_2}]) \in F \\ 0 & \text{otherwise.} \end{cases} \quad = \mathrm{wt}'(t|_{w_1}) \cdot \mathrm{wt}(t|_{w_2}) \cdot [\![q]\!]_{\mathscr{A}}(c_2)$$
$$\overset{\ddagger}{=} \mathrm{wt}'(t|_{w_1}) \cdot \frac{\mathrm{wt}'(t|_{w_2})}{s} \cdot s \cdot [\![q']\!]_{\mathscr{A}}(c_2) = \mathrm{wt}'(t|_{w_1}) \cdot \mathrm{wt}'(t|_{w_2}) \cdot [\![q']\!]_{\mathscr{A}}(c_2) \enspace ,$$

which can now be iterated to obtain

$$= \mathrm{wt}'(t|_{w_1}) \cdot \ldots \cdot \mathrm{wt}'(t|_{w_m}) \cdot [\![q']\!]_{\mathscr{A}}(c_m) = \mathrm{wt}'(t|_{w_1}) \cdot \ldots \cdot \mathrm{wt}'(t|_{w_m}) \cdot [\![q']\!]_{\mathscr{B}}(c_m) \overset{\dagger}{=} \mathscr{B}(t) \enspace ,$$

where the second-to-last step is justified because the state q is not used when processing the context c_m. This proves the statement. □

Theorem 7. *A minimal* DWTA *is hyper-minimal if and only if it has no pair of different, but almost-equivalent states, of which at least one is a preamble state.*

Proof. Let $\mathscr{A}$ be the minimal DWTA. For the "only if" part, we know by Lemma 6 that the smaller DWTA $\mathrm{merge}_{\mathscr{A}}(q \xrightarrow{s} q')$ is almost-equivalent to $\mathscr{A}$ if $q \approx q'$ (s) and q is a preamble state. For the "if" direction, suppose that $\mathscr{B}$ is almost-equivalent to $\mathscr{A}$ and $|P| < |Q|$.[6] For all $t \in T_\Sigma$ we have $\delta(t) \approx \mu(t)$ by Lemma 2. Since $|P| < |Q|$, there exist $t_1, t_2 \in T_\Sigma$ with $q_1 = \delta(t_1) \neq \delta(t_2) = q_2$ but $\mu(t_1) = p = \mu(t_2)$. Consequently, $q_1 = \delta(t_1) \approx \mu(t_1) = p = \mu(t_2) \approx \delta(t_2) = q_2$, which yields $q_1 \approx q_2$. By assumption, q_1 and q_2 are kernel states. Using a variation of the above argument (see [3, Theorem 3.3]) we can obtain t_1 and t_2 with the above properties such that $\mathrm{ht}(t_1), \mathrm{ht}(t_2) \geq |Q|^2$. Due to their heights, we can pump the trees t_1 and t_2, which yields that the states $\langle q_1, p\rangle$ and $\langle q_2, p\rangle$ are kernel states of the HADAMARD product $\mathscr{A} \cdot \mathscr{B}$. Since $\mathscr{A}$ and $\mathscr{B}$ are almost-equivalent, we have

$$\mathrm{wt}(t_1) \cdot [\![q_1]\!]_{\mathscr{A}}(c) \overset{\dagger}{=} [\![\mathscr{A}]\!](c[t_1]) = [\![\mathscr{B}]\!](c[t_1]) \overset{\dagger}{=} \mathrm{wt}'(t_1) \cdot [\![p]\!]_{\mathscr{B}}(c)$$
$$\mathrm{wt}(t_2) \cdot [\![q_2]\!]_{\mathscr{A}}(c) \overset{\dagger}{=} [\![\mathscr{A}]\!](c[t_2]) = [\![\mathscr{B}]\!](c[t_2]) \overset{\dagger}{=} \mathrm{wt}'(t_2) \cdot [\![p]\!]_{\mathscr{B}}(c)$$

for almost all $c \in C_\Sigma$ using again the tree variant of (†). Moreover, since both $\langle q_1, p\rangle$ and $\langle q_2, p\rangle$ are kernel states, we can select t_1 and t_2 such that the previous statements are actually true for all $c \in C_\Sigma$. Consequently,

$$\frac{\mathrm{wt}(t_1) \cdot [\![q_1]\!]_{\mathscr{A}}(c)}{\mathrm{wt}'(t_1)} = \frac{\mathrm{wt}(t_2) \cdot [\![q_2]\!]_{\mathscr{A}}(c)}{\mathrm{wt}'(t_2)} \quad \text{and} \quad [\![q_1]\!]_{\mathscr{A}}(c) = s \cdot [\![q_2]\!]_{\mathscr{A}}(c)$$

for all $c \in C_\Sigma$ and $s = \frac{\mathrm{wt}'(t_1) \cdot \mathrm{wt}(t_2)}{\mathrm{wt}'(t_2) \cdot \mathrm{wt}(t_1)}$, which yields $q_1 \equiv q_2$. This contradicts minimality since $q_1 \neq q_2$, which shows that such a DWTA $\mathscr{B}$ cannot exist. □

[6] Recall that almost-equivalent DWTA do not permit a scaling factor; their semantics need to coincide for almost all trees.

Algorithm 1 Structure of the hyper-minimization algorithm.

Require: a DWTA $\mathscr{A}$ with n states
Return: an almost-equivalent hyper-minimal DWTA

```
   $\mathscr{A} \leftarrow$ MINIMIZE($\mathscr{A}$)                              // $\mathscr{O}(m \log n)$
2: $K \leftarrow$ COMPUTEKERNEL($\mathscr{A}$)                         // $\mathscr{O}(m)$
   $\overline{K} \leftarrow$ COMPUTECOKERNEL($\mathscr{A}$)                       // $\mathscr{O}(m)$
4: $(\sim, t) \leftarrow$ COMPUTEALMOSTEQUIVALENCE($\mathscr{A}, \overline{K}$)   // Algorithm 2 — $\mathscr{O}(m \log n)$
   return MERGESTATES($\mathscr{A}, K, \sim, t$)                  // Algorithm 3 — $\mathscr{O}(m)$
```

4 Hyper-minimization

Next, we consider some algorithmic aspects of hyper-minimization for DWTA. Since the unweighted case is already well-described in the literature [21], we focus on the weighted case, for which we need the additional notion of co-preamble states [24], which in analogy to [24] are those states with finite support of their weighted context language. Let P and K be the sets of preamble and kernel states of $\mathscr{A}$, respectively.

Definition 8. *A state $q \in Q$ is a* co-preamble state *if* $\operatorname{supp}([\![q]\!]_{\mathscr{A}})$ *is finite. Otherwise it is a* co-kernel state*. The sets of all co-preamble states and all co-kernel states are $\overline{P}$ and $\overline{K} = Q - \overline{P}$, respectively.*

Transitions entering a co-preamble state can be ignored while checking almost-equivalence because (up to a finite number of weight differences) the reached states behave like the sink state $\perp$. Trivially, all co-preamble states are almost-equivalent. In addition, a co-preamble state cannot be almost-equivalent to a co-kernel state. The interesting part of the almost-equivalence is thus completely determined by the weighted languages of the co-kernel states. This special role of the co-preamble states has already been pointed out in [12] in the context of DFA.

All hyper-minimization algorithms [3, 2, 12, 19] share the same overall structure (Algorithm 1). In the final step we perform state merges (see Definition 5). Merging only preamble states into almost-equivalent states makes sure that the resulting DWTA is almost-equivalent to the input DWTA by Lemma 6. Algorithm 1 first minimizes the input DWTA using, for example, the algorithm of [25]. With the help of a weight redistribution along the transitions (pushing), it reduces the problem to DTA minimization, for which we can use a variant of HOPCROFT's algorithm [18]. In the next step, we compute the set K of kernel states of $\mathscr{A}$ [21] using any algorithm that computes strongly connected components (for example, TARJAN's algorithm [29]). By [21] a state is a kernel state if and only if it is reachable from (i) a nontrivial strongly connected component or (ii) a state with a self-loop. Essentially, the same approach can be used to compute the co-kernel states. In line 4 we compute the almost-equivalence on the states Q, which is the part where the algorithms [3, 2, 12, 19] differ. Finally, we merge almost-equivalent states according to Lemma 6 until the obtained DWTA is hyper-minimal (see Theorem 7).

Lemma 9. *Let $\mathscr{A}$ be a minimal* DWTA*. The states $q, q' \in Q$ are almost-equivalent if and only if there is $n \in \mathbb{N}$ such that $\delta(c[q]) = \delta(c[q'])$ for all $c \in C_\Sigma$ such that $\square$ occurs at position w in c with $|w| \geq n$.*

Our algorithm for computing the almost-equivalence is an extension of the algorithm of [24]. As in [24], we need to handle the scaling factors, for which we introduced the standardized signature in [24]. Roughly speaking, we ignore transitions into co-preamble states and normalize the transition weights. Recall that C_δ is the set of transition contexts; i.e., transitions with exactly one occurrence of the symbol $\square$. Moreover, for every $q \in Q$, we let c_q be the smallest transition context $c_q \in C_\delta$ such that

$\delta(c_q[q]) \in \overline{K}$, where the total order on C_δ is arbitrary as assumed earlier, but it needs to be consistently used.

Definition 10. *Given $q \in Q$, its* standardized signature *is*

$$\mathrm{Sig}(q) = \left\{ \langle c, \delta(c[q]), \frac{\mathrm{wt}(c[q])}{\mathrm{wt}(c_q[q])} \rangle \;\middle|\; c \in C_\delta,\, \delta(c[q]) \in \overline{K} \right\} \enspace .$$

Next, we show that states with equal standardized signature are indeed almost-equivalent.

Lemma 11. *For all $q, q' \in Q$, if $\mathrm{Sig}(q) = \mathrm{Sig}(q')$, then $q \approx q'$.*

Proof. If q or q' is a co-preamble state, then both q and q' are co-preamble states and thus $q \approx q'$. Now, let $q, q' \in \overline{K}$, and let $c_q \in C_\delta$ be the smallest transition context such that $c_q[q] \in \overline{K}$. Since q' has the same signature, $c_q = c_{q'}$. In addition, let $s = \frac{\mathrm{wt}(c_q[q])}{\mathrm{wt}(c_q[q'])}$. For every $c \in C_\delta$ and $c' \in C_\Sigma$,

$$[\![q]\!]_{\mathscr{A}}(c'[c]) \stackrel{\dagger}{=} \mathrm{wt}(c[q]) \cdot [\![\delta(c[q])]\!]_{\mathscr{A}}(c') \quad \text{and} \quad [\![q']\!]_{\mathscr{A}}(c'[c]) \stackrel{\dagger}{=} \mathrm{wt}(c[q']) \cdot [\![\delta(c[q'])]\!]_{\mathscr{A}}(c') \enspace .$$

First, let $\langle c, q_c, s_c \rangle \notin \mathrm{Sig}(q) = \mathrm{Sig}(q')$ for all $q_c \in Q$ and $s_c \in \underline{\mathbb{S}}$. Then c takes both q and q' into a co-preamble state and thus $[\![q]\!]_{\mathscr{A}}(c'[c]) = 0 = s \cdot [\![q']\!]_{\mathscr{A}}(c'[c])$ for almost all $c' \in C_\Sigma$. Second, suppose that $\langle c, q_c, s_c \rangle \in \mathrm{Sig}(q) = \mathrm{Sig}(q')$ for some $q_c \in Q$ and $s_c \in \underline{\mathbb{S}}$. Since $\delta(c[q]) = q_c = \delta(c[q'])$, and we obtain

$$\begin{aligned} [\![q]\!]_{\mathscr{A}}(c'[c]) &= \frac{\mathrm{wt}(c[q])}{\mathrm{wt}(c_q[q])} \cdot \mathrm{wt}(c_q[q]) \cdot [\![q_c]\!]_{\mathscr{A}}(c') \;= s_c \cdot \mathrm{wt}(c_q[q]) \cdot [\![q_c]\!]_{\mathscr{A}}(c') \\ &= \frac{\mathrm{wt}(c[q'])}{\mathrm{wt}(c_q[q'])} \cdot \mathrm{wt}(c_q[q]) \cdot [\![q_c]\!]_{\mathscr{A}}(c') = s \cdot [\![q']\!]_{\mathscr{A}}(c'[c]) \end{aligned}$$

for every $c' \in C_\Sigma$, which shows that $q \approx q' \;\; (s)$ because the scaling factor s does not depend on the transition context c. □

In fact, the previous proof can also be used to show that at most the empty context □ yields a difference in the weighted context languages $[\![q]\!]_{\mathscr{A}}$ and $[\![q']\!]_{\mathscr{A}}$ (up to the common factor). For the completeness, we also need a (restricted) converse for minimal DWTA, which shows that as long as there are almost-equivalent states, we can also identify them using the standardized signature.

Lemma 12. *Let $\mathscr{A}$ be minimal, and let $q \approx q'$ be such that $\mathrm{Sig}(q) \neq \mathrm{Sig}(q')$. Then there exist $r, r' \in Q$ such that $r \neq r'$ and $\mathrm{Sig}(r) = \mathrm{Sig}(r')$.*

Proof. Since $q \approx q'$, there exists an integer h such that $\delta(c[q]) = \delta(c[q'])$ for all $c \in C_\Sigma$ such that $w \in \mathrm{pos}(c)$ with $c(w) = \square$ and $|w| \geq h$ by Lemma 9. Let $c' \in C_\Sigma$ be a maximal context such that $r = \delta(c'[q]) \neq \delta(c'[q']) = r'$. Since c' is maximal, we have $\delta(c''[c'[q]]) = q_{c''} = \delta(c''[c'[q']])$ for all $c'' \in C_\delta$. If $q_{c''}$ is a co-preamble state, then $\langle c, q_c, s_c \rangle \notin \mathrm{Sig}(r) = \mathrm{Sig}(r')$ for all $q_c \in Q$ and $s_c \in \underline{\mathbb{S}}$. On the other hand, let $q_{c''}$ be a co-kernel state, and let $c_r \in C_\delta$ be the smallest transition context such that $\delta(c_r[r]) \in \overline{K}$. Since $q \approx q'$ and $\approx$ is a congruence relation by Lemma 2, we have $r \approx r' \;\; (s)$ for some $s \in \underline{\mathbb{S}}$, which means that $[\![r]\!]_{\mathscr{A}}(c) = s \cdot [\![r']\!]_{\mathscr{A}}(c)$ for almost all $c \in C_\Sigma$. Consequently,

$$\begin{aligned} \mathrm{wt}(c''[r]) \cdot [\![q_{c''}]\!]_{\mathscr{A}}(c) &= s \cdot \mathrm{wt}(c''[r']) \cdot [\![q_{c''}]\!]_{\mathscr{A}}(c) \\ \mathrm{wt}(c_r[r]) \cdot [\![\delta(c_r[r])]\!]_{\mathscr{A}}(c) &= s \cdot \mathrm{wt}(c_r[r']) \cdot [\![\delta(c_r[r])]\!]_{\mathscr{A}}(c) \end{aligned}$$

Algorithm 2 Algorithm computing the almost-equivalence $\approx$ and scaling map f.

Require: minimal DWTA $\mathscr{A}$ and its co-kernel states $\overline{K}$
Return: almost-equivalence $\approx$ as a partition and scaling map $f\colon Q \to \underline{\mathbb{K}}$

```
    for all $q \in Q$ do
 2:     $\pi(q) \leftarrow \{q\}$; $f(q) \leftarrow 1$                         // trivial initial blocks
    $h \leftarrow \emptyset$; $I \leftarrow Q$                                  // hash map of type $h\colon$ Sig $\to Q$
 4: for all $q \in I$ do
        succ $\leftarrow$ Sig$(q)$          // compute standardized signature using current $\delta$ and $\overline{K}$
 6:     if HasValue$(h, \text{succ})$ then
            $q' \leftarrow$ Get$(h, \text{succ})$                          // retrieve state in bucket 'succ' of $h$
 8:         if $|\pi(q')| \geq |\pi(q)|$ then
                Swap$(q, q')$                                          // exchange roles of $q$ and $q'$
10:         $I \leftarrow I \cup \{r \in Q - \{q'\} \mid \exists c \in C_\delta\colon \delta(c[r]) = q'\}$      // add predecessors of $q'$
            $f(q') \leftarrow \frac{\text{wt}(c_q[q'])}{\text{wt}(c_q[q])}$                                // $c_q$ is as in Definition 10
12:         $\mathscr{A} \leftarrow \text{merge}_{\mathscr{A}}(q' \overset{f(q')}{\to} q)$                              // merge $q'$ into $q$
            $\pi(q) \leftarrow \pi(q) \cup \pi(q')$                          // $q$ and $q'$ are almost-equivalent
14:         for all $r \in \pi(q')$ do
                $f(r) \leftarrow f(r) \cdot f(q')$                           // recompute scaling factors
16:     $h \leftarrow$ Put$(h, \text{succ}, q)$                              // store $q$ in $h$ under key 'succ'
    return $(\pi, f)$
```

for almost all $c \in C_\Sigma$. Since both $q_{c''}$ and $\delta(c_r[r])$ are co-kernel states, we immediately can conclude that $\text{wt}(c''[r]) = s \cdot \text{wt}(c''[r'])$ and $\text{wt}(c_r[r]) = s \cdot \text{wt}(c_r[r'])$, which yields

$$\frac{\text{wt}(c''[r])}{\text{wt}(c_r[r])} = \frac{s \cdot \text{wt}(c''[r'])}{s \cdot \text{wt}(c_r[r'])} = \frac{\text{wt}(c''[r'])}{\text{wt}(c_r[r'])} \quad .$$

This proves $\text{Sig}(r) = \text{Sig}(r')$ as required. □

Lemmata 11 and 12 suggest Algorithm 2 for computing the almost-equivalence and a map representing the scaling factors. This map contains a scaling factor for each state with respect to a representative state of its block. Algorithm 2 is a straightforward modification of an algorithm by [19] using our standardized signatures. We first compute the standardized signature for each state and store it into a (perfect) hash map [9] to avoid pairwise comparisons. If we find a collision (i.e., a pair of states with the same signature), then we merge them such that the state representing the bigger block survives (see Lines 9 and 12). Each state is considered at most $\log n$ times because the size of the "losing" block containing it at least doubles. After each merge, scaling factors of the "losing" block are computed with respect to the new representative. Again, we only recompute the scaling factor of each state at most $\log n$ times. Hence the small modifications compared to [19] do not increase the asymptotic run-time of Algorithm 2, which is $\mathscr{O}(n \log n)$ where n is the number of states (see Theorem 9 in [19]). Alternatively, we can use the standard reduction to a weighted finite-state automaton using each transition context $c \in C_\delta$ as a new symbol.

Algorithm 3 Merging almost-equivalent states.

Require: a minimal DWTA $\mathscr{A}$, its kernel states K, its almost-equivalence $\approx$, and a scaling map $f\colon Q \to \underline{\mathbb{S}}$
Return: hyper-minimal DWTA $\mathscr{A}$ that is almost-equivalent to the input DWTA

```
    for all $B \in (Q/\approx)$ do
 2:     select $q \in B$ such that $q \in K$ if possible
        for all $q' \in B - K$ do
 4:         $\mathscr{A} \leftarrow \text{merge}_{\mathscr{A}}(q' \xrightarrow{\frac{f(q')}{f(q)}} q)$
```

Proposition 13. *Algorithm 2 can be implemented to run in time $\mathscr{O}(m \log n)$, where $m = |\Sigma(Q)|$ and $n = |Q|$.*

Finally, we need an adjusted merging process that takes the scaling factors into account. When merging one state into another, their mutual scaling factor can be computed from the scaling map by multiplicaton of one scaling factor with the inverse of the other. Therefore, merging (see Algorithm 3) can be implemented in time $\mathscr{O}(n)$, and hyper-minimization (Algorithm 1) can be implemented in time $\mathscr{O}(m \log n)$ in the weighted setting.

Proposition 14. *Our hyper-minimization algorithm can be implemented to run in time $\mathscr{O}(m \log n)$.*

It remains to prove the correctness of our algorithm. To prove the correctness of Algorithm 2, we still need a technical property.

Lemma 15. *Let $q, q' \in Q$ be states with $q \neq q'$ but $\text{Sig}(q) = \text{Sig}(q')$. Moreover, let $\mathscr{B} = \text{merge}_{\mathscr{A}}(q' \xrightarrow{s} q)$ with $s = \frac{f(q')}{f(q)}$, and let $\cong$ be its almost-equivalence (restricted to P). Then $\cong\, = \,\approx \cap (P \times P)$ where $P = Q - \{q'\}$.*

Proof. Let $p_1 \approx p_2$ with $p_1, p_2 \in P$. Let $c = c_\ell[c_{\ell-1}[\cdots[c_1]\cdots]]$ with $c_1, \dots, c_\ell \in C_\delta$. Then we obtain the runs

$$R_{p_1} = \langle \delta(c_1[p_1]), \delta(c_2[c_1[p_1]]), \cdots, \delta(c[p_1]) \rangle \quad \text{with weight } \text{wt}(c[p_1])$$
$$R_{p_2} = \langle \delta(c_1[p_2]), \delta(c_2[c_1[p_2]]), \cdots, \delta(c[p_2]) \rangle \quad \text{with weight } \text{wt}(c[p_2]).$$

The corresponding runs R'_{p_1} and R'_{p_2} in $\mathscr{B}$ replace every occurrence of q' in both R_{p_1} and R_{p_2} by q. Their weights are

$$\text{wt}'(c[p_1]) = \begin{cases} \text{wt}(c[p_1]) & \text{if } \delta(c[p_1]) \neq q' \\ \text{wt}(c[p_1]) \cdot s & \text{otherwise} \end{cases}$$
$$\text{wt}'(c[p_2]) = \begin{cases} \text{wt}(c[p_2]) & \text{if } \delta(c[p_2]) \neq q' \\ \text{wt}(c[p_2]) \cdot s & \text{otherwise.} \end{cases}$$

Since $\delta(c'[p_1]) = \delta(c'[p_2])$ for suitably tall contexts $c' \in C_\Sigma$ and $p_1 \approx p_2$, we obtain that $p_1 \cong p_2$. The same reasoning can be used to prove the converse. □

Theorem 16. *Algorithm 2 computes $\approx$ and a scaling map.*

Proof sketch. If there exist different, but almost-equivalent states, then there exist different states with the same standardized signature by Lemma 12. Lemma 11 shows that such states are almost-equivalent.

Finally, Lemma 15 shows that we can continue the computation of the almost-equivalence after a weighted merge of such states. The correctness of the scaling map is shown implicitly in the proof of Lemma 11. □

Theorem 17. *We can hyper-minimize* DWTA *in time* $\mathcal{O}(m \log n)$*, where* $m = |\Sigma(Q)|$ *and* $n = |Q|$.

References

[1] Andrew Badr (2008): *Hyper-Minimization in* $O(n^2)$. In: Proc. 13th CIAA, LNCS 5148, Springer, pp. 223–231, doi:10.1007/978-3-540-70844-5_23.

[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] Andrew Badr, Viliam Geffert & Ian Shipman (2009): *Hyper-minimizing minimized deterministic finite state automata*. RAIRO Theor. Inf. Appl. 43(1), pp. 69–94, doi:10.1051/ita:2007061.

[4] Jean Berstel & Christophe Reutenauer (1982): *Recognizable Formal Power Series on Trees*. Theor. Comput. Sci. 18(2), pp. 115–148, doi:10.1016/0304-3975(82)90019-6.

[5] Björn Borchardt (2003): *The Myhill-Nerode Theorem for Recognizable Tree Series*. In: Proc. 7th DLT, LNCS 2710, Springer, pp. 146–158, doi:10.1007/3-540-45007-6_11.

[6] Björn Borchardt (2005): *The Theory of Recognizable Tree Series*. Ph.D. thesis, Technische Universität Dresden.

[7] Walter S. Brainerd (1968): *The Minimalization of Tree Automata*. Information and Control 13(5), pp. 484–491, doi:10.1016/S0019-9958(68)90917-0.

[8] Cezar Câmpeanu, Nicolae Santean & Sheng Yu (2001): *Minimal cover-automata for finite languages*. Theor. Comput. Sci. 267(1–2), pp. 3–16, doi:10.1016/S0304-3975(00)00292-9.

[9] Martin Dietzfelbinger, Anna R. Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert & Robert Endre Tarjan (1994): *Dynamic Perfect Hashing: Upper and Lower Bounds*. SIAM J. Comput. 23(4), pp. 738–761, doi:10.1137/S0097539791194094.

[10] Jason Eisner (2003): *Simpler and More General Minimization for Weighted Finite-State Automata*. In: Proc. HLT-NAACL, The The Association for Computational Linguistics, pp. 64–71.

[11] Zoltán Fülöp & Heiko Vogler (2009): *Weighted tree automata and tree transducers*. In Manfred Droste, Werner Kuich & Heiko Vogler, editors: Handbook of Weighted Automata, chapter IX, EATCS Monographs on Theoret. Comput. Sci., Springer, pp. 313–403, doi:10.1007/978-3-642-01492-5_9.

[12] Paweł Gawrychowski & Artur Jeż (2009): *Hyper-minimisation Made Efficient*. In: Proc. 34th MFCS, LNCS 5734, Springer, pp. 356–368, doi:10.1007/978-3-642-03816-7_31.

[13] Ferenc Gécseg & Magnus Steinby (1984): *Tree Automata*. Akadémiai Kiadó, Budapest.

[14] Ferenc Gécseg & Magnus Steinby (1997): *Tree Languages*. In Grzegorz Rozenberg & Arto Salomaa, editors: Handbook of Formal Languages, chapter 1, 3, Springer, pp. 1–68, doi:10.1007/978-3-642-59126-6_1.

[15] Jonathan S. Golan (1999): *Semirings and their Applications*. Kluwer Academic, Dordrecht, doi:10.1007/978-94-015-9333-5.

[16] David Gries (1973): *Describing an Algorithm by Hopcroft*. Acta Inform. 2(2), pp. 97–109, doi:10.1007/BF00264025.

[17] Udo Hebisch & Hanns J. Weinert (1998): *Semirings — Algebraic Theory and Applications in Computer Science*. World Scientific, doi:10.1142/9789812815965_0001.

[18] Johanna Högberg, Andreas Maletti & Jonathan May (2009): *Backward and Forward Bisimulation Minimization of Tree Automata*. Theor. Comput. Sci. 410(37), pp. 3539–3552, doi:10.1016/j.tcs.2009.03.022.

[19] Markus Holzer & Andreas Maletti (2010): *An $n \log n$ Algorithm for Hyper-Minimizing a (Minimized) Deterministic Automaton*. Theor. Comput. Sci. 411(38–39), pp. 3404–3413, doi:10.1016/j.tcs.2010.05.029.

[20] John E. Hopcroft (1971): *An $n \log n$ Algorithm for Minimizing States in a Finite Automaton*. In: Theory of Machines and Computations, Academic Press, pp. 189–196.

[21] Artur Jeż & Andreas Maletti (2013): *Hyper-minimization for deterministic tree automata*. Int. J. Found. Comput. Sci. 24(6), pp. 815–830, doi:10.1142/S0129054113400200.

[22] Dexter Kozen (1992): *On the Myhill-Nerode theorem for trees*. Bulletin of the EATCS 47, pp. 170–173.

[23] Werner Kuich (1998): *Formal Power Series over Trees*. In: Proc. 3rd DLT, Aristotle University of Thessaloniki, pp. 61–101.

[24] Andreas Maletti & Daniel Quernheim (2011): *Hyper-minimisation of deterministic weighted finite automata over semifields*. In: Proc. 13th AFL, Nyíregyháza College, pp. 285–299.

[25] Andreas Maletti & Daniel Quernheim (2011): *Pushing for Weighted Tree Automata*. In: Proc. 36th MFCS, LNCS 6907, Springer, pp. 460–471, doi:10.1007/978-3-642-22993-0_42.

[26] Mehryar Mohri (1997): *Finite-State Transducers in Language and Speech Processing*. Comput. Linguist. 23(2), pp. 269–311.

[27] Slav Petrov, Leon Barrett, Romain Thibaux & Dan Klein (2006): *Learning Accurate, Compact, and Interpretable Tree Annotation*. In: Proc. 44th ACL, The Association for Computational Linguistics, pp. 433–440, doi:10.3115/1220175.1220230.

[28] Daniel Quernheim (2010): *Hyper-minimisation of weighted finite automata*. Master's thesis, Institut für Linguistik, Universität Potsdam.

[29] Robert Endre Tarjan (1972): *Depth-First Search and Linear Graph Algorithms*. SIAM J. Comput. 1(2), pp. 146–160, doi:10.1137/0201010.

[30] Antti Valmari & Petri Lehtinen (2008): *Efficient Minimization of DFAs with Partial Transition Functions*. In: Proc. 25th STACS, LIPIcs 1, Schloss Dagstuhl — Leibniz-Zentrum für Informatik, Germany, pp. 645–656, doi:10.4230/LIPIcs.STACS.2008.1328.

K-Position, Follow, Equation and K-C-Continuation Tree Automata Constructions

Ludovic Mignot
Laboratoire LITIS - EA 4108 Université
de Rouen, Avenue de l'Université 76801
Saint-Étienne-du-Rouvray Cedex, France
ludovic.mignot@univ-rouen.fr

Nadia Ouali Sebti
Laboratoire LITIS - EA 4108 Université
de Rouen, Avenue de l'Université 76801
Saint-Étienne-du-Rouvray Cedex, France
nadia.ouali-sebti@univ-rouen.fr

Djelloul Ziadi[*]
Laboratoire LITIS - EA 4108 Université
de Rouen, Avenue de l'Université 76801
Saint-Étienne-du-Rouvray Cedex, France
djelloul.ziadi@univ-rouen.fr

There exist several methods of computing an automaton recognizing the language denoted by a given regular expression: In the case of words, the *position automaton* $\mathscr{P}$ due to Glushkov, the *c-continuation automaton* $\mathscr{C}$ due to Champarnaud and Ziadi, the *follow automaton* $\mathscr{F}$ due to Ilie and Yu and the equation automaton $\mathscr{E}$ due to Antimirov. It has been shown that $\mathscr{P}$ and $\mathscr{C}$ are isomorphic and that $\mathscr{E}$ (resp. $\mathscr{F}$) is a quotient of $\mathscr{C}$ (resp. of $\mathscr{P}$).

In this paper, we define from a given regular tree expression the k-position tree automaton $\mathscr{P}$ and the follow tree automaton $\mathscr{F}$. Using the definition of the equation tree automaton $\mathscr{E}$ of Kuske and Meinecke and our previously defined k-C-continuation tree automaton $\mathscr{C}$, we show that the previous morphic relations are still valid on tree expressions.

1 Introduction

Regular expressions are used in numerous domains of applications in computer science. They are an easy and compact way to represent potentially infinite regular languages, that are well-studied objects leading to efficient decision problems. Among them, the membership test, that is to determine whether or not a given word belongs to a language. Given a regular expression E with n symbols and a word w, to determine whether w is in the language denoted by E can be polynomially performed (with respect to n) *via* the computation of a finite state machine, called an automaton, that can be seen as a symbol-labelled graph with initial and final states. There exist several methods to compute such an automaton.

The first approach is to determine particular properties over the syntactic structure of the regular expression E. Glushkov [8] proposed the computation of four position functions Null, First, Last, and Follow, which once computed, lead to the computation of a $(n+1)$- state automaton. Ilie and Yu showed in [9] how to reduce it by merging similar states. Another method is to compute the transition function of the automaton as follows: associating a regular expression with a state s, any path labelled by a word w brings the automaton from the state s into a finite set of states $S' = \{s'_1, \ldots, s'_k\}$ such that these states denote the quotient $w^{-1}(L(s))$ of the language $L(s)$ by w, that contains the word w' such that ww' belongs to $L(s)$. Basically, it is a computation that tries to determine what words w' can be accepted after reading a prefix w. The first author that introduced such a process is Brzozowski [2]. He showed how

[*]D. Ziadi was supported by the MESRS - Algeria under Project 8/U03/7015.

to compute a regular expression denoting $w^{-1}(L(E))$ from the expression E: this expression, denoted by $d_w(E)$, is called the *derivative* of E with respect to w. Furthermore, the set of dissimilar derivatives, combined with reduction according to associativity, commutativity and idempotence of the sum, is finite and can lead to the computation of a deterministic finite automaton. Antimirov [1] extended this method to the computation of partial derivatives, that are no longer expressions but sets of expressions. These so-called derived terms produce the *equation automaton*. Finally, by deriving expressions after having them indexed, Champarnaud and Ziadi [4] computed the *c-continuation automaton*.

The different morphic links between these four automata have been studied too: Ilie and Yu showed that the follow automaton is a quotient of the position automaton; Champarnaud and Ziadi proved that the position automaton and the c-continuation automaton are isomorphic and that the equation automaton is a quotient of the position automaton. Finally, using a join of the two previously defined quotients, Garcia *et al.* presented in [7] an automaton that is smaller than both the follow and the equation automata.

In this paper, we extend the study of these morphic links to different computations of tree automata. We define two new tree automata constructions, *the k-position automaton* and *the follow automaton*, and we study their morphic links with two other already known automata constructions, the *equation automaton* of Kuske and Meinecke [11] and our *k-C-continuation automaton* [14,15]. Notice that a position automaton and a reduced automaton have already been defined in [12]. However, they are not isomorphic with the automata we define in this paper. This study is motivated by the development of a library of functions for handling rational kernels [6] in the case of trees. The first problem consists in converting a regular tree expression into a tree transducer. Section 2 recalls basic definitions and properties of regular tree languages and regular tree expressions. In Section 3, we define two new automata computations, *the k-position automaton* and *the follow automaton* and recall the definition of the *equation automaton* and of the *k-C-continuation automaton*; we also present the morphic links between these four methods in this section. Section 4 is devoted to the comparison of the follow automaton and of the equation automaton; it is proved that there are no morphic link between them. Moreover, we extend the computation of the Garcia *et al.* equivalence leading to a smaller automaton in this section.

2 Preliminaries

Let (Σ, ar) be *a ranked alphabet*, where Σ is a finite set and ar represents the *rank* of Σ which is a mapping from Σ into $\mathbb{N}$. The set of symbols of rank n is denoted by Σ_n. The elements of rank 0 are called *constants*. A *tree t* over Σ is inductively defined as follows: $t = a$, $t = f(t_1, \ldots, t_k)$ where a is any symbol in Σ_0, k is any integer satisfying $k \geq 1$, f is any symbol in Σ_k and $t_1, \ldots, t_k$ are any k trees over Σ. We denote by T_Σ the set of trees over Σ. *A tree language* is a subset of T_Σ. Let $\Sigma_{\geq 1} = \Sigma \backslash \Sigma_0$ denote the set of *non-constant symbols* of the ranked alphabet Σ. *A Finite Tree Automaton* (FTA) [5,11] $\mathscr{A}$ is a tuple (Q, Σ, Q_T, Δ) where Q is a finite set of states, $Q_T \subset Q$ is the set of *final states* and $\Delta \subset \bigcup_{n \geq 0}(Q \times \Sigma_n \times Q^n)$ is the set of *transition rules*. This set is equivalent to the function Δ from $Q^n \times \Sigma_n$ to 2^Q defined by $(q, f, q_1, \ldots, q_n) \in \Delta \Leftrightarrow q \in \Delta(q_1, \ldots, q_n, f)$. The domain of this function can be extended to $(2^Q)^n \times \Sigma_n$ as follows: $\Delta(Q_1, \ldots, Q_n, f) = \bigcup_{(q_1, \ldots, q_n) \in Q_1 \times \cdots \times Q_n} \Delta(q_1, \ldots, q_n, f)$. Finally, we denote by Δ^* the function from $T_\Sigma \to 2^Q$ defined for any tree in T_Σ as follows: $\Delta^*(t) = \Delta(a)$ if $t = a$ with $a \in \Sigma_0$, $\Delta^*(t) = \Delta(\Delta^*(t_1), \ldots, \Delta^*(t_n), f)$ if $t = f(t_1, \ldots, t_n)$ with $f \in \Sigma_n$ and $t_1, \ldots, t_n \in T_\Sigma$. A tree is *accepted* by $\mathscr{A}$ if and only if $\Delta^*(t) \cap Q_T \neq \emptyset$. The language $\mathscr{L}(\mathscr{A})$ *recognized* by A is the set of trees accepted by $\mathscr{A}$ *i.e.* $\mathscr{L}(\mathscr{A}) = \{t \in T_\Sigma \mid \Delta^*(t) \cap Q_T \neq \emptyset\}$. Let $\sim$ be an equivalence relation over Q. We denote by $[q]$ the equivalence class of any state q in Q. The *quotient of* A w.r.t. $\sim$ is the tree automaton $A_{/\sim} = (Q_{/\sim}, \Sigma, Q_{T/\sim}, \Delta_{/\sim})$ where: $Q_{/\sim} = \{[q] \mid q \in Q\}$, $Q_{T/\sim} = \{[q] \mid q \in Q_T\}$, $\Delta_{/\sim} =$

$\{([q],f,[q_1],\ldots,[q_n]) \mid (q,f,q_1,\ldots,q_n) \in \Delta\}$. Notice that a transition $([q],f,[q_1],\ldots,[q_n])$ in $\Delta_{/\sim}$ does not imply a transition $(q,f,q_1,\ldots,q_n)$ in Δ. Moreover, the relation $\sim$ is not necessarily a congruence w.r.t. the transition function: in this paper, we will deal with specific equivalence relations (similarity relations) that turn to be congruences. This particular considerations will be clarified in Subsection 3.2.

For any integer $n \geq 0$, for any n languages $L_1,\ldots,L_n \subset T_\Sigma$, and for any symbol $f \in \Sigma_n$, $f(L_1,\ldots,L_n)$ is the tree language $\{f(t_1,\ldots,t_n) \mid t_i \in L_i\}$. The *tree substitution* of a constant c in Σ by a language $L \subset T_\Sigma$ in a tree $t \in T_\Sigma$, denoted by $t\{c \leftarrow L\}$, is the language inductively defined by: L if $t = c$; $\{d\}$ if $t = d$ where $d \in \Sigma_0 \setminus \{c\}$; $f(t_1\{c \leftarrow L\},\ldots,t_n\{c \leftarrow L\})$ if $t = f(t_1,\ldots,t_n)$ with $f \in \Sigma_n$ and $t_1,\ldots,t_n$ any n trees over Σ. Let c be a symbol in Σ_0. The *c-product* $L_1 \cdot_c L_2$ of two languages $L_1, L_2 \subset T_\Sigma$ is defined by $L_1 \cdot_c L_2 = \bigcup_{t \in L_1} \{t\{c \leftarrow L_2\}\}$. The *iterated c-product* is inductively defined for $L \subset T_\Sigma$ by: $L^{0_c} = \{c\}$ and $L^{(n+1)_c} = L^{n_c} \cup L \cdot_c L^{n_c}$. The *c-closure* of L is defined by $L^{*_c} = \bigcup_{n \geq 0} L^{n_c}$.

A *regular expression* over a ranked alphabet Σ is inductively defined by $\mathrm{E} = 0$, $\mathrm{E} \in \Sigma_0$, $\mathrm{E} = f(\mathrm{E}_1,\cdots,\mathrm{E}_n)$, $\mathrm{E} = (\mathrm{E}_1 + \mathrm{E}_2)$, $\mathrm{E} = (\mathrm{E}_1 \cdot_c \mathrm{E}_2)$, $\mathrm{E} = (\mathrm{E}_1{}^{*_c})$, where $c \in \Sigma_0$, $n \in \mathbb{N}$, $f \in \Sigma_n$ and $\mathrm{E}_1,\mathrm{E}_2,\ldots,\mathrm{E}_n$ are any n regular expressions over Σ. Parenthesis can be omitted when there is no ambiguity. We write $\mathrm{E}_1 = \mathrm{E}_2$ if E_1 and E_2 graphically coincide. We denote by $\mathrm{RegExp}(\Sigma)$ the set of all regular expressions over Σ. Every regular expression E can be seen as a tree over the ranked alphabet $\Sigma \cup \{+, \cdot_c, *_c \mid c \in \Sigma_0\}$ where $+$ and $\cdot_c$ can be seen as symbols of rank 2 and $*_c$ has rank 1. This tree is the syntax-tree T_E of E. The *alphabetical width* $||\mathrm{E}||$ of E is the number of occurrences of symbols of Σ in E. *The size* $|\mathrm{E}|$ of E is the size of its syntax tree T_E. The *language* $[\![\mathrm{E}]\!]$ *denoted by* E is inductively defined by $[\![0]\!] = \emptyset$, $[\![c]\!] = \{c\}$, $[\![f(\mathrm{E}_1,\mathrm{E}_2,\cdots,\mathrm{E}_n)]\!] = f([\![\mathrm{E}_1]\!],\ldots,[\![\mathrm{E}_n]\!])$, $[\![\mathrm{E}_1 + \mathrm{E}_2]\!] = [\![\mathrm{E}_1]\!] \cup [\![\mathrm{E}_2]\!]$, $[\![\mathrm{E}_1 \cdot_c \mathrm{E}_2]\!] = [\![\mathrm{E}_1]\!] \cdot_c [\![\mathrm{E}_2]\!]$, $[\![\mathrm{E}_1{}^{*_c}]\!] = [\![\mathrm{E}_1]\!]^{*_c}$ where $n \in \mathbb{N}$, $\mathrm{E}_1,\mathrm{E}_2,\ldots,\mathrm{E}_n$ are any n regular expressions, $f \in \Sigma_n$ and $c \in \Sigma_0$. It is well known that a tree language is accepted by some tree automaton if and only if it can be denoted by a regular expression [5, 11]. A regular expression E defined over Σ is *linear* if every symbol of rank greater than 1 appears at most once in E. Note that any constant symbol may occur more than once. Let E be a regular expression over Σ. The *linearized regular expression* $\overline{\mathrm{E}}$ in E of a regular expression E is obtained from E by marking differently all symbols of a rank greater than or equal to 1 (symbols of $\Sigma_{\geq 1}$). The marked symbols form together with the constants in Σ_0 a ranked alphabet $\mathrm{Pos}_E(E)$ the symbols of which we call *positions*. The mapping h is defined from $\mathrm{Pos}_\mathrm{E}(\mathrm{E})$ to Σ with $h(\mathrm{Pos}_\mathrm{E}(\mathrm{E})_m) \subset \Sigma_m$ for every $m \in \mathbb{N}$. It associates with a marked symbol $f_j \in \mathrm{Pos}_\mathrm{E}(\mathrm{E})_{\geq 1}$ the symbol $f \in \Sigma_{\geq 1}$ and for a symbol $c \in \Sigma_0$ the symbol $h(c) = c$. We can extend the mapping h naturally to $\mathrm{RegExp}(\mathrm{Pos}_\mathrm{E}(\mathrm{E})) \to \mathrm{RegExp}(\Sigma)$ by $h(a) = a$, $h(\mathrm{E}_1 + \mathrm{E}_2) = h(\mathrm{E}_1) + h(\mathrm{E}_2)$, $h(\mathrm{E}_1 \cdot_c \mathrm{E}_2) = h(\mathrm{E}_1) \cdot_c h(\mathrm{E}_2)$, $h(\mathrm{E}_1^{*_c}) = h(\mathrm{E}_1)^{*_c}$, $h(f_j(\mathrm{E}_1,\ldots,\mathrm{E}_n)) = f(h(\mathrm{E}_1),\ldots,h(\mathrm{E}_n))$, with $n \in \mathbb{N}$, $a \in \Sigma_0$, $f \in \Sigma_n$, $f_j \in \mathrm{Pos}_\mathrm{E}(\mathrm{E})_n$ such that $h(f_j) = f$ and $\mathrm{E}_1,\ldots,\mathrm{E}_n$ any regular expressions over $\mathrm{Pos}_\mathrm{E}(\mathrm{E})$.

3 Tree Automata from Regular Expressions

In this section, we show how to compute from a regular expression E four tree automata accepting $[\![\mathrm{E}]\!]$: we introduce two new constructions, the K-position automaton and the follow automaton of E, and then we recall two already-known constructions, the equation automaton [11] and the C-continuation automaton [14].

Regular languages defined over ranked alphabet Σ are exactly the languages denoted by a regular expression on Σ. There may exist many distinct regular expressions which denote the same regular language. Two regular expressions are said to be *equivalent* if they denote the same language. To simplify handling regular expressions, we define *trivial identities* for which regular expressions denote the same language. Let $\mathrm{E}_1 \ldots \mathrm{E}_n$ be n regular expressions over a ranked alphabet Σ and c be a symbol in

Σ_0. It can be trivially shown that:
$[\![\mathrm{E}_1+0]\!] = [\![0+\mathrm{E}_1]\!] = [\![\mathrm{E}_1]\!]$, $[\![\mathrm{E}_1\cdot_c 0]\!] = \mathrm{E}_{1c\leftarrow 0}$, $[\![[\![0\cdot_c \mathrm{E}_1]\!] = [\![0]\!]$, $[\![0^{*_c}]\!] = [\![c]\!]$, $[\![f(\mathrm{E}_1,\ldots,0,\ldots,\mathrm{E}_n)]\!] = [\![0]\!]$, where $\mathrm{E}_{c\leftarrow 0}$ is obtained by substituting the expression 0 to any symbol c in an expression E.
Consequently, we extend the equivalence $=$ as follows:

$$\mathrm{E}_1+0 = 0+\mathrm{E}_1 = \mathrm{E}_1,\ \mathrm{E}_1\cdot_c 0 = E_{1c\leftarrow 0},\ 0\cdot_c \mathrm{E}_1 = 0,\ 0^{*_c} = c,\ f(\mathrm{E}_1,\ldots,0,\ldots,\mathrm{E}_n) = 0.$$

It is easy to see that these equalities preserve the language. Consequently, any regular expression E denotes the same language as a regular expression E' with no occurrence of 0 in E' or $\mathrm{E}' = 0$.

In the following of this section, E is a regular expression over a ranked alphabet Σ. The set of symbols in Σ that appear in an expression F is denoted by Σ_F.

3.1 The *K*-Position Tree Automaton

In this section, we show how to compute the K-position tree automaton of a regular expression E, recognizing $[\![E]\!]$. This is an extension of the well-known position automaton [8] for word regular expressions where the K represents the fact that any k-ary symbol is no longer a state of the automaton, but is exploded into k states. The same method was presented independently by McNaughton and Yamada [13]. Its computation is based on the computations of particular *position functions*, defined in the following.

In what follows, for any two trees s and t, we denote by $s \preccurlyeq t$ the relation "s is a subtree of t". Let $t = f(t_1,\ldots,t_n)$ be a tree. We denote by $\mathrm{root}(t)$ the root of t, by k-child(t) the k^{th} child of f in t, that is the root of t_k if it exists, and by $\mathrm{Leaves}(t)$ the set of the leaves of t, *i.e.* $\{s \in \Sigma_0 \mid s \preccurlyeq t\}$.

Let E be linear, $1 \le k \le m$ be two integers and f be a symbol in Σ_m. The set First(E) is the subset of Σ defined by $\{\mathrm{root}(t) \in \Sigma \mid t \in [\![\mathrm{E}]\!]\}$; The set Follow(E, f,k) is the subset of Σ defined by $\{g \in \Sigma \mid \exists t \in [\![\mathrm{E}]\!], \exists s \preccurlyeq t, \mathrm{root}(s) = f, k\text{-child}(s) = g\}$; The set Last(E) is the subset of Σ_0 defined by $\mathrm{Last(E)} = \bigcup_{t\in[\![\mathrm{E}]\!]} \mathrm{Leaves}(t)$.

Example 1. *Let* $\Sigma = \Sigma_0 \cup \Sigma_1 \cup \Sigma_2$ *be defined by* $\Sigma_0 = \{a,b,c\}$, $\Sigma_1 = \{f,h\}$ *and* $\Sigma_2 = \{g\}$. *Let us consider the regular expression* E *and its linearized form defined by:*

$$\mathrm{E} = (f(a)^{*_a}\cdot_a b + h(b))^{*_b} + g(c,a)^{*_c}\cdot_c (f(a)^{*_a}\cdot_a b + h(b))^{*_b},$$
$$\overline{\mathrm{E}} = (f_1(a)^{*_a}\cdot_a b + h_2(b))^{*_b} + g_3(c,a)^{*_c}\cdot_c (f_4(a)^{*_a}\cdot_a b + h_5(b))^{*_b}.$$

The language denoted by $\overline{\mathrm{E}}$ *is* $[\![\overline{\mathrm{E}}]\!] = \{b, f_1(b), f_1(f_1(b)), f_1(h_2(b)), h_2(b), h_2(f_1(b)), h_2(h_2(b)),\ldots, g_3(b,a), g_3(g_3(b,a),a), g_3(f_4(b),a), g_3(h_5(b),a), f_4(f_4(b)), f_4(h_5(b), h_5(f_4(b)), h_5(h_5(b)),\ldots\}$.

Consequently, $\mathrm{First}(\overline{\mathrm{E}}) = \{b, f_1, h_2, g_3, f_4, h_5\}$ *and* $\mathrm{Follow}(\overline{\mathrm{E}}, f_1, 1) = \{b, f_1, h_2\}$, $\mathrm{Follow}(\overline{\mathrm{E}}, h_2, 1) = \{b, f_1, h_2\}$, $\mathrm{Follow}(\overline{\mathrm{E}}, g_3, 1) = \{b, g_3, f_4, h_5\}$, $\mathrm{Follow}(\overline{\mathrm{E}}, g_3, 2) = \{a\}$, $\mathrm{Follow}(\overline{\mathrm{E}}, f_4, 1) = \{b, f_4, h_5\}$, $\mathrm{Follow}(\overline{\mathrm{E}}, h_5, 1) = \{b, f_4, h_5\}$.

Let us first show that the position functions First and Follow are inductively computable.

Lemma 1. *Let* E *be linear. The set* First(E) *can be computed as follows:*

$$\mathrm{First}(0) = \emptyset,\ \mathrm{First}(a) = \{a\},\ \mathrm{First}(f(\mathrm{E}_1,\cdots,\mathrm{E}_m)) = \{f\},$$
$$\mathrm{First}(\mathrm{E}_1+\mathrm{E}_2) = \mathrm{First}(\mathrm{E}_1)\cup\mathrm{First}(\mathrm{E}_2),\ \mathrm{First}({\mathrm{E}_1}^{*_c}) = \mathrm{First}(\mathrm{E}_1)\cup\{c\},$$
$$\mathrm{First}(\mathrm{E}_1\cdot_c \mathrm{E}_2) = \begin{cases} (\mathrm{First}(\mathrm{E}_1)\setminus\{c\})\cup\mathrm{First}(\mathrm{E}_2) & \text{if } c \in [\![\mathrm{E}_1]\!], \\ \mathrm{First}(\mathrm{E}_1) & \text{otherwise.} \end{cases}$$

Lemma 2. *Let* E *be linear,* $1 \le k \le m$ *be two integers and* f *be a symbol in* Σ_m. *The set of symbols* Follow(E, f,k) *can be computed inductively as follows:*

$$\mathrm{Follow}(0,f,k) = \mathrm{Follow}(a,f,k) = \emptyset,$$
$$\mathrm{Follow}(g(\mathrm{E}_1,\ldots,\mathrm{E}_n),f,k) = \begin{cases} \mathrm{First}(\mathrm{E}_k) & \text{if } f = g, \\ \mathrm{Follow}(\mathrm{E}_l,f,k) & \text{if } \exists l \mid f \in \Sigma_{E_l}, \\ \emptyset & \text{otherwise .} \end{cases}$$

$$\mathrm{Follow}(\mathrm{E}_1+\mathrm{E}_2,f,k)=\begin{cases}\mathrm{Follow}(\mathrm{E}_1,f,k) & \textit{if } f\in\Sigma_{\mathrm{E}_1},\\ \mathrm{Follow}(\mathrm{E}_2,f,k) & \textit{if } f\in\Sigma_{\mathrm{E}_2},\\ \emptyset & \textit{otherwise}\ .\end{cases}$$

$$\mathrm{Follow}(\mathrm{E}_1\cdot_c\mathrm{E}_2,f,k)=\begin{cases}(\mathrm{Follow}(\mathrm{E}_1,f,k)\setminus\{c\})\cup\mathrm{First}(\mathrm{E}_2) & \textit{if } c\in\mathrm{Follow}(\mathrm{E}_1,f,k),\\ \mathrm{Follow}(\mathrm{E}_1,f,k) & \textit{if } f\in\Sigma_{\mathrm{E}_1}\wedge c\notin\mathrm{Follow}(\mathrm{E}_1,f,k),\\ \mathrm{Follow}(\mathrm{E}_2,f,k) & \textit{if } f\in\Sigma_{\mathrm{E}_2}\wedge c\in\mathrm{Last}(\mathrm{E}_1),\\ \emptyset & \textit{otherwise,}\end{cases}$$

$$\mathrm{Follow}(\mathrm{E}_1^{*_c},f,k)=\begin{cases}\mathrm{Follow}(\mathrm{E}_1,f,k)\cup\mathrm{First}(\mathrm{E}_1) & \textit{if } c\in\mathrm{Follow}(\mathrm{E}_1,f,k),\\ \mathrm{Follow}(\mathrm{E}_1,f,k) & \textit{otherwise,}\end{cases}$$

The two functions First and Follow are sufficient to compute the *K-position tree automaton* of *E*.

Definition 1. *Let* E *be linear. The K*-position automaton $\mathscr{P}_{\mathrm{E}}$ *is the automaton* (Q,Σ,Q_T,Δ) *defined by*

$$Q=\{f^k\mid f\in\Sigma_m\wedge 1\le k\le m\}\cup\{\varepsilon^1\}\ \textit{with } \varepsilon^1 \textit{ a new symbol not in } \Sigma,\ Q_T=\{\varepsilon^1\},$$
$$\begin{aligned}\Delta=\ &\{(f^k,g,g^1,\ldots,g^n)\mid f\in\Sigma_m\wedge k\le m\wedge g\in\Sigma_n\wedge g\in\mathrm{Follow}(\mathrm{E},f,k)\}\\ &\cup\{(\varepsilon^1,f,f^1,\ldots,f^m)\mid f\in\Sigma_m\wedge f\in\mathrm{First}(\mathrm{E})\}\\ &\cup\{(\varepsilon^1,c)\mid c\in\Sigma_0\wedge c\in\mathrm{First}(\mathrm{E})\}\\ &\cup\{(f^k,c)\mid f\in\Sigma_m\wedge k\le m\wedge c\in\mathrm{Follow}(\mathrm{E},f,k)\}\end{aligned}$$

In order to show that the *K*-position tree automaton of E accepts $[\![\mathrm{E}]\!]$, we characterize the membership of a tree *t* in the language denoted by E using the functions First and Follow.

Proposition 1. *Let* E *be linear. A tree t belongs to* $[\![\mathrm{E}]\!]$ *if and only if:*

1. $\mathrm{root}(t)\in\mathrm{First}(\mathrm{E})$ *and*
2. *for every subtree* $f(t_1,\ldots,t_m)$ *of t, for any integer k in* $\{1,\ldots,m\}$, $\mathrm{root}(t_k)\in\mathrm{Follow}(\mathrm{E},f,k)$.

Let us show how to link the characterization in Proposition 1 with the transition sequences in $\mathscr{P}_{\mathrm{E}}$.

Proposition 2. *Let* E *be linear and* $\mathscr{P}_{\mathrm{E}}=(Q,\Sigma,Q_T,\Delta)$. *Let* $t=f(t_1,\ldots,t_m)$ *be a term in* T_Σ. *Then the two following propositions are equivalent:*

1. $\forall g(s_1,\ldots,s_l)\preccurlyeq t$, $\forall p\le l$, $\mathrm{root}(s_p)\in\mathrm{Follow}(\mathrm{E},g,p)$,
2. $\forall 1\le k\le m$, $f^k\in\Delta^*(t_k)$.

As a direct consequence of the two previous propositions, it can be shown that the *K*-position automaton of E recognizes the language denoted by *E*.

Theorem 1. *If* E *is linear, then* $\mathscr{L}(\mathscr{P}_{\mathrm{E}})=[\![\mathrm{E}]\!]$.

This construction can be extended to expressions that are not necessarily linear using the linearization and the mapping *h*. The *K-Position Automaton* $\mathscr{P}_{\mathrm{E}}$ associated with E is obtained by replacing each transition $(f_j^k,g_i,g_i^1,\ldots,g_i^n)$ of the tree automaton $\mathscr{P}_{\overline{\mathrm{E}}}$ by $(f_j^k,h(g_i),g_i^1,\ldots,g_i^n)$.

Corollary 1. $h([\![\overline{\mathrm{E}}]\!])=h(\mathscr{L}(\mathscr{P}_{\overline{\mathrm{E}}}))=\mathscr{L}(\mathscr{P}_{\mathrm{E}})=[\![\mathrm{E}]\!]$.

Example 2. *Let* $\mathrm{E}=(f(a)^{*_a}\cdot_a b+h(b))^{*_b}+g(c,a)^{*_c}\cdot_c(f(a)^{*_a}\cdot_a b+h(b))^{*_b}$ *be the regular expression of Example 1. The k-Position Automaton* $\mathscr{P}_{\overline{\mathrm{E}}}$ *associated with* $\overline{\mathrm{E}}$ *is given in Figure 1. The set of states is* $Q=\{\varepsilon^1,f_1^1,h_2^1,g_3^1,g_3^2,f_4^1,h_5^1\}$. *The set of final states is* $Q_T=\{\varepsilon^1\}$. *The set of transition rules* Δ *is*

$$\begin{array}{lllllll} f_1(f_1^1)\to f_1^1 & f_1(f_1^1)\to\varepsilon^1 & f_1(h_2^1)\to\varepsilon^1 & f_1(h_2^1)\to f_1^1 & h_2(f_1^1)\to\varepsilon^1 & b\to f_1^1 & b\to h_2^1\\ h_2(f_1^1)\to h_2^1 & h_2(h_2^1)\to\varepsilon^1 & h_2(h_2^1)\to h_2^1 & g_3(f_4^1,g_3^2)\to\varepsilon^1 & g_3(h_5^1,g_3^2)\to\varepsilon^1 & b\to g_3^1 & a\to g_3^2\\ f_4(f_4^1)\to\varepsilon^1 & f_4(f_4^1)\to f_4^1 & f_4(h_5^1)\to\varepsilon^1 & f_4(h_5^1)\to f_4^1 & h_5(f_4^1)\to\varepsilon^1 & b\to h_5^1 & b\to f_4^1\\ b\to\varepsilon^1 & h_5(f_4^1)\to h_5^1 & h_5(h_5^1)\to h_5^1 & g_3(g_3^1,g_3^2)\to\varepsilon^1 & h_5(h_5^1)\to\varepsilon^1 & & \end{array}$$

The number of states is $|Q|=7$ *and the number of transition rules is* $|\Delta|=26$.

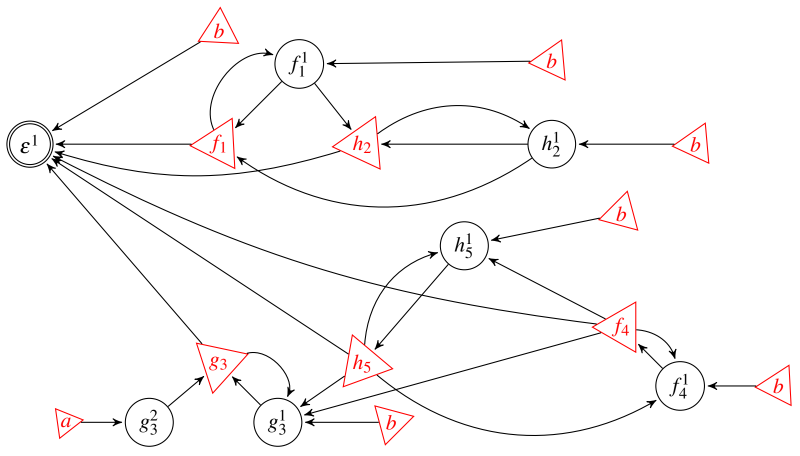


Figure 1: The k-Position Automaton $\mathscr{P}_{\overline{\mathrm{E}}}$.

3.2 The Follow Tree Automaton

In this section, we define the follow tree automaton which is a generalisation of the Follow automaton introduced by L. Ilie and S. Yu in [9] in the case of words, and that it is a quotient of the K-position automaton, similarly to the case of words. Notice that in this automaton, states are no longer positions, but sets of positions.

Definition 2. *Let* E *be linear. The* Follow Automaton *of* E *is the tree automaton* $\mathscr{F}_\mathrm{E} = (Q, \Sigma, Q_T, \Delta)$ *defined as follows*

$$Q = \{\mathrm{First}(\mathrm{E})\} \cup \bigcup_{f \in \Sigma_{E_m}} \{\mathrm{Follow}(\mathrm{E}, f, k) \mid 1 \le k \le m\},\ Q_T = \{\mathrm{First}(\mathrm{E})\},$$

$$\begin{aligned}\Delta = \quad & \{(\mathrm{Follow}(\mathrm{E}, g, l), f, \mathrm{Follow}(\mathrm{E}, f, 1), \ldots, \mathrm{Follow}(\mathrm{E}, f, m) \mid f \in \Sigma_{E\,m} \wedge f \in \mathrm{Follow}(\mathrm{E}, g, l) \wedge \\ & g \in \Sigma_n \wedge l \le n\} \\ & \cup \{(I, c) \mid c \in I \wedge c \in \Sigma_0\}\end{aligned}$$

Let us show that $\mathscr{F}_\mathrm{E}$ is a quotient of $\mathscr{P}_\mathrm{E}$ w.r.t. a similarity relation ; since this kind of quotient preserves the language, this method is consequently a proof of the fact that the language denoted by E is recognized by $\mathscr{F}_\mathrm{E}$.

A *similarity relation* over an automaton $A = (Q, \Sigma, Q_T, \Delta)$ is an equivalence relation $\sim$ over Q such that for any two states q and q' in Q: $q \sim q' \Rightarrow \forall f \in \Sigma_n$, $\forall (q_1, \ldots, q_n) \in Q^n$, $(q, f, q_1, \ldots, q_n) \in \Delta \Leftrightarrow (q', f, q_1, \ldots, q_n) \in \Delta$. In other words, two similar states admit the same predecessors w.r.t. any symbol.

Proposition 3. *Let* $\mathscr{A}$ *be an automaton and* $\sim$ *be a similarity relation over* $\mathscr{A}$. *Then* $\mathscr{L}(\mathscr{A}_{/\sim}) = \mathscr{L}(\mathscr{A})$.

The quotient from $\mathscr{P}_E$ to $\mathscr{F}_E$ is defined by the following similarity relation. Notice that we extend the definition of the function Follow to the position ε^1 by $\mathrm{Follow}(E, \varepsilon^1, 1) = \mathrm{First}(E)$. Let E be linear and $\mathscr{P}_\mathrm{E} = (Q, \Sigma, Q_T, \Delta)$. The *Follow Relation* is the relation $\sim_\mathscr{F}$ defined for any two states f^k and g^l in Q by $f^k \sim_\mathscr{F} g^l \Leftrightarrow \mathrm{Follow}(\mathrm{E}, f, k) = \mathrm{Follow}(\mathrm{E}, g, l)$.

Proposition 4. *Let E be linear. The relation* $\sim_\mathscr{F}$ *is the largest similarity relation over* $\mathscr{P}_E$.

Proposition 5. *Let* E *be linear. The finite tree automaton* $\mathscr{P}_\mathrm{E} / \sim_\mathscr{F}$ *is isomorphic to* $\mathscr{F}_\mathrm{E}$.

As a direct consequence of the previous results, the following theorem can be shown.

Theorem 2. *Let* E *be linear. Then* $\mathscr{L}(\mathscr{F}_\mathrm{E}) = [\![\mathrm{E}]\!]$.

Finally, this method can be extended to expressions that are not necessarily linear as follows. The *Follow Automaton* $\mathscr{F}_{\mathrm{E}}$ associated with E is obtained by replacing each transition $(I, f_j, \mathrm{Follow}(\mathrm{E}, f_j, 1), \ldots, \mathrm{Follow}(\mathrm{E}, f_j, m))$ of $\mathscr{F}_{\overline{\mathrm{E}}}$ by $(I, h(f_j), \mathrm{Follow}(\mathrm{E}, f_j, 1), \ldots, \mathrm{Follow}(\mathrm{E}, f_j, m))$.

Corollary 2. $\mathscr{L}(\mathscr{F}_{\mathrm{E}}) = [\![\mathrm{E}]\!]$.

Example 3. *The Follow Automaton $\mathscr{F}_{\mathrm{E}}$ associated with $\mathrm{E} = (f(a)^{*_a} \cdot_a b + h(b))^{*_b} + g(c,a)^{*_c} \cdot_c (f(a)^{*_a} \cdot_a b + h(b))^{*_b}$ of Example 1 is given in Figure 2.*

The set of states is $Q = \{\{a\}, \{b, f_1, h_2\}, \{b, f_1, h_2, g_3, f_4, h_5\}, \{b, g_3, f_4, h_5\}, \{b, f_4, h_5\}\}$ and $Q_T = \{\{b, f_1, h_2, g_3, f_4, h_5\}\}$. The set of transition rules Δ is

$f(\{b,f_1,h_2\}) \to \{b,f_1,h_2\}$	$h(\{b,f_1,h_2\}) \to \{b,f_1,h_2,g_3,f_4,h_5\}$	$b \to \{b,f_1,h_2,g_3,f_4,h_5\}$
$h(\{b,f_1,h_2\}) \to \{b,f_1,h_2\}$	$f(\{b,f_4,h_5\}) \to \{b,f_4,h_5\}$	$b \to \{b,g_3,f_4,h_5\}$
$f(\{b,f_4,h_5\}) \to \{b,g_3,f_4,h_5\}$	$f(\{b,f_4,h_5\}) \to \{b,f_1,h_2,g_3,f_4,h_5\}$	$a \to \{a\}$
$h(\{b,f_4,h_5\}) \to \{b,f_1,h_2,g_3,f_4,h_5\}$	$h(\{b,f_4,h_5\}) \to \{b,f_4,h_5\}$	$b \to \{b,f_1,h_2\}$
$h(\{b,f_4,h_5\}) \to \{b,g_3,f_4,h_5\}$	$f(\{b,f_1,h_2\}) \to \{b,f_1,h_2,g_3,f_4,h_5\}$	$b \to \{b,f_4,h_5\}$
$g(\{b,g_3,f_4,h_5\},\{a\}) \to \{b,g_3,f_4,h_5\}$		$g(\{b,g_3,f_4,h_5\},\{a\}) \to \{b,f_1,h_2,g_3,f_4,h_5\}$

The number of states is $|Q| = 5$ and the number of transition rules is $|\Delta| = 17$.

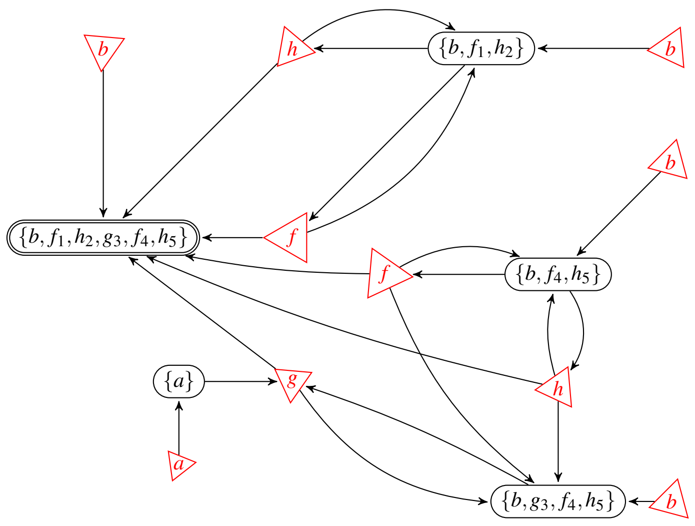


Figure 2: The Follow Automaton $\mathscr{F}_{\mathrm{E}}$.

3.3 The Equation Tree Automaton

In [11], Kuske and Meinecke extend the notion of word partial derivatives [1] to tree partial derivatives in order to compute from E a tree automaton recognizing $[\![\mathrm{E}]\!]$. Due to the notion of ranked alphabet, partial derivatives are no longer sets of expressions, but sets of tuples of expressions.

Let $\mathscr{N} = (\mathrm{E}_1, \ldots, \mathrm{E}_n)$ be a tuple of regular expressions, F and G be some regular expressions and $c \in \Sigma_0$. Then $\mathscr{N} \cdot_c \mathrm{F}$ is the tuple $(\mathrm{E}_1 \cdot_c \mathrm{F}, \ldots, \mathrm{E}_n \cdot_c \mathrm{F})$. For a set $\mathscr{S}$ of tuples of regular expressions, $\mathscr{S} \cdot_c \mathrm{F}$ is the set $\mathscr{S} \cdot_c \mathrm{F} = \{\mathscr{N} \cdot_c \mathrm{F} \mid \mathscr{N} \in \mathscr{S}\}$. Finally, $\mathrm{SET}(\mathscr{N}) = \{\mathrm{E}_1, \cdots, \mathrm{E}_m\}$ and $\mathrm{SET}(\mathscr{S}) = \bigcup_{\mathscr{N} \in \mathscr{S}} \mathrm{SET}(\mathscr{N})$.

Let f be a symbol in $\Sigma_{>0}$. The set $f^{-1}(\mathrm{E})$ of tuples of regular expressions is defined as follows:

$$f^{-1}(0) = \emptyset, \qquad f^{-1}(F+G) = f^{-1}(\mathrm{F}) \cup f^{-1}(\mathrm{G}), \qquad f^{-1}(\mathrm{F}^{*_c}) = f^{-1}(\mathrm{F}) \cdot_c \mathrm{F}^{*_c},$$

$$f^{-1}(g(\mathrm{E}_1, \cdots, \mathrm{E}_n)) = \begin{cases} \{(\mathrm{E}_1, \cdots, \mathrm{E}_n)\} & \text{if } f = g, \\ \emptyset & \text{otherwise,} \end{cases} \quad f^{-1}(\mathrm{F} \cdot_c \mathrm{G}) = \begin{cases} f^{-1}(\mathrm{F}) \cdot_c \mathrm{G} & \text{if } c \notin [\![\mathrm{F}]\!] \\ f^{-1}(\mathrm{F}) \cdot_c \mathrm{G} \cup f^{-1}(\mathrm{G}) & \text{otherwise.} \end{cases}$$

The function f^{-1} is extended to any set S of regular expressions by $f^{-1}(S)=\bigcup_{\mathrm{E}\in S} f^{-1}(\mathrm{E})$.
The *partial derivative* of E w.r.t. a word $w\in\Sigma_{\geq 1}^*$, denoted by $\partial_w(\mathrm{E})$, is the set of regular expressions inductively defined by:

$$\partial_w(\mathrm{E})=\begin{cases}\{\mathrm{E}\} & \text{if } w=\varepsilon,\\ \mathrm{SET}(f^{-1}(\partial_u(\mathrm{E}))) & \text{if } w=uf, f\in\Sigma_{\geq 1}, u\in\Sigma_{\geq 1}^*, f^{-1}(\partial_u(\mathrm{E}))\neq\emptyset,\\ \{0\} & \text{if } w=uf, f\in\Sigma_{\geq 1}, u\in\Sigma_{\geq 1}^*, f^{-1}(\partial_u(\mathrm{E}))=\emptyset.\end{cases}$$

The *Equation Automaton* of E is the tree automaton $\mathscr{A}_\mathrm{E}=(Q,\Sigma,Q_T,\Delta)$ defined by $Q=\{\partial_w(\mathrm{E})\mid w\in\Sigma_{\geq 1}^*\}$, $Q_T=\{\mathrm{E}\}$, and

$$\begin{aligned}\Delta = \quad & \{(\mathrm{F},f,\mathrm{G}_1,\ldots,\mathrm{G}_m)\mid \mathrm{F}\in Q, f\in\Sigma_m, m\geq 1,(\mathrm{G}_1,\ldots,\mathrm{G}_m)\in f^{-1}(\mathrm{F})\}\\ \cup\, & \{(\mathrm{F},c)\mid\ \mathrm{F}\in Q\wedge c\in([\![\mathrm{F}]\!]\cap\Sigma_0)\}\end{aligned}$$

Example 4. *Let* $\mathrm{E}=\underbrace{(f(a)^{*_a}\cdot_a b+h(b))^{*_b}}_{\mathrm{F}}+\underbrace{g(c,a)^{*_c}}_{\mathrm{G}}\cdot_c\underbrace{(f(a)^{*_a}\cdot_a b+h(b))^{*_b}}_{\mathrm{F}}$ *(Example 1).*

$$\begin{array}{lll}\partial_h(\mathrm{E})=\{b\cdot_b\mathrm{F}\}, & \partial_f(\mathrm{E})=\{((a\cdot_a f(a)^{*_a})\cdot_a b)\cdot_b\mathrm{F}\} & \partial_{ff}(\mathrm{E})=\{((a\cdot_a f(a)^{*_a})\cdot_a b)\cdot_b\mathrm{F}\},\\ \partial_{fh}(\mathrm{E})=\{b\cdot_b\mathrm{F}\}, & \partial_g(\mathrm{E})=\{(a\cdot_c\mathrm{G})\cdot_c\mathrm{F},\ (c\cdot_c\mathrm{G})\cdot_c\mathrm{F}\} & \partial_{hf}(\mathrm{E})=\{((a\cdot_a f(a)^{*_a})\cdot_a b)\cdot_b\mathrm{F}\}\\ \partial_{gh}(\mathrm{E})=\{b\cdot_b\mathrm{F}\} & \partial_{hh}(\mathrm{E})=\{((a\cdot_a f(a)^{*_a})\cdot_a b)\cdot_b\mathrm{F}\}, & \partial_{gf}(\mathrm{E})=\{((a\cdot_a f(a)^{*_a})\cdot_a b)\cdot_b\mathrm{F}\}\end{array}$$

$$\partial_{gg}(\mathrm{E})=\{(a\cdot_c\mathrm{G})\cdot_c\mathrm{F},\ (c\cdot_c\mathrm{G})\cdot_c\mathrm{F}\},$$

The set of states Q is $q_0=\mathrm{E}$, $q_1=((a\cdot_a f(a)^{*_a})\cdot_a b)\cdot_b\mathrm{F}$, $q_2=b\cdot_b\mathrm{F}$, $q_3=(c\cdot_c\mathrm{G})\cdot_c\mathrm{F}$, $q_4=(a\cdot_c\mathrm{G})\cdot_c\mathrm{F}$. The set of final states is $Q_T=\{q_0\}$. The set of transition rules is

$$\begin{array}{lllll}b\to q_0 & b\to q_1 & b\to q_3 & b\to q_2 & f(q_1)\to q_0\\ h(q_2)\to q_0 & g(q_3,q_4)\to q_0 & h(q_2)\to q_1 & g(q_3,q_4)\to q_4 & f(q_1)\to q_1\\ h(q_2)\to q_2 & f(q_1)\to q_2 & f(q_1)\to q_4 & h(q_2)\to q_4 & a\to q_4\end{array}$$

The number of states is $|Q|=5$ and the number of transition rules is $|\Delta|=15$. The Equation Automaton associated with E *is given in Figure 3.*

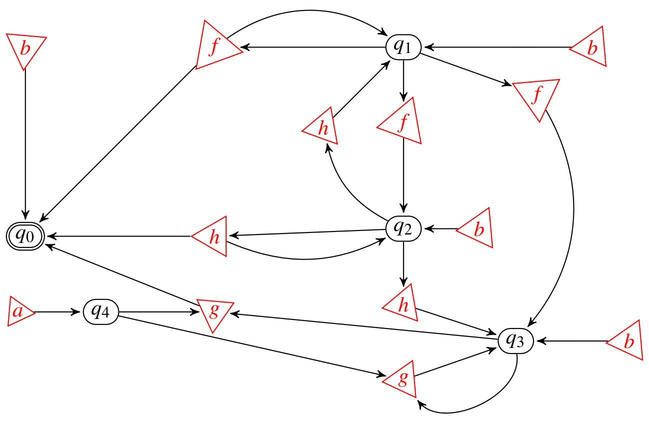


Figure 3: The Equation Automaton $\mathscr{A}_\mathrm{E}$.

3.4 The k-C-Continuation Tree Automaton

In [11], Kuske and Meinecke show how to efficiently compute the equation tree automaton of a regular expression *via* an extension of Champarnaud and Ziadi's C-Continuation [3, 4, 10]. In [14, 15], we show how to inductively compute them. We also show how to efficiently compute the k-C-Continuation tree automaton associated with a regular expression. In this section, we prove that this automaton is isomorphic to the k-position tree automaton, similarly to the case of words.

Definition 3 ([14, 15]). *Let* $\mathrm{E} \neq 0$ *be linear. Let* k *and* m *be two integers such that* $1 \leq k \leq m$*. Let* f *be in* $(\Sigma_{\mathrm{E}} \cap \Sigma_m)$*. The* k-C-continuation $C_{f^k}(\mathrm{E})$ *of* f *in* E *is the regular expression defined by:*

$$C_{f^k}(g(\mathrm{E}_1,\cdots,\mathrm{E}_m)) = \begin{cases} \mathrm{E}_k & \text{if } f = g \\ C_{f^k}(\mathrm{E}_j) & \text{if } f \in \Sigma_{\mathrm{E}_j} \end{cases}$$
$$C_{f^k}(\mathrm{E}_1 + \mathrm{E}_2) = \begin{cases} C_{f^k}(E_1) & \text{if } f \in \Sigma_{E_1} \\ C_{f^k}(E_2) & \text{if } f \in \Sigma_{E_2} \end{cases}$$
$$C_{f^k}(E_1 \cdot_c E_2) = \begin{cases} C_{f^k}(E_1) \cdot_c \mathrm{G} & \text{if } f \in \Sigma_{E_1} \\ C_{f^k}(E_2) & \text{if } f \in \Sigma_{E_2} \\ & \text{and } c \in \mathrm{Last}(E_1) \\ 0 & \text{otherwise} \end{cases}$$
$$C_{f^k}(\mathrm{F}^{*_c}) = C_{f^k}(\mathrm{F}) \cdot_c \mathrm{F}^{*_c}$$

By convention, we set $C_{\varepsilon^1}(\mathrm{E}) = \mathrm{E}$.

Let us now show how to compute the k-C-Continuation tree automaton.

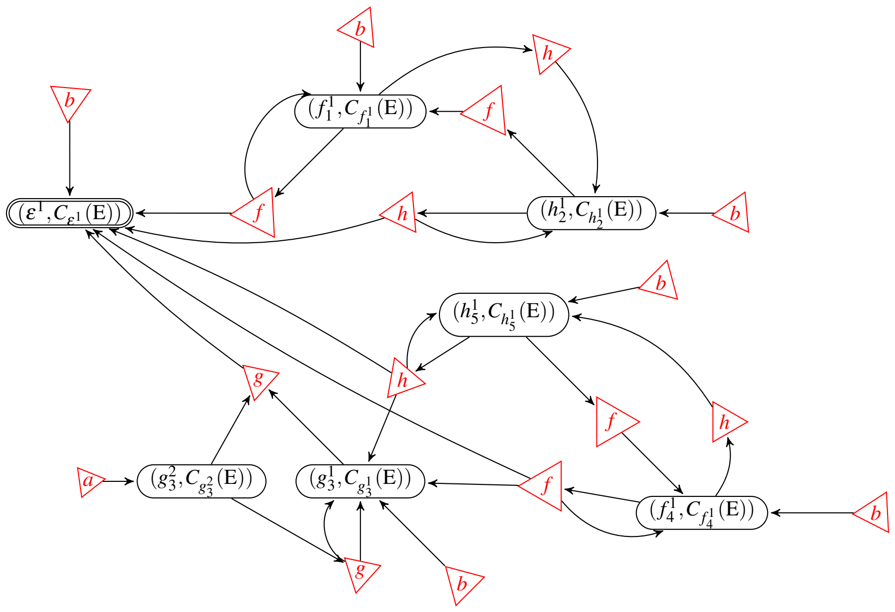
Definition 4 ([14, 15]). *Let* $\mathrm{E} \neq 0$ *be linear. The automaton* $\mathscr{C}_{\mathrm{E}} = (Q_{\mathscr{C}}, \Sigma_E, \{C_{\varepsilon^1}(\mathrm{E})\}, \Delta_{\mathscr{C}})$ *is defined by*

- $Q_{\mathscr{C}} = \{(f^k, C_{f^k}(\mathrm{E})) \mid f \in \Sigma_m, 1 \leq k \leq m\} \cup \{(\varepsilon^1, C_{\varepsilon^1}(\mathrm{E}))\}$,
- $$\begin{aligned} \Delta_{\mathscr{C}} = \quad & \{((x, C_x(\mathrm{E})), g, ((g^1, C_{g^1}(\mathrm{E})), \ldots, (g^m, C_{g^m}(\mathrm{E})))) \mid g \in \Sigma_{E\,m}, \\ & m \geq 1, (C_{g^1}(\mathrm{E}), \ldots, C_{g^m}(\mathrm{E})) \in g^{-1}(C_x(\mathrm{E}))\} \\ & \cup \{((x, C_x(\mathrm{E})), c) \mid, c \in [\![C_x(\mathrm{E})]\!] \cap \Sigma_0\} \end{aligned}$$

The *C-Continuation tree automaton* $\mathscr{C}_{\mathrm{E}}$ associated with E is obtained by relabelling the transitions of $\mathscr{C}_{\overline{\mathrm{E}}}$ using the mapping h.

Theorem 3 ([14, 15]). *The automaton* $\mathscr{C}_{\mathrm{E}}$ *accepts* $[\![\mathrm{E}]\!]$.

Example 5. *Let* $\mathrm{E} = (f(a)^{*_a} \cdot_a b + h(b))^{*_b} + g(c,a)^{*_c} \cdot_c (f(a)^{*_a} \cdot_a b + h(b))^{*_b}$ *defined in Example 1 and* $\overline{\mathrm{E}} = \underbrace{(f_1(a)^{*_a} \cdot_a b + h_2(b))^{*_b}}_{\mathrm{F}_1} + \underbrace{g_3(c,a)^{*_c}}_{\mathrm{G}_2} \cdot_c \underbrace{(f_4(a)^{*_a} \cdot_a b + h_5(b))^{*_b}}_{\mathrm{F}_3}$.



b
h
$(f_1^1, C_{f_1^1}(\mathrm{E}))$
f
b
$(\varepsilon^1, C_{\varepsilon^1}(\mathrm{E}))$
f
h
$(h_2^1, C_{h_2^1}(\mathrm{E}))$
b
b
$(h_5^1, C_{h_5^1}(\mathrm{E}))$
g
h
f
h
a
$(g_3^2, C_{g_3^2}(\mathrm{E}))$
$(g_3^1, C_{g_3^1}(\mathrm{E}))$
f
$(f_4^1, C_{f_4^1}(\mathrm{E}))$
b
g
b

Figure 4: The k-C-Continuation Automaton $\mathscr{C}_{\mathrm{E}}$.

The computation of the k-C-Continuations of E *using the Definition 3 is given in Table 1.*

$C_{f_1^1}(\overline{\mathrm{E}}) = ((a \cdot_a f_1(a)^{*_a}) \cdot_a b) \cdot_b \mathrm{F}_1$	$h(C_{f_1^1}(\overline{\mathrm{E}})) = ((a \cdot_a f(a)^{*_a}) \cdot_a b) \cdot_b \mathrm{F},$
$C_{h_2^1}(\overline{\mathrm{E}}) = b \cdot_b \mathrm{F}_1$	$h(C_{h_2^1}(\overline{\mathrm{E}})) = b \cdot_b \mathrm{F},$
$C_{g_3^1}(\overline{\mathrm{E}}) = (c \cdot_c g_3(c,a)^{*_c}) \cdot_c \mathrm{F}_3$	$h(C_{g_3^1}(\overline{\mathrm{E}})) = (c \cdot_c g(c,a)^{*_c}) \cdot_c \mathrm{F},$
$C_{g_3^2}(\overline{\mathrm{E}}) = (a \cdot_c g_3(c,a))^{*_c}) \cdot_c \mathrm{F}_3$	$h(C_{g_3^2}(\overline{\mathrm{E}})) = (a \cdot_c g(c,a))^{*_c}) \cdot_c \mathrm{F},$
$C_{f_4^1}(\overline{\mathrm{E}}) = ((a \cdot_a f_4(a)^{*_a}) \cdot_a b) \cdot_b \mathrm{F}_3$	$h(C_{f_4^1}(\overline{\mathrm{E}})) = ((a \cdot_a f(a)^{*_a}) \cdot_a b) \cdot_b \mathrm{F},$
$C_{h_5^1}(\overline{\mathrm{E}}) = b \cdot_b \mathrm{F}_3$	$h(C_{h_5^1}(\overline{\mathrm{E}})) = b \cdot_b \mathrm{F}.$

Table 1: The k-C-Continuations of $\overline{\mathrm{E}}$.

The set of states of the automaton $\mathscr{C}_{\mathrm{E}}$ *is* $Q = \{(\varepsilon^1, C_{\varepsilon^1}(\overline{\mathrm{E}})), (f_1^1, C_{f_1^1}(\overline{\mathrm{E}})), (h_2^1, C_{h_2^1}(\overline{\mathrm{E}})), (g_3^1, C_{g_3^1}(\overline{\mathrm{E}})), (g_3^2, C_{g_3^2}(\overline{\mathrm{E}})), (f_4^1, C_{f_4^1}(\overline{\mathrm{E}})), (h_5^1, C_{h_5^1}(\overline{\mathrm{E}}))\}$.

The set of transition rules Δ *is*

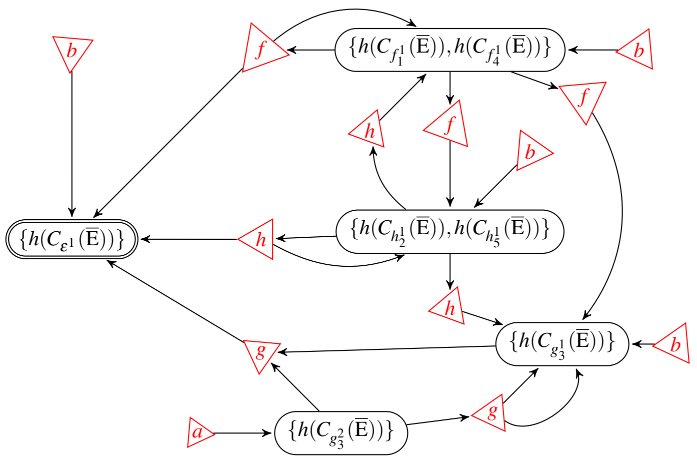
$f((f_1^1, C_{f_1^1}(\overline{\mathrm{E}}))) \to (\varepsilon^1, C_{\varepsilon^1}(\mathrm{E}))$	$f((f_4^1, C_{f_4^1}(\overline{\mathrm{E}}))) \to (\varepsilon^1, C_{\varepsilon^1}(\mathrm{E}))$	$b \to (f_1^1, C_{f_1^1}(\overline{\mathrm{E}}))$
$g((g_3^1, C_{g_3^1}(\overline{\mathrm{E}})), (g_3^2, C_{g_3^2}(\overline{\mathrm{E}}))) \to (\varepsilon^1, C_{\varepsilon^1}(\mathrm{E}))$	$h((h_5^1, C_{h_5^1}(\overline{\mathrm{E}}))) \to (\varepsilon^1, C_{\varepsilon^1}(\mathrm{E}))$	$b \to (\varepsilon^1, C_{\varepsilon^1}(\mathrm{E}))$
$f((f_1^1, C_{f_1^1}(\overline{\mathrm{E}}))) \to (f_1^1, C_{f_1^1}(\overline{\mathrm{E}}))$	$h((h_2^1, C_{h_2^1}(\overline{\mathrm{E}}))) \to (h_2^1, C_{h_2^1}(\overline{\mathrm{E}}))$	$b \to (g_3^1, C_{g_3^1}(\overline{\mathrm{E}}))$
$h((h_2^1, C_{h_2^1}(\overline{\mathrm{E}}))) \to (\varepsilon^1, C_{\varepsilon^1}(\mathrm{E}))$	$f((f_1^1, C_{f_1^1}(\overline{\mathrm{E}}))) \to (h_2^1, C_{h_2^1}(\overline{\mathrm{E}}))$	$b \to (h_2^1, C_{h_2^1}(\overline{\mathrm{E}}))$
$f((f_4^1, C_{f_4^1}(\overline{\mathrm{E}}))) \to (f_4^1, C_{f_4^1}(\overline{\mathrm{E}}))$	$f((f_4^1, C_{f_4^1}(\overline{\mathrm{E}}))) \to (h_5^1, C_{h_5^1}(\overline{\mathrm{E}}))$	$a \to (g_3^2, C_{g_3^2}(\overline{\mathrm{E}}))$
$h((h_2^1, C_{h_2^1}(\overline{\mathrm{E}}))) \to (f_1^1, C_{f_1^1}(\overline{\mathrm{E}}))$	$h((h_5^1, C_{h_5^1}(\overline{\mathrm{E}}))) \to (h_5^1, C_{h_5^1}(\overline{\mathrm{E}}))$	$b \to (h_5^1, C_{h_5^1}(\overline{\mathrm{E}}))$
$f((f_4^1, C_{f_4^1}(\overline{\mathrm{E}}))) \to (g_3^1, C_{g_3^1}(\overline{\mathrm{E}}))$	$h((h_5^1, C_{h_5^1}(\overline{\mathrm{E}}))) \to (f_4^1, C_{f_4^1}(\overline{\mathrm{E}}))$	$b \to (f_4^1, C_{f_4^1}(\overline{\mathrm{E}}))$
$g((g_3^1, C_{g_3^1}(\overline{\mathrm{E}})), (g_3^2, C_{g_3^2}(\overline{\mathrm{E}}))) \to (g_3^1, C_{g_3^1}(\overline{\mathrm{E}}))$	$h((h_5^1, C_{h_5^1}(\overline{\mathrm{E}}))) \to (g_3^1, C_{g_3^1}(\overline{\mathrm{E}}))$	

The number of states is $|Q| = 5$ *and the number of transition rules is* $|\Delta| = 15$. *The k-C-Continuation Automaton associated with* E *is given in Figure 4.*

Let $\sim_e$ be the equivalence relation over the set of states of $\mathscr{C}_{\mathrm{E}}$ defined for any two states $(f_j^k, C_{f_j^k}(\overline{\mathrm{E}}))$ and $(g_i^p, C_{g_i^p}(\overline{\mathrm{E}}))$ by $(f_j^k, C_{f_j^k}(\overline{\mathrm{E}})) \sim_e (g_i^p, C_{g_i^p}(\overline{\mathrm{E}})) \Leftrightarrow h(C_{f_j^k}(\overline{\mathrm{E}})) = h(C_{g_i^p}(\overline{\mathrm{E}}))$.

Proposition 6 ([14, 15]). *The automaton* $\mathscr{C}_{\mathrm{E}} / _{\sim_e}$ *is isomorphic to* $\mathscr{A}_{\mathrm{E}}$.

Example 6. *Using the equivalence-relation* $\sim_e$ *over the set of states of k-C-Continuation Automaton* $\mathscr{C}_{\mathrm{E}}$ *(Figure 4) we see that* $h(C_{f_1^1}(\overline{\mathrm{E}})) = h(C_{f_4^1}(\overline{\mathrm{E}}))$ *and* $h(C_{h_2^1}(\overline{\mathrm{E}})) = h(C_{h_5^1}(\overline{\mathrm{E}}))$. *The automaton* $\mathscr{C}_{\mathrm{E}} / _{\sim_e}$ *is given in Figure 5.*



$\{h(C_{f_1^1}(\overline{\mathrm{E}})), h(C_{f_4^1}(\overline{\mathrm{E}}))\}$
$\{h(C_{h_2^1}(\overline{\mathrm{E}})), h(C_{h_5^1}(\overline{\mathrm{E}}))\}$
$\{h(C_{\varepsilon^1}(\overline{\mathrm{E}}))\}$
$\{h(C_{g_3^1}(\overline{\mathrm{E}}))\}$
$\{h(C_{g_3^2}(\overline{\mathrm{E}}))\}$
b
f
h
g
a

Figure 5: The Automaton $\mathscr{C}_{\mathrm{E}}/_{\sim_e}$.

In order to show that the k-C-continuation tree automaton of E is isomorphic to the k-position automaton of E, we first show the link between the position functions and the C-continuations.

Proposition 7 ([14, 15]). *Let* E *be linear,* $1 \leq k \leq m$ *be two integers and* f *be a position in* $\Sigma_E \cap \Sigma_m$. *Then* $\mathrm{Follow}(E, f, k) = \mathrm{First}(C_{f^k}(\overline{\mathrm{E}}))$.

Lemma 3. *Let* E *be linear and* g *be a symbol in* $\Sigma_{\geq 1}$. *Then* $g^{-1}(\mathrm{E}) \neq \emptyset \Leftrightarrow g \in \mathrm{First}(\mathrm{E})$.

Corollary 3. *Let* E *be linear,* $1 \leq k \leq m$ *be two integers and* f *and* g *be two symbols in* Σ. *Then,* $g^{-1}(C_{f^k}(\mathrm{E})) \neq \emptyset \Leftrightarrow g \in \mathrm{First}(C_{f^k}(\mathrm{E}))$.

Lemma 4. *Let* E *be linear,* $1 \leq k \leq m$ *be two integers and* f *and* g *be two symbols in* Σ. *Then,* $g^{-1}(C_{f^k}(\mathrm{E})) \neq \emptyset \Leftrightarrow g \in \mathrm{Follow}(\mathrm{E}, f, k)$.

Proposition 8. *Let E be linear. The automaton* $\mathscr{C}_{\mathrm{E}}$ *is isomorphic to* $\mathscr{P}_{\mathrm{E}}$.

This proposition can be extended to expressions that are not necessarily linear since $\mathscr{C}_{\mathrm{E}}$ and $\mathscr{P}_{\mathrm{E}}$ are relabelings of $\mathscr{C}_{\overline{\mathrm{E}}}$ and $\mathscr{P}_{\overline{\mathrm{E}}}$.

Corollary 4. *The automaton* $\mathscr{C}_{\mathrm{E}}$ *is isomorphic to* $\mathscr{P}_{\mathrm{E}}$.

We define the similarity relation denoted by $\equiv$ over the set of states of the automaton $\mathscr{C}_{\mathrm{E}}$ as follows:

$$(f^k, C_{f^k}(\overline{\mathrm{E}})) \equiv (g^p, C_{g^p}(\overline{\mathrm{E}})) \Leftrightarrow \mathrm{Follow}(\overline{\mathrm{E}}, f, k) = \mathrm{Follow}(\overline{\mathrm{E}}, g, p).$$

Corollary 5. *The finite tree automaton* $\mathscr{C}_{\mathrm{E}}/_{\equiv}$ *is isomorphic to the follow automaton* $\mathscr{F}_{\mathrm{E}}$.

4 Comparison between the Equation and the Follow Automata

We discuss in this section two examples to compare the equation and the follow automata.

Let $\Sigma = \Sigma_0 \cup \Sigma_1$ be the ranked alphabet defined by $\Sigma_0 = \{a\}$ and $\Sigma_1 = \{f_1, \ldots, f_n\}$. Let us consider the linear regular expression $\mathrm{E} = ((f_1(a)^{*_a} \cdot_a f_2(a)^{*_a}) \cdot_a \ldots) \cdot_a f_n(a)^{*_a}))^{*_a}$ defined over Σ. Then the size of E is $|\mathrm{E}| = 4n - 1$ and its alphabet width is $||\mathrm{E}|| = n + 1$. We have $\mathrm{First}(\mathrm{E}) = \{a, f_1, f_2, \ldots, f_n\}$ and $\mathrm{Follow}(\mathrm{E}, f_1, 1) = \mathrm{Follow}(\mathrm{E}, f_2, 1) = \ldots = \mathrm{Follow}(\mathrm{E}, f_n, 1) = \{a, f_1, f_2, \ldots, f_n\}$.
The partial derivatives associated with E are:

$$\partial_{f_1}(\mathrm{E}) = \{(((a \cdot_a f_1(a)^{*_a} \cdot_a f_2(a)^{*_a}) \cdot_a \ldots) \cdot_a f_n(a)^{*_a}) \cdot_a \mathrm{E}\}$$
$$\partial_{f_2}(\mathrm{E}) = \{((a \cdot_a f_2(a)^{*_a} \cdot_a \ldots) \cdot_a f_n(a)^{*_a}) \cdot_a \mathrm{E}\}, \ldots$$

$$\partial_{f_n}(\mathrm{E}) = \{(a \cdot_a f_n(a)^{*_a}) \cdot_a \mathrm{E}\}.$$

The K-position automaton associated with E has $n+1$ states.
The follow automaton associated with E has 1 state.
The equation automaton associated with E has: $n+1$ states.

Let $\mathrm{F} = \underbrace{(f(a)^{*_a} + f(a)^{*_a} + \cdots + f(a)^{*_a})}_{f(a)^{*_a}\ n\text{-times}}$ be a regular expression defined over the ranked alphabet $\Sigma = \Sigma_0 \cup \Sigma_1$ such that $\Sigma_0 = \{a\}$ and $\Sigma_1 = \{f\}$. We have $|\mathrm{E}| = 4n-1$ and $||\mathrm{E}|| = n+1$. The linearized form associated with F is $\overline{\mathrm{F}} = (f_1(a)^{*_a} + f_2(a)^{*_a} + \cdots + f_n(a)^{*_a})$. The set $\mathrm{First}(\mathrm{F}) = \{a, f_1, f_2, \ldots, f_n\}$, $\mathrm{Follow}(\overline{\mathrm{F}}, f_1, 1) = \{a, f_1\}$, $\mathrm{Follow}(\overline{\mathrm{F}}, f_2, 1) = \{a, f_2\}, \ldots$, and $\mathrm{Follow}(\overline{\mathrm{F}}, f_n, 1) = \{a, f_n\}$.
The partial derivatives associated with F are $\partial_f(\mathrm{F}) = \{a \cdot_a f(a)^{*_a}\}$, $\partial_{ff}(\mathrm{F}) = \{a \cdot_a f(a)^{*_a}\}$.
The K-position automaton associated with F has $n+1$ states.
The follow automaton associated with F has: $n+1$ states.
The equation automaton associated with F has: 2 states.

From these examples we state that the two automata are incomparable:

Proposition 9. *The Follow tree automaton and the Equation Tree Automaton are incomparable though they are derived from two isomorphic automata,* i.e. *Neither is a quotient of the other.*

4.1 A smaller automaton

In [7] P. García *et al.* proposed an algorithm to obtain an automaton from a word regular expression. Their method is based on the computation of both the partial derivatives automaton and the follow automaton. They join two relations, the first relation is over the states of the word follow automaton and the second relation is over the word c-continuations automaton, in one relation denoted by $\equiv_V$. What we propose is to extend the relation $\equiv_V$ to the case of trees as follows:

$$C_{f_j^k}(\overline{\mathrm{E}}) \equiv_V C_{g_i^p}(\overline{\mathrm{E}}) \Leftrightarrow \begin{cases} (\exists C_{h_m^l}(\overline{\mathrm{E}}) \sim_{\mathscr{F}} C_{f_j^k}(\overline{\mathrm{E}}) \mid C_{h_m^l}(\overline{\mathrm{E}}) \sim_e C_{g_i^p}(\overline{\mathrm{E}})) \\ \vee\ (\exists C_{h_m^l}(\overline{\mathrm{E}}) \sim_{\mathscr{F}} C_{g_i^p}(\overline{\mathrm{E}})) \mid C_{h_m^l}(\overline{\mathrm{E}}) \sim_e C_{f_j^k}(\overline{\mathrm{E}})) \end{cases}$$

The idea is to define the follow relation $\sim_{\mathscr{F}}$ over the states of the c-continuation automaton $\mathscr{C}_\mathrm{E}$ as follows: $C_{f_j^k}(\overline{\mathrm{E}}) \sim_{\mathscr{F}} C_{g_i^p}(\overline{\mathrm{E}}) \Leftrightarrow \mathrm{Follow}(C_{f_j^k}(\overline{\mathrm{E}}), f_j, k) = \mathrm{Follow}(C_{g_i^p}(\overline{\mathrm{E}}), g_i, p)$ such that we keep all the equivalent k-C-Continuations in the merged states. The obtained automaton is denoted by $\mathscr{C}_\mathrm{E}/_{\sim_{\mathscr{F}}}$. Then apply the equivalence relation $\sim_e$ (apply the mapping h) over the states of the automaton $\mathscr{C}_\mathrm{E}/_{\sim_{\mathscr{F}}}$ and merge the states which have at least one expression in common.

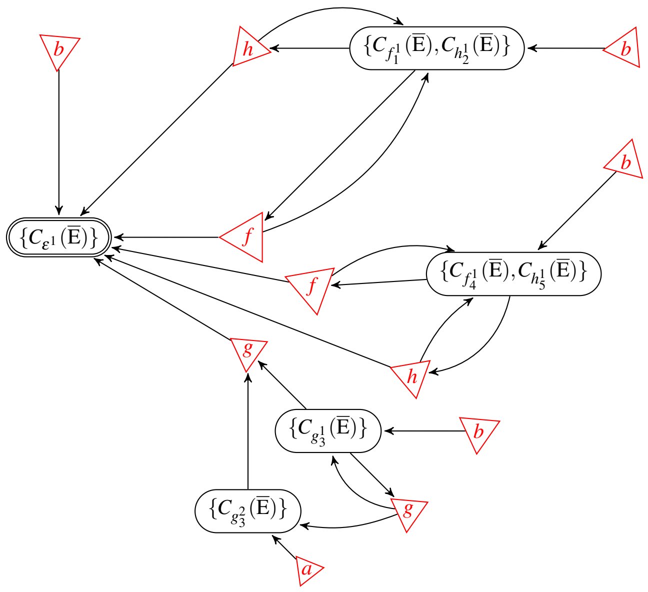
Example 7. *Let* $\mathrm{E} = (f(a)^{*_a} \cdot_a b + h(b))^{*_b} + g(c,a)^{*_c} \cdot_c (f(a)^{*_a} \cdot_a b + h(b))^{*_b}$ *defined in Example 1 and* $\overline{\mathrm{E}} = \underbrace{(f_1(a)^{*_a} \cdot_a b + h_2(b))^{*_b}}_{\mathrm{F}_1} + \underbrace{g_3(c,a)^{*_c}}_{\mathrm{G}_2} \cdot_c \underbrace{(f_4(a)^{*_a} \cdot_a b + h_5(b))^{*_b}}_{\mathrm{F}_3}$.

$$\begin{aligned}
C_{f_1^1}(\overline{\mathrm{E}}) &= ((a \cdot_a f_1(a)^{*_a}) \cdot_a b) \cdot_b (f_1(a)^{*_a} \cdot_a b + h_2(b))^{*_b}, \\
C_{h_2^1}(\overline{\mathrm{E}}) &= b \cdot_b (f_1(a)^{*_a} \cdot_a b + h_2(b))^{*_b}, \\
C_{g_3^1}(\overline{\mathrm{E}}) &= (c \cdot_c g_3(c,a)^{*_c}) \cdot_c (f_4(a)^{*_a} \cdot_a b + h_5(b))^{*_b}, \\
C_{g_3^2}(\overline{\mathrm{E}}) &= (a \cdot_c g_3(c,a))^{*_c}) \cdot_c (f_4(a)^{*_a} \cdot_a b + h_5(b))^{*_b}, \\
C_{f_4^1}(\overline{\mathrm{E}}) &= ((a \cdot_a f_4(a)^{*_a}) \cdot_a b) \cdot_b (f_4(a)^{*_a} \cdot_a b + h_5(b))^{*_b}, \\
C_{h_5^1}(\overline{\mathrm{E}}) &= b \cdot_b (f_4(a)^{*_a} \cdot_a b + h_5(b))^{*_b}.
\end{aligned}$$

Applying $\sim_{\mathscr{F}}$ *over the states of* $\mathscr{C}_\mathrm{E}$ *we obtain:* $C_{f_1^1}(\overline{\mathrm{E}}) \sim_{\mathscr{F}} C_{h_2^1}(\overline{\mathrm{E}})$ *then the two states are merged,* $C_{f_4^1}(\overline{\mathrm{E}}) \sim_{\mathscr{F}} C_{h_5^1}(\overline{\mathrm{E}})$ *so they are merged. The states* $C_{g_3^1}(\overline{\mathrm{E}})$ *and* $C_{g_3^2}(\overline{\mathrm{E}})$ *are not merged with anyone.*

The number of states is $|Q| = 5$ *and the number of transition rules is* $|\Delta| = 15$.

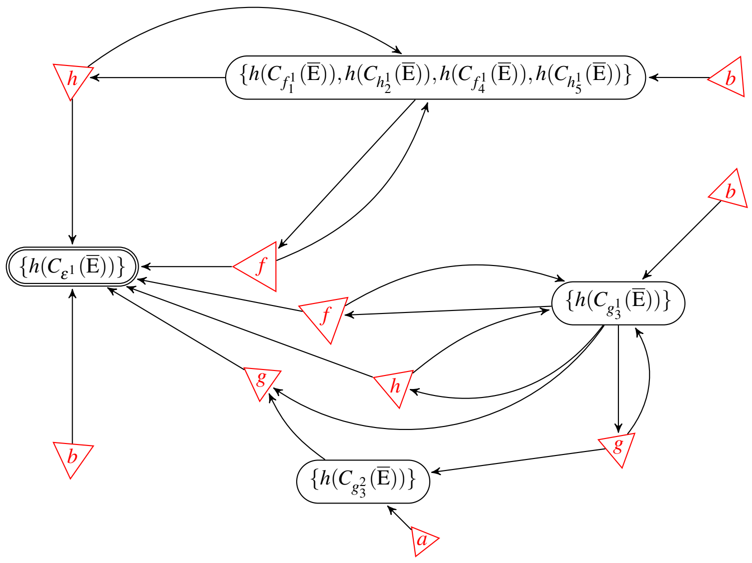
The quotient automaton of this automaton by the equivalence relation $\sim_{\mathscr{F}}$ is given in Figure 6.



$\{C_{\varepsilon^1}(\overline{\mathrm{E}})\}$
$\{C_{f_1^1}(\overline{\mathrm{E}}),C_{h_2^1}(\overline{\mathrm{E}})\}$
$\{C_{f_4^1}(\overline{\mathrm{E}}),C_{h_5^1}(\overline{\mathrm{E}})\}$
$\{C_{g_3^1}(\overline{\mathrm{E}})\}$
$\{C_{g_3^2}(\overline{\mathrm{E}})\}$
b
h
b
f
b
f
h
g
b
g
a

Figure 6: The Automaton $\mathscr{C}_{\mathrm{E}}/_{\sim_{\mathscr{F}}}$.

The quotient automaton of the automaton $\mathscr{C}_{\mathrm{E}}/_{\sim_{\mathscr{F}}}$ by the equivalent relation $\sim_e$ is given in Figure 7. The number of states is $|Q| = 4$ and the number of transition rules is $|\Delta| = 14$.



$\{h(C_{f_1^1}(\overline{\mathrm{E}})),h(C_{h_2^1}(\overline{\mathrm{E}})),h(C_{f_4^1}(\overline{\mathrm{E}})),h(C_{h_5^1}(\overline{\mathrm{E}}))\}$
$\{h(C_{\varepsilon^1}(\overline{\mathrm{E}}))\}$
$\{h(C_{g_3^1}(\overline{\mathrm{E}}))\}$
$\{h(C_{g_3^2}(\overline{\mathrm{E}}))\}$
h
b
b
f
f
h
g
g
b
a

Figure 7: The resulting automaton.

5 Conclusion

In this paper we define and recall different constructions of tree automata from a regular expression.

The different automata and their relations (quotient, isomorphism) defined in this paper are represented in Figure 8.

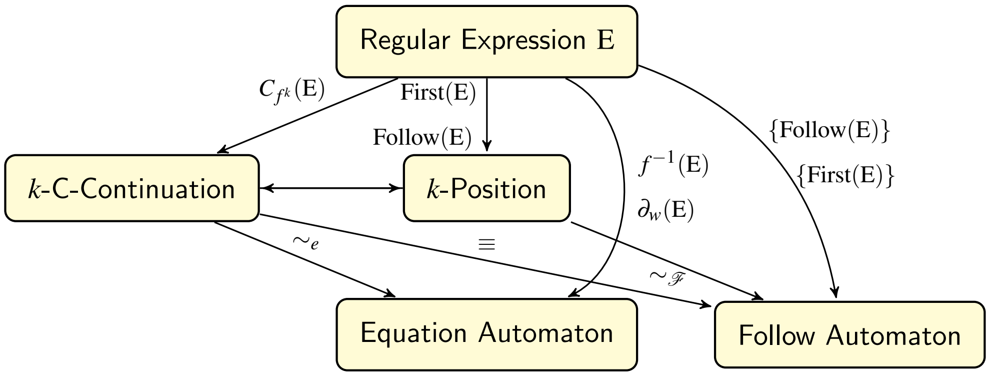


Figure 8: Relation between Automata

Looking for reductions of the set of states, we applied the algorithm by García *et al.* [7] which allowed us to compute an automaton the size of which is bounded above by the size of the smaller of the follow and the equation automata.

References

[1] Valentin M. Antimirov (1996): *Partial Derivatives of Regular Expressions and Finite Automaton Constructions*. Theor. Comput. Sci. 155(2), pp. 291–319. Available at `http://dx.doi.org/10.1016/0304-3975(95)00182-4`.

[2] Janusz A. Brzozowski (1964): *Derivatives of Regular Expressions*. J. ACM 11(4), pp. 481–494. Available at `http://doi.acm.org/10.1145/321239.321249`.

[3] Jean-Marc Champarnaud & Djelloul Ziadi (2001): *From C-Continuations to New Quadratic Algorithms for Automaton Synthesis*. IJAC 11(6), pp. 707–736. Available at `http://dx.doi.org/10.1142/S0218196701000772`.

[4] Jean-Marc Champarnaud & Djelloul Ziadi (2002): *Canonical derivatives, partial derivatives and finite automaton constructions*. Theor. Comput. Sci. 289(1), pp. 137–163. Available at `http://dx.doi.org/10.1016/S0304-3975(01)00267-5`.

[5] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, C. Loding, S. Tison & M. Tommasi (2007): *Tree Automata Techniques and Applications*. Available on: `http://www.grappa.univ-lille3.fr/tata`.

[6] Corinna Cortes, Patrick Haffner & Mehryar Mohri (2004): *Rational Kernels: Theory and Algorithms*. Journal of Machine Learning Research 5, pp. 1035–1062. Available at `http://www.ai.mit.edu/projects/jmlr/papers/volume5/cortes04a/cortes04a.pdf`.

[7] Pedro García, Damián López, José Ruiz & Gloria Inés Alvarez (2011): *From regular expressions to smaller NFAs*. Theor. Comput. Sci. 412(41), pp. 5802–5807. Available at `http://dx.doi.org/10.1016/j.tcs.2011.05.058`.

[8] V.-M. Glushkov (1961): *The abstract theory of automata*. Russian Mathematical Surveys 16, pp. 1–53.

[9] Lucian Ilie & Sheng Yu (2003): *Follow automata*. Inf. Comput. 186(1), pp. 140–162. Available at `http://dx.doi.org/10.1016/S0890-5401(03)00090-7`.

[10] Ahmed Khorsi, Faissal Ouardi & Djelloul Ziadi (2008): *Fast equation automaton computation*. J. Discrete Algorithms 6(3), pp. 433–448. Available at `http://dx.doi.org/10.1016/j.jda.2007.10.003`.

[11] Dietrich Kuske & Ingmar Meinecke (2011): *Construction of tree automata from regular expressions*. RAIRO - Theor. Inf. and Applic. 45(3), pp. 347–370. Available at `http://dx.doi.org/10.1051/ita/2011107`.

[12] Éric Laugerotte, Nadia Ouali Sebti & Djelloul Ziadi (2013): *From Regular Tree Expression to Position Tree Automaton*. In Adrian Horia Dediu, Carlos Martín-Vide & Bianca Truthe, editors: LATA, Lecture Notes in Computer Science 7810, Springer, pp. 395–406. Available at `http://dx.doi.org/10.1007/978-3-642-37064-9_35`.

[13] R. McNaughton & H. Yamada (1960): *Regular Expressions and State Graphs for Automata*. IEEE Trans. on Electronic Computers 9, pp. 39–47.

[14] Ludovic Mignot, Nadia Ouali Sebti & Djelloul Ziadi (2014): *An Efficient Algorithm for the Equation Tree Automaton via the k-C-Continuations*. In A. Beckmann, E. Csuhaj varjù & K. Meer (Eds.), editors: Computability in Europe- 10th International Conference, CiE 2014, Budapest, Hungary, June 23-27, 2014. Proceedings, Lecture Notes in Computer Science 8493, Springer, pp. 303–313.

[15] Ludovic Mignot, Nadia Ouali Sebti & Djelloul Ziadi (2014): *An Efficient Algorithm for the Equation Tree Automaton via the k-C-Continuations*. CoRR abs/1401.5951.

Boolean Circuit Complexity of Regular Languages

Maris Valdats
University of Latvia
Faculty of Computing
Riga, Raiņa Bulv. 19, Latvia
d20416@lanet.lv

In this paper we define a new descriptional complexity measure for Deterministic Finite Automata, BC-complexity, as an alternative to the state complexity. We prove that for two DFAs with the same number of states BC-complexity can differ exponentially. In some cases minimization of DFA can lead to an exponential increase in BC-complexity, on the other hand BC-complexity of DFAs with a large state space which are obtained by some standard constructions (determinization of NFA, language operations), is reasonably small. But our main result is the analogue of the ”Shannon effect” for finite automata: almost all DFAs with a fixed number of states have BC-complexity that is close to the maximum.

State complexity of deterministic finite automata (DFA) [1][5] has been analyzed for more than 50 years and all this time has been the main measure to estimate the descriptional complexity of finite automata. Minimization algorithm [6] for it was developed as well as methods to prove upper and lower bounds for various languages.

It is hard to find any evidence of another complexity measure for finite automata. Transition complexity [3] could be one, it counts the number of transitions, but there is not much use of it for DFAs (it is proportional to the state complexity), it is used in the nondeterministic case.

But intuitively not all DFAs with the same number of states have the same complexity. We try to illustrate it with the following example.

Consider a DFA that recognizes a language in the binary alphabet which consists of words in which there is an even number of ones among the last 1000 input letters. One can easily prove that it needs 2^{1000} states, however such a DFA can easily be implemented by keeping its state space in a 1000 bit register which remembers the last 1000 input letters.

On the other hand, consider a ”random” DFA with a binary input tape and 2^{1000} states. There is essentially no better way to describe it as with its state transition table which consists of 2^{1001} lines which (as it is widely assumed) is more than particles in our universe.

It is easy to represent a large number of states in a compact form: 2^n states fit into n state bits of the state register. This is true for the ”random” DFA as well. But the computation performed by the transition function on this register can be very easy in some cases and hard in some other. Therefore it seems natural to introduce a complexity measure for DFAs which measures the complexity of the transition function.

Automata with a large state space which is kept in a state register have been used before, but not in the widest sense. One example of such a usage is FAPKC [8] (Finite Automata Public Key Cryptosystem), a public key cryptosystem developed in the 80’s by Renji Tao. In FAPKC the state space of an automaton is considered to be a vector space and the transition function is expressed as a polynomial over a finite field.

In this paper we consider the following model: we arbitrarily encode the state space into a bit vector (state register) and express the transition function as a Boolean circuit. The BC-complexity of the DFA is (approximately) the complexity of this circuit and this notion extends to regular languages in a natural way.

BC-complexity was first analyzed in [9] where it was considered for transducers. Here we define it for DFAs what allows to extend the definition to regular languages.

The main result of this paper is the Shannon effect for the BC-complexity of regular languages: it turns out that most of the languages have BC-complexity that is close to the maximum. To obtain it we first estimate upper (and lower) bounds for BC-complexity compared to state complexity (Theorem. 3.3), afterwards by counting argument we show that the complexity of most of the languages is around this upper bound.

Influence of state minimization to BC-complexity were analyzed already in [9] for transducers and for DFAs it is essentially the same: it turns out that for some regular languages BC-complexity of their minimal automaton is much (superpolynomially) larger than BC-complexity for some other (non-minimal) DFA that recognizes it. Finally we look how BC-complexity behaves if we do some standard constructions on automata (determinization of an NFA, language operations).

1 Preliminaries

1.1 Finite Automata and Regular Languages

We use a standard notion of DFA [4], it is a tuple $(Q,\Sigma,\delta,q_0,\tilde{Q})$, where Q is the state space, Σ is the input alphabet, $\delta : \Sigma \times Q \to Q$ is the transition function, $q_0 \in Q$ is the start state and $\tilde{Q} \subseteq Q$ is the set of accepting states.

DFA starts computation in the state q_0 and in each step it reads an input letter $x \in \Sigma$ and changes its state. If the current state of a DFA is $q \in Q$ and it reads an input letter $x \in \Sigma$ then it moves to state $\delta(x,q)$. If after reading the input word DFA is in a state $q \in \tilde{Q}$ then this word is accepted, otherwise it is rejected. DFA A recognizes language L iff it accepts all words from this language and rejects all words not in the language. Two DFAs that recognize the same language are called equivalent.

The state complexity of a DFA is the number of states in its state space $C_s(A) = |Q|$. For each DFA A there is a unique minimal DFA $M(A)$ which is equivalent to A and has minimal state complexity. There is an effective minimization algorithm for finding it [1].

We will need the estimation of the number of DFAs with s states. Denote $\mathfrak{A}_s$ to be the number of pairwise non-equivalent minimal DFAs with s states over k-letter alphabet. In [2] it is estimated to be larger than $2^{s-1}(s-1)s^{(k-1)s}$, we will use the following reduced estimation (true for $s \geq 3$) which will be sufficient for us:

Theorem 1.1 ([2]) $\mathfrak{A}_s \geq 2^s s^{(k-1)s}$ *for* $s \geq 3$.

1.2 Boolean circuits

We will use the standard notion of a Boolean circuit and restrict our attention to circuits in the standard base (&, $\vee$, $\neg$). The size of the circuit $C(F)$ is the number of gates plus the number of outputs of the circuit F. Boolean circuit F with n inputs and m outputs represents a Boolean function $(y_1,\dots,y_m) = F(x_1,\dots,x_n)$ in a natural way.

Each function $f : \{0,1\}^n \to \{0,1\}^m$ can be represented by a Boolean circuit in (infinitely many) different ways. The complexity of this function $C(f)$ is the size of the smallest circuit that represents this function.

We will also need a formula for the upper bound of the number of different Boolean circuits with a given complexity C. Denote $N(n,m,C)$ to be the number of circuits with n input variables, m output variables and no more than C gates, that correspond to different Boolean functions. Then:

Theorem 1.2

$$N(n,m,C) \leq 9^{C+n}(C+n)^{C+m}$$

Proof Assign to inputs numbers from 1 to n, and numbers from $n+1$ to $n+C$ to the gates. Each gate is characterised with its two inputs (at most $(n+C)^2$ possibilities) and type (AND, OR, NOT, 3 possibilities). There are no more than $(n+C)^m$ ways how to assign outputs of the circuit and each circuit is counted $C!$ times, one for each numbering of gates. Therefore the total number of circuits can be estimated as:

$$N(n,m,C) < \frac{(3(C+n)^2)^C \cdot (C+n)^m}{C!} < 9^C(1+\frac{n}{C})^C(C+n)^{C+m},$$

here we have used, that $C! > C^C/3^C$ for all C.

Further, as $(1+1/x)^x < e < 9$ for arbitrary $x > 0$, then

$$(1+\frac{n}{C})^C = ((1+\frac{n}{C})^{\frac{C}{n}})^n < 9^n$$

from where the result follows. □

A classical result about Boolean functions states that most of functions $f : \{0,1\}^n \to \{0,1\}^m$ have approximately the same circuit complexity which is close to maximum. This property is called Shannon effect.

Theorem 1.3 ([7]) *For any Boolean function* $f : \{0,1\}^n \to \{0,1\}^m$

$$C(f) \lesssim \frac{m2^n}{n+\log m},$$

For almost all Boolean functions $f : \{0,1\}^n \to \{0,1\}^m$

$$C(f) \gtrsim \frac{m2^n}{n+\log m}.$$

Here and further $\log = \log_2$ and we use the notation

$$f(n) \lesssim g(n) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} \leq 1.$$

2 Encodings and Representations of a DFA

Classical representations of automata are table forms or state transition diagrams. They are essentially the same, a state diagram can be thought of as a visualization of a table form. Table form lists the transition function of an automaton as a table where each line corresponds to a pair of state and input letter. In state transition diagram each state is denoted by a circle and for each transition $(q,x) \to q'$ an arrow is drawn from state q to state q' above which letter x is written.

Both of these representations show each state of an automaton separately, therefore with these methods it is not possible to effectively describe an automaton with a large number of states.

One can encode s states into $\lceil \log(s) \rceil$ (or more) state bits which can be kept in a *state register*. Also, input letters can be encoded as a bit vectors. Every automaton has infinitely many such encodings.

The transition function in this case will take as an input a state register and an encoded input letter, and produce a (next) state register. It is thus a Boolean function and it is natural to represent it with a Boolean circuit.

Another question is how to represent the set of accepting states $\tilde{Q}$. We represent it by a Boolean circuit implementing its characteristic function. Therefore a representation of a DFA will consist of an encoding of its state space and input alphabet and two circuits: one for its transition function and one for the characteristic function of the set of accepting states. We call these circuits *transition circuit* and *acceptance circuit*, respectively.

An encoding $E(X)$ of a set X onto a binary string is an injective mapping $f_X : X \to \{0,1\}^{b_X}$ where b_X is the length of the encoding. As the mapping is injective then $b_X \geq \lceil \log|X| \rceil$.

An encoding of a DFA consists of an encoding of its input alphabet f_Σ and an encoding of the state space f_Q which we call input encoding and state encoding, respectively. Additionally for the state encoding we ask that the start state is encoded as a string of all zeros $f_Q(q_0) = 0^{b_Q}$.

Definition Let $A = (Q, \Sigma, \delta, q_0, \tilde{Q})$ be a given DFA and (f_Σ, f_Q) be its encoding. A pair of Boolean circuits (F, G) is a representation of A under encoding (f_Σ, f_Q) iff

- F has $b_\Sigma + b_Q$ input variables and b_Q output variables,
- G has b_Q input variables and one output variable,
- for all $x \in \Sigma$ and $q \in Q$ if $q' = \delta(x, q)$, then $f_Q(q') = F(f_\Sigma(x), f_Q(q))$,
- $G(f_Q(q)) = 1 \iff q \in \tilde{Q}$ for all $q \in Q$.

In other words, transition circuit F reads encoded input $f_\Sigma(x)$ as its first b_Σ input bits, encoded state $f_Q(q)$ as following b_Q input bits and has encoded next state $f_Q(q')$ as its b_Q output bits. Acceptance circuit G reads encoded state $f_Q(q)$ and outputs 1 as its only output bit iff $q \in \tilde{Q}$.

As noted before minimal values for b_Σ and b_Q are $\lceil \log(|\Sigma|) \rceil$ and $\lceil \log(|Q|) \rceil$ respectively, but they can be larger as well. Whether allowing them to be larger gives a possibility to construct smaller representations of DFAs, is an interesting open question.

It is natural to encode the state space Q with $|Q|$ lexicographically first bit strings of length $\lceil \log|Q| \rceil$, in such a case we will say that the state encoding is minimal. The notion of minimal input encoding is introduced similarly. We call an encoding of a DFA *minimal encoding* if both encodings: state and input are minimal.

3 BC-complexity

In this section we define the main concept of this paper, BC-complexity of a DFA. We start from the bottom:

Definition BC-complexity of a representation of a DFA (F, G) is the sum of complexities of its transition circuit and acceptance circuit and the number of state bits:

$$C_{\mathrm{BC}}((F,G)) = C(F) + C(G) + b_Q.$$

The number of state bits b_Q is included in the definition to avoid situation that an automaton has a large number of states but zero BC-complexity. It is natural to assume that it costs something to create a circuit even if it has no gates and this is one of the possibilities how to reflect this in the definition. Another possibility would be to use the complexity of ”wires” instead of the complexity of gates for the underlying circuits, but we prefer to use the standard complexity for the circuits.

Definition BC-complexity of a DFA A, $C_{\mathrm{BC}}(A)$, is the minimal BC-complexity of its representations:

$$C_{\mathrm{BC}}(A) = \min\{C_{\mathrm{BC}}((F,G)) : (F,G) \text{ represents } A\}.$$

Although the name ”circuit complexity” also sounds reasonable, we use the abbreviation ”BC-complexity” to avoid confusion with the circuit complexity of regular languages.

Definition BC-complexity of a regular language L is the minimal BC-complexity of all DFAs that recognize L:

$$C_{\mathrm{BC}}(L) = \min\{C_{\mathrm{BC}}(A) : A \text{ recognizes } L\}.$$

First we observe that we can optimize our acceptance circuit by rearranging states. If we encode states in such a way that all accepting states have smaller index than rejecting states (or vice versa) then the acceptance circuit can be reduced to a comparison operation whose complexity is not greater than $4n$ where n is the number of state bits.

But this is not the best optimization that can be achieved by rearranging states. For the upper bound in the following Theorem 3.1 different arrangement is used.

Theorem 3.1 *If $|\Sigma| = k \geq 2$ then for any DFA A with s states,*

$$\lceil \log(s) \rceil \leq C_{\mathrm{BC}}(A) \lesssim (k-1)s.$$

If $|\Sigma| = 1$ then for any DFA A with s states,

$$\lceil \log(s) \rceil \leq C_{\mathrm{BC}}(A) \lesssim \frac{s}{\log s}.$$

Proof Lower bound. For any representation (F,G) there are $b_Q \geq \lceil \log s \rceil$ state bits, therefore BC-complexity cannot be smaller than $\lceil \log s \rceil$.

For upper bound if we just construct an optimal representation (F,G) under some arbitrary minimal encoding (with $\lceil \log s \rceil$ state bits) then the BC-complexity of this representation according to Theorem 1.3 can be estimated as

$$C_{\mathrm{BC}}((F,G)) \leq C(F) + C(G) + \lceil \log s \rceil \lesssim \frac{ks\lceil \log s \rceil}{\log(ks\lceil \log s \rceil)} + \frac{s}{\log s} + \lceil \log s \rceil \lesssim ks$$

To improve the result to $(k-1)s$ we will choose a specific minimal encoding where states are ordered in a way that for one input letter the corresponding transition function is simple. Denote q to be the encoding of the current state, q' to be the encoding of the next state and x to be the encoding of the input letter. We split the transition circuit F in two parts F_1 and F_2 where part F_1 computes the next state for one specific input letter a and part F_2 does it for other $k-1$ input letters.

If we look at the state transition graph for the letter a then it splits into connected components each of which has the form

$$q_1 \to q_2 \to \ldots q_{m-1} \to q_m \to q_j$$

where $1 \leq j \leq m$. Each such component is uniquely defined with two numbers m (the number of states in it) and j (the length of the ”tail”), we call m to be the length of a component. We order all these components by m and j lexicographically what naturally leads to the ordering of states. Consider all components with parameters (m, j) and denote by $M = M(m, j)$ the index (encoding) of the first state of the first such component and by $N = N(m, j)$ the index of the last state of the last such component.

The transition function is $q' = q + 1$ except for the last state q_m of each component for which it is $q' = q - (m - j)$. As each of these components have m states then q corresponds to the last state of some component iff $q + 1 = M \mod m$.

The circuit F_1 should compute the following function $q' = F_1(q)$:

```
q' = q+1
for all pairs (m, j)
  if M(m, j)<=q<=N(m, j) and q+1 == M(m, j) mod m:
    q' = q-(m-j)
```

Here M and N are the boundaries within which all components with parameters (m, j) are placed. It is easy to check that circuits for subtraction $q' = q - (m - j)$ and comparison $(M \leq q \leq N)$ are of size $O(\log s)$, for modulo comparison $q + 1 = M \mod m$ it is of size $O(\log s^2)$. Therefore the total size of the circuit F_1 is $K * c \log s^2$ where K is the number of different pairs (m, j) that correspond to some components that are present in the transition graph and c is some constant. We need to estimate the maximum value of K.

One can easily see that maximum value of K is obtained when each component with parameters (m, j) appears exactly once and all the smallest components are used. Let u be the maximum length of a component (maximal value of m) under the condition that all the possible smallest components are used. For each m there are m possible different types of components $(1 \leq j \leq m)$ therefore $K \leq u(u + 1)/2$.

On the other hand the total length of all components up to the length $u - 1$ should be less than s:

$$\sum_{m=1}^{u-1} m^2 = \frac{(u-1)u(2u-1)}{6} \leq s$$

whence it follows that $u \leq 2\sqrt[3]{s}$. Therefore

$$K \leq \frac{u(u+1)}{2} \leq \frac{2\sqrt[3]{s}(2\sqrt[3]{s}+1)}{2} \leq 4s^{\frac{2}{3}}$$

.

The size of the transition circuit F_2 for the other $k - 1$ input letters can be estimated from Theorem 1.3:

$$C(F_2) \lesssim \frac{(k-1)s\lceil \log s \rceil}{\log((k-1)s\lceil \log s \rceil)} \lesssim (k-1)s.$$

The size of the acceptance circuit can be estimated (from Theorem 1.3) as $C(G) \lesssim \frac{s}{\log s}$.

After reordering of states we also have to ensure that the start state is 0. This can increase the complexity of both circuits by no more than $3 \log s$ (it is necessary to add at most n negations to the input of the transition circuit F, the output of the transition circuit F, and the input of the acceptance circuit G). There are also $\log s$ state bits which are included in the computation of BC-complexity. We omit these terms of logarithmic order in the computation of BC-complexity because asymptotically they are negligible.

The BC-complexity of the automaton therefore can be estimated as

$$C_{\mathrm{BC}}(A) \leq C(F) + C(G) + b_Q \lesssim 4c(\log s)^2 s^{2/3} + \frac{s}{\log s} + (k-1)s$$

If $k \geq 2$ then the dominant term of this expression is $(k - 1)s$ and $C_{\mathrm{BC}}(A) \lesssim (k - 1)s$. For one letter alphabet the dominant term is $s/\log s$, therefore $C_{\mathrm{BC}}(A) \lesssim \frac{s}{\log s}$. $\square$

Consider language L_n in binary alphabet $\Sigma = \{0,1\}$ such that $x \in L_n$ iff $|x| = k$ and $x_{k-n+1} = 1$ (the n-th letter from the end is ”1”). The state complexity of this language is 2^n, one has to remember in a state register the last n input letters. But the BC-complexity of it is n. Circuits F, G that represent the natural encoding of a DFA A_n that recognizes L_n have no gates, they are shown in figure 1. Therefore the BC-complexity of (the representation (F,G) of) A_n is the number of state bits which is n. This example shows that the lower bound of Theorem 3.1 is strictly reachable.

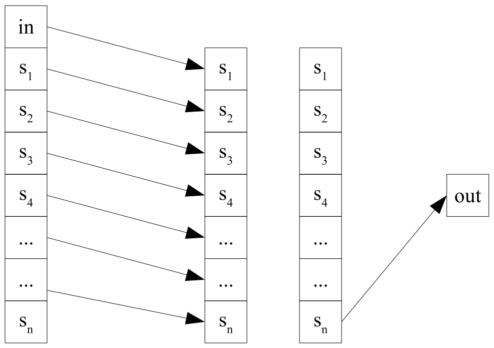


Figure 1: Representation (F,G) of the DFA A_n

Further we try to reach the upper bound. First we find a language (based on the Shannon function) for which the BC-complexity is at least $s/\log^2(s)$, afterwards by counting argument we show that BC-complexity for most languages is close to $(k-1)s$. That matches the upper bound of Theorem 3.1 and can be thought of as the Shannon effect for BC-complexity.

Denote by Sh_n the Shannon function on n bits: lexicographically first Boolean function with n input bits and one output bit with maximal complexity of its minimal circuit. Consider a language L_n^{Sh} that consists of all words $x_1x_2\ldots x_k$ in binary alphabet such that $Sh_n(x_{k-n+1}, x_{k-n+2}, \ldots, x_k) = 1$. State complexity of this language is not larger than 2^n: it is enough to remember the last n input letters. But its BC-complexity is at least $2^n/n^2$.

Theorem 3.2 *BC-complexity of L_n^{Sh} is at least $2^n/n^2$.*

Proof Let (F,G) be a pair of Boolean circuits that represents some DFA A_n recognizing L_n^{Sh}. Assume F has one input bit that represents input letter from the tape and $m = b_Q$ state bits. By concatenating n circuits F together with one circuit G as in figure 2. (state bit output of j-th circuit is passed as state bit input of $j+1$-st) one can obtain a circuit whose size is not larger than $nC(F) + C(G)$ and which computes Shannon function Sh_n on its n input bits. From Theorem 1.3 the complexity of this circuit is at least $2^n/n$. From $nC(F) + C(G) > 2^n/n$ we get that $C(F) + C(G) > 2^n/n^2$. □

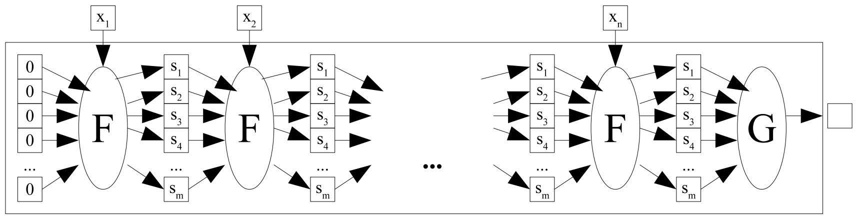


Figure 2: Circuit construction for the Shannon function Sh_n

Theorem 3.2 shows that for some language L_n^{Sh} with s states its BC-complexity is at least $s/(\log s)^2$. Next theorem is an extension of this result. With the use of nonconstructive methods (counting argument) one can show that this value can be raised up to $(k-1)s$. But in the beginning we will need a formula to estimate the number of automata with a given BC-complexity.

Theorem 3.3 *Fix Σ and denote $\mathfrak{A}(c)$ to be the class of those minimal DFAs whose BC-complexity is less than c. If $|\Sigma| = k \geq 2$ then for any $\varepsilon > 0$*

$$\lim_{s\to\infty} \frac{|\mathfrak{A}((1-\varepsilon)(k-1)s)|}{|\mathfrak{A}_s|} = 0$$

If $|\Sigma| = 1$ then for any $\varepsilon > 0$

$$\lim_{s\to\infty} \frac{|\mathfrak{A}((1-\varepsilon)\frac{s}{\log s})|}{|\mathfrak{A}_s|} = 0$$

Proof By Theorem 1.1 $\mathfrak{A}_s \geq 2^s s^{(k-1)s}$. Denote $l = 2^k$, it is clear that no more than l input bits for data input will be used for the representation for which BC-complexity is minimal. If more bits are used, then some of them will be equal as there are only 2^k functions that maps k inputs letters to $\{0,1\}(bits)$.

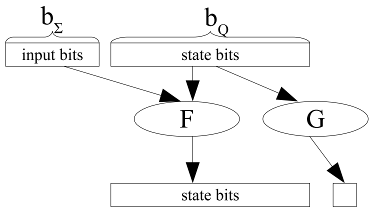


Figure 3: Merged acceptance and transition circuits

Consider a representation (F,G) of some encoding $E(A)$ of A. Merge these two circuits F and G and obtain one circuit H with $b_Q + b_\Sigma$ inputs and $b_Q + 1$ output bits, the first b_Q of which correspond to the output of the transition circuit F, but the last output bit corresponds to the output of the acceptance circuit G (Figure 3). The complexity of this circuit H is $C(F) + C(G)$, for any two minimal automata these ”merged” circuits will be different.

Now we want to estimate the number of representations with BC-complexity less that c. Such representations have at least $\lceil \log s \rceil$ and no more than c state bits. The complexity of the ”merged” circuit H for a representation with b_Q state bits cannot be more than $c - b_Q$, the number of such circuits H from theorem 1.2 is not larger than

$$N(b_Q + b_\Sigma, b_Q + 1, c - b_Q) < N(b_Q + l, b_Q + 1, c - b_Q) < 9^{c+l}(c+l)^{c+1}.$$

Therefore the number of representations with complexity c is not larger than

$$\sum_{b_Q=1}^{c} N(b_Q + l, b_Q + 1, c - b_Q) < c9^{c+l}(c+l)^{c+1} < 9^{c+l}(c+l)^{c+2}.$$

To prove the theorem we have to show that

$$\lim_{s\to\infty} \frac{9^{c+l}(c+l)^{c+2}}{2^s s^{(k-1)s}} = 0$$

or what is equivalent to that

$$\lim_{s\to\infty} \log\left(9^{c+l}(c+l)^{c+2}\right) - \log\left(2^s s^{(k-1)s}\right) = -\infty$$

for the stated values of c.

For $k \geq 2$ if we substitute $c = (1-\varepsilon)(k-1)s$ then after simplification we obtain an equation of the form

$$\lim_{s\to\infty} -\varepsilon s \log s + O(s) = -\infty$$

which is true. The same happens in the case $k = 1$ if we substitute $c = (1-\varepsilon)s/(\log s)$. $\square$

We have shown in Theorem 3.1 that BC-complexity for any regular language with state complexity s and input alphabet of size $k \geq 2$ is not "much larger" than $(k-1)s$. Theorem 3.3 states that for minimal encodings recognition of almost all such languages would require circuits of size around $(k-1)s$. This can be thought of as the "Shannon effect" for the BC-complexity of automata: for almost all automata its value is close to the maximum.

4 Minimization of BC-complexity

For the state complexity of DFA an efficient minimization algorithm [6] is well known which, given a DFA, finds the state complexity of it as well as the the minimal DFA itself. This is in a big contrast with complexity measures of general programs (Turing machines) for which their complexity (space or time) cannot be determined by any means in the general case.

It is easy to notice that finding the BC-complexity of a DFA is NP-hard.

Theorem 4.1 *Finding the BC-complexity of a DFA given its arbitrary representation is NP-hard.*

Proof We will reduce SAT problem to finding minimal BC-complexity of a DFA. Given a SAT problem instance that contains n variables, consider a DFA with n state bits (2^n states), that works in one letter alphabet, its state transition function is a "circle", that goes through all the states, and accepting states are those, for which this SAT instance gives positive output.

Now assume that this SAT instance is not satisfiable — then this DFA never accepts and therefore its minimal DFA has 1 state (0 state bits) and its BC-complexity is 0. If this SAT instance is satisfiable, then any representation of it will have some state bits and therefore its BC-complexity will be at least 1. Therefore if one could efficiently find BC-complexity of a given DFA, he could also solve any SAT problem. $\square$

Further we show one interesting property of BC-complexity — that for some DFAs BC-complexity is significantly smaller than for their equivalent minimal DFAs. The theorem is based on the conjecture that *PSPACE* $\not\subseteq$ *P*/*Poly*. The proof of this theorem for transducers can be found in [9], for DFAs it is almost the same and is omitted here. Denote by $M(L)$ the minimal DFA recognizing language L.

Theorem 4.2 *If there is a polynomial $p(x)$ such that $C_{\mathrm{BC}}(M(L)) < p(C_{\mathrm{BC}}(L))$ for all regular languages L then PSPACE $\subseteq$ P/Poly.*

It means that in some cases by minimizing the number of states (minimizing state complexity) BC-complexity of the transition function can increase superpolynomially. And on the other hand, sometimes allowing equivalent states in the automaton helps to keep BC-complexity small.

5 BC-complexity applications

5.1 Nondeterministic automata

Theorems 3.3 and 3.1 suggest that for most DFAs in k-letter alphabet with s states BC-complexity is around $(k-1)s$. But in many cases when DFAs with a large state space are constructed by some standard method, it turns out that their BC-complexity is exponentially smaller than this maximal expected value — it is of order $Polylog(s)$. Further we look at some of these standard constructions starting with the determinization of an NFA.

Theorem 5.1 *If a language R over alphabet Σ, $|\Sigma| = k$ can be recognized by an NFA N with n states and t transitions, then it can also be recognized by a DFA A for which $C_{BC}(A) \leq t + (k+1)n + k\log k$.*

Proof Consider a DFA A that is obtained by a standard construction from NFA N. Its set of states is the powerset of the set of states of N. The state space of A will consist of 2^n states (may be some of them will not be reachable), which can be encoded in n state bits. Each state bit of an encoding of A will correspond to one state of N. For input letters we choose arbitrary minimal input encoding into $\log k$ bits.

The transition circuit of A can be obtained from the transition function of N. NFA N after reading input letter $x \in \Sigma$ will be in state q_i, if there is a state q_j, in which it was before (NFA can be in many states simultaneously) and from which reading input letter x leads to state q_i. Denote by Q^i_a subset of states of N from which reading letter a leads to state q_i. Denote by Q_t a subset of states in which N is after reading t letters. If N reads input letter a in step t then:

$$q_i \in Q_{t+1} \leftrightarrow (Q_t \cap Q^i_a) \neq \emptyset.$$

In the circuit it means that if x denotes the encoded input letter then

$$q'_i = \bigvee_{a\in\Sigma} ((x = a) \& \bigvee_{q\in Q^i_a} q).$$

To construct all k subcircuits $x = a$ we need $\log k$ negations and $k(\log k - 1)$ conjunctions.

The size of the block $\& \bigvee_{q\in Q^i_a} q$ is the number of transitions entering state q on input a therefore the total number of these inner disjunctions and conjunctions for all output bits q'_i is t. There are also $(k-1)n$ outer disjunctions $\bigvee_{a\in\Sigma}$. In total the complexity of the transition circuit is not larger than $k(\log k - 1) + \log k + (k-1)n + t \leq t + (k-1)n + k\log k$.

Acceptance circuit G is a disjunction of all the final states of N, the complexity of this it is not larger than $n-1$. Also $b_Q = n$ have to be added to the BC-complexity. Therefore the total BC-complexity of A is not larger than $t + (k+1)n + k\log k$. □

As the number of transitions is not larger than kn^2 then

Corollary 5.2 *If a language R in alphabet Σ, $|\Sigma| = k$ can be recognized with an NFA N with n states, then it can also be recognized with a DFA A for which $C_{BC}(A) \leq kn^2 + (k+1)n + k\log k$.*

5.2 Language operations

State complexity of language operations has been studied long ago, e.g. in [10]. The result of some of the operations (e.g. reversing) can lead to exponentially larger automata than the original one. Here we analyze how BC-complexity changes with languages operations and observe that in those cases when the state complexity increases exponentially it leads to automata whose state transition function is very structured therefore its BC-complexity is exponentially smaller than state complexity.

For all operations we assume that we are given two languages L_1 and L_2 and $m = C_s(L_1)$, $n = C_s(L_2)$, $a = C_{\mathrm{BC}}(L_1)$, $b = C_{\mathrm{BC}}(L_2)$, $k = |\Sigma|$. We start with the union and intersection.

Theorem 5.3 *If $L_3 = L_1 \cup L_2$ or $L_3 = L_1 \cap L_2$ then $C_{\mathrm{BC}}(L_3) \leq a+b+1$.*

Proof Assume circuits (F_1, G_1) represent a DFA recognizing L_1 and (F_2, G_2) represent a DFA recognizing L_2. The transition function for a DFA recognizing L_3 would consist of circuits F_1 and F_2 working in parallel. The acceptance circuit consists of circuits G_1 and G_2 working on corresponding parts of bit vector followed by a disjunction (for union) or conjunction (for intersection) gate. The number of state bits is the sum of state bits for representations (F_1, G_1) and (F_2, G_2). The complexity of such a representation is $C(F_1) + C(F_2) + C(G_1) + C(G_2) + 1 + b_Q = a + b + 1$. □

The complement of the language can be computed by the same pair of circuits as the language itself with negation added at the end of the acceptance circuit.

Theorem 5.4 *If $L_3 = \Sigma^* \setminus L_1$ then $C_{\mathrm{BC}}(L_3) \leq a+1$.*

A word $x_1x_2\ldots x_n$ belongs to the reverse language L_1^R iff $x_n \ldots x_2x_1$ belongs to L_1. NFA N recognizing L_1^R can be obtained from the DFA A recognizing L_1 by setting the start state of N to be any accepting state of A, setting q_0 of A to be the only accepting state of N and reversing all the arrows. DFA recognizing L_1^R can be obtained from N by running the standard process of determinization.

Theorem 5.5 $C_{BC}(L_1^R) \leq (2k+1)m + k\log k$

Proof This follows directly from Theorem 5.1 and the fact, that NFA obtained by reversing all the transitions has exactly km transitions. □

Language L_1L_2 which is the concatenation of languages L_1 and L_2 consists of all words uw such that $u \in L_1$ and $w \in L_2$.

Theorem 5.6 $C_{\mathrm{BC}}(L_1L_2) \leq a + (2k+1)n + k\log k$

Proof Assume DFA A_1 recognizes L_1, DFA A_2 recognizes L_2. NFA that recognizes L_1L_2 can be obtained from A_1 and A_2 by adding ε-transitions from all the accepting states of A_1 to the start state of A_2. The standard construction of DFA from this NFA can be optimized — it will consist of circuits F_1 and G_1 representing A_1 together with a circuit $N(A_2)$ constructed from A_2 as from NFA as in Theorem 5.1. Circuit G_1 sets state bit corresponding to state q_0 of A_2 to "1" iff A_1 is in accepting state.

By Theorem 5.1 $C(N(A_2)) \leq t + (k+1)n + k\log k$ and, since A_2 is a deterministic automaton, $t = kn$. Together it gives that $C_{\mathrm{BC}}(L_1L_2) \leq C(F_1) + C(G_1) + C(N(A_2)) \leq a + kn + (k+1)n + k\log k = a + (2k+1)n + k\log k$. □

Theorem 5.7 $C_{\mathrm{BC}}(L_1^*) \leq km^2 + (k+1)m + k\log k$.

Proof NFA recognizing L_1^* can be obtained from DFA recognizing L_1 by adding ε-transitions from all the accepting states to the start state. The resulting NFA therefore also has m states and the result follows from Corollary 5.2. □

Operation	State complexity	BC-complexity
$L_1 \cup L_2$	mn	$a+b+1$
$L_1 \cap L_2$	mn	$a+b+1$
$\Sigma^* - L_1$	m	$a+1$
L^R	2^m	$(2k+1)m+k\log k$
L_1L_2	$(2m-1)2^{n-1}$	$a+(2k+1)n+k\log k$
L_1^*	$2^{m-1}+2^{m-2}$	$km^2+(k+1)m+k\log k$

Table 1: State complexity and BC-complexity of language operations

6 Conclusions and open problems

In this paper a new measure of complexity, BC-complexity of DFAs and regular languages, was considered. Transition function of a DFA as well as the characteristic function of the set of accepting states are expressed as Boolean circuits and their circuit complexity is taken as a complexity measure (BC-complexity) of this DFA. It turns out that BC-complexity can vary exponentially for DFA with the same number of states (Theorem 3.1). Theorem 3.3 states that almost all DFAs BC-complexity is close to maximum (”Shannon effect”).

In all asymptotic constructions minimal encodings for state and input alphabet where used, but it is not known if minimal encodings are always optimal. We think that sometimes they are not, but showing an example where other encoding than minimal would be more efficient (in the sense of minimizing BC-complexity) is an interesting open question.

In section 4 it was shown that BC-complexity of a regular language can be much smaller than the BC-complexity of the minimal DFA that recognizes it. On the other hand, DFAs with a large state space that are obtained in many standard operations (determinization of NFA, language operations), have a ”good” structure so that their BC-complexity can be relatively small.

References

[1] Trakhtenbrot B. Barzdins J.: *Finite Automata: Behavior and synthesis*. Science, Moscow.

[2] Michael Domaratzki, Derek Kisman & Jeffrey Shallit (2002): *On the number of distinct languages accepted by finite automata with n states*. *Journal of Automata, Languages and Combinatorics* 7(4), pp. 469–486.

[3] Gregor Gramlich & Georg Schnitger (2007): *Minimizing nfa's and regular expressions*. *Journal of Computer and System Sciences* 73(6), pp. 908 – 923, doi:10.1016/j.jcss.2006.11.002.

[4] Mealy George H. (1955): *A method for synthesizing sequential circuits*. *Bell System Technical Journal* 34(5), pp. 1045–1079, doi:10.1002/j.1538-7305.1955.tb03788.x.

[5] J.E. Hopcroft & J.D. Ullman (1979): *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Cambridge.

[6] John Hopcroft (1971): *An $n\log n$ Algorithm for Minimizing States in a Finite Automaton*. *Theory of Machines and Computations*, pp. 189–196.

[7] Lupanov O.B. (1984): *Asymptotic Estimates of Complexity of Control Systems*. Moscow University Press.

[8] Tao R. (2009): *Finite Automata and Application to Cryptography*. Springer.

[9] Maris Valdats (2011): *Transition Function Complexity of Finite Automata*. In Markus Holzer, Martin Kutrib & Giovanni Pighizzini, editors: *Proc. of DCFS*, *Lecture Notes in Computer Science* 6808, Springer, pp. 301–313, doi:10.1007/978-3-642-22600-7.

[10] Sheng Yu (2000): *State Complexity of Regular Languages*. *Journal of Automata, Languages and Combinatorics* 6, pp. 221–234.

A Simple Character String Proof of the “True but Unprovable” Version of Gödel’s First Incompleteness Theorem

Antti Valmari
Tampere University of Technology, Department of Mathematics
PO Box 553, FI-33101 Tampere, FINLAND
Antti.Valmari@tut.fi

A rather easy yet rigorous proof of a version of Gödel’s first incompleteness theorem is presented. The version is “each recursively enumerable theory of natural numbers with 0, 1, $+$, $\cdot$, $=$, $\land$, $\neg$, and $\forall$ either proves a false sentence or fails to prove a true sentence”. The proof proceeds by first showing a similar result on theories of finite character strings, and then transporting it to natural numbers, by using them to model strings and their concatenation. Proof systems are expressed via Turing machines that halt if and only if their input string is a theorem. This approach makes it possible to present all but one parts of the proof rather briefly with simple and straightforward constructions. The details require some care, but do not require significant background knowledge. The missing part is the widely known fact that Turing machines can perform complicated computational tasks.

Mathematics Subject Classification 2010: 03F40 Gödel numberings and issues of incompleteness

1 Introduction

Kurt Gödel’s first incompleteness theorem [2] is certainly one of the most important results in mathematical logic. Together with an improvement by Barkley Rosser [11], the theorem says that *any recursive sufficiently strong theory of natural numbers either proves a contradiction, or leaves both some sentence and its negation without a proof*. (We postpone discussion on Gödel’s original formulation to Section 8, because it uses a concept that cannot be explained briefly at this stage. “Recursive” and other background concepts are informally introduced in Section 2.)

More recently, the theorem has often been presented in the form *any recursively enumerable sufficiently expressive theory of natural numbers either proves a sentence that does not hold or fails to prove a sentence that does hold*. This form is not equivalent to Gödel’s and Rosser’s formulation. In some sense it promises less and in some sense more. However, it is easier to prove and perhaps also easier to understand. It makes the assumption of sufficient expressiveness (explained in Section 2) instead of the stronger assumption of sufficient strength (explained in Section 8). This is the version discussed in the major part of this paper. It is compared to Gödel’s and Rosser’s formulation in Section 8.

Both Gödel’s original proof and most of the modern expositions are long and technical. On the other hand, its overall strategy can be explained rather briefly and is intuitively inspiring. As a consequence, the proof is one of the most popularized ones. We only mention here the excellent book by Douglas R. Hofstadter [3]. Unfortunately, to really grasp the result, the technicalities are necessary.

The goal of this paper is to present a *rigorous* proof which, excluding one detail, can be checked *in full* by a reader with *little background* (but not necessarily with little effort). We hope that our proof makes the result accessible to a wider audience than before. The skipped detail is the fact that some simple things can be computed by so-called Turing machines. Its rigorous proof would take many dull pages. On the other hand, Turing machines have been very widely accepted as a universal theoretical model of computation. Therefore, as long as it is obvious that something could be programmed in a modern programming language, it is common practice to skip the proof that a Turing machine can compute it.

Our trick is to first prove that theories of finite character strings with string literals, concatenation, and equality are incomplete. Then we derive the incompleteness of natural number arithmetic as a corollary. In this way, the main constructions of the proof are made using finite character strings, while other proofs make them using natural numbers. This makes our constructions much simpler and much more understandable. The presentation of our proof in this paper is not remarkably short, but this is partly due to the fact that it is very detailed.

Some background concepts are informally explained in Section 2. Our language on finite character strings is defined in Section 3. Not every character can be represented by itself in a string literal. Therefore, an encoding of characters is needed. Section 4 shows that the claim "string y is the sequence of the encodings of the characters in string x" can be formulated in the language. Computations of Turing machines are encoded in Section 5. The incompleteness of theories of strings is shown in Section 6, and of theories of arithmetic in Section 7. Section 8 compares the version of the theorem in this paper to Gödel's and Rosser's versions. Discussion on related work and the conclusions are in Section 9.

An earlier, not peer-reviewed version of this paper appeared as arXiv:1402.7253v1.

2 Informal Background

A *recursive theory* consists of a language for formulating claims about some domain of discourse, together with a recursive proof system. In the case of Gödel's theorem, the domain of discourse is the natural numbers 0, 1, 2, ... together with addition ($+$), multiplication (variably denoted with $\times$, $\cdot$, $*$, or nothing such as in $3x+1$), and equality ($=$) with their familiar properties. A *sentence* is a claim without input, formulated in the language. For instance, "3 is a prime number" lacks input but "p is a prime number" has p as input. Of course, whether or not a claim can be formulated depends on the language.

When Gödel published his theorem, the notion of "recursive proof system" had not yet developed into its modern form. Indeed, instead of "recursive", he used a word that is usually translated as "effective". Gödel meant a mathematical reasoning system for proving sentences, where any proof could be checked against a fixed set of straigthforward rules. Proofs were checked by humans, but the requirement was that they could do that in a mechanical fashion, without appealing to intuition on the meaning of formulae. This makes the proof system independent of the different insights that different people might have.

Today, "in a mechanical fashion" means, in essence, "with a computer that has at least as much memory as needed". It suffices that there is a computer program that inputs a finite character string and eventually halts if it is a valid proof, and otherwise runs forever. If such a program exists, then there also is a program that systematically starts the former program on finite character strings one by one in increasing length and executes them in parallel until a proof for the given sentence is found. (We will see in Section 7 how all finite character strings can be scanned systematically.) If the sentence has a proof, the program eventually finds it and halts; otherwise it runs forever in a futile attempt to find a proof.

For mathematical analysis, computers and their programs are usually formalized as *Turing machines*. We will introduce Turing machines in Section 5.

We will not need the assumption that a proof system resembles mathematical reasoning systems. Indeed, we will not need any other assumption than machine-checkability. So we define a *recursively enumerable proof system* as any Turing machine M that reads a finite character string and does or does not halt, such that if the string is not a sentence, then M does not halt. If the string is a sentence, it is considered as proven if and only if M halts. A *recursive proof system* adds to this the requirement of the existence of another Turing machine that halts precisely on those inputs, on which M does not halt.

Section 1 assumed that the language for expressing claims about natural numbers is sufficiently expressive. It suffices that the language has constant symbols 0 and 1, an unbounded supply of variable symbols, binary arithmetic operators $+$ and $\cdot$, binary relation symbol $=$, binary logical operator $\wedge$ (that is, "and"), unary logical operator $\neg$ (that is, "not"), the so-called universal quantifier $\forall$, and parentheses $($ and $)$. All symbols have their familiar syntactical rules and meanings. The universal quantifier is used to formulate claims of the form $\forall n : P(n)$ (that is, "for every natural number n, $P(n)$ holds").

For convenience, logical or $\vee$ can also be used without changing the expressiveness of the language, because it can be built from $\wedge$ and $\neg$, since $P \vee Q$ is logically equivalent to $\neg(\neg P \wedge \neg Q)$. Also logical implication $\rightarrow$, existential quantifier $\exists$, inequality $\neq$, less than $<$, and all familiar numeric constants 2, 3, ..., 9, 10, 11, ... can be used, because $P \rightarrow Q$ is equivalent to $\neg(P \wedge \neg Q)$, $\exists x : P(x)$ is equivalent to $\neg\forall x : \neg P(x)$, $x \neq y$ is equivalent to $\neg(x = y)$, $x < y$ is equivalent to $(\exists z : x + z + 1 = y)$, and any such numeric constant has the same value as some expression of the form $(1 + 1 + \ldots + 1)$.

3 A First-Order Language on Finite Character Strings

In this section we define our language for expressing claims about *finite character strings*, that is, finite sequences of characters. We use `this font` when writing in that language. To make it explicit where a string in that language ends and ordinary text continues, we put white space on both sides of the string even if normal writing rules of English would tell us not to do so. So we write "characters `a` , ..., `z` are" instead of "characters `a`, ..., `z` are".

The language uses the following characters:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9 " \ = ≠ ( ) ~ & | - > A E : + * <
```

We chose this set of characters for convenience. Any finite set containing at least two characters could have been used, at the expense of a somewhat more complicated proof. This set facilitates the use of familiar notation for many things. The fact that its size 53 is a prime number will be exploited in Section 7. The characters `a` , ..., `z` are *lower case letters*, and `0` , ..., `9` are *digits*.

A *finite character string* or just *string* is any finite sequence of characters.

An *encoded character* is `\0` , `\1` , or any other character than `"` and `\` . The encoded character `\0` denotes the character `\` , `\1` denotes `"` , and each remaining encoded character denotes itself. A *string literal* is any sequence of characters of the form `"`α`"` , where α is any finite sequence of encoded characters. It denotes the corresponding sequence of (unencoded) characters. For instance, `""` denotes the empty string and `"backslash=\1\0\1"` denotes the string `backslash="\"` . The purpose of encoding is to facilitate the writing of `"` inside a string literal, without causing confusion with the `"` that marks the end of the literal.

A *variable* is any string that starts with a lower case letter and then consists of zero or more digits. For instance, `a` , `x0` , and `y365` are variables but `49` and `cnt` are not. The value of a variable is a string. We say that the variable *contains* the string.

A *term* is any non-empty finite sequence of variables and/or string literals. It denotes the concatenation of the strings that the variables contain and/or string literals denote. For instance, `"theorem"` , `"theo""rem"` and `"the""""o""rem"` denote the same string `theorem` . If the variable `x` contains the string `or` , then also `"the"x"em"` denotes `theorem` .

An *atomic proposition* is any string of the form t`=`u or of the form t`≠`u , where t and u are terms. The first one expresses the claim that the strings denoted by t and u are the same string, and the second one expresses the opposite claim. So `"theorem"="theo""rem"` is a true atomic proposition, and `"theorem"≠"theo""rem"` is not. Indeed, t`≠`u expresses the same claim as `~`t`=`u , where `~` is introduced soon.

A *formula* is either an atomic proposition or any string of the following forms, where π and ρ are formulae and x is a variable: `(`π`)` , `~`π , π`&`ρ , π`|`ρ , π`->`ρ , `A`x`:`π , and `E`x`:`π . The parentheses `(` and `)` are used like in everyday mathematics, to force the intended interpretation. In the absence of parentheses, formulae are interpreted according to the following precedences: concatenation has the highest precedence, then `=` , `~` , `&` , `|` , `->` , and `:` in this order. For instance, `~b=c|c="hello"&a=bc` denotes the same as `(~(b=c))|((c="hello")&(a=bc))` , and `Ax:~x="8"|x="8"->"u"="u"` denotes the same as `Ax:(((~(x="8"))|(x="8"))->("u"="u"))` . All operators associate to the left, so π`->`ρ`->`σ means the same as `(`π`->`ρ`)->`σ .

The formulae express the following claims:

`(`π`)`	expresses the same claim as π,
`~`π	expresses that π does not hold,
π`&`ρ	expresses that π and ρ hold,
π`|`ρ	expresses that π holds or ρ holds or both hold,
π`->`ρ	expresses that if π holds, then also ρ holds,
`A`x`:`π	expresses that for any string x, π holds, and
`E`x`:`π	expresses that there is a string x such that π holds.

To improve readability, we often add spaces into a formula, like `Ax: x≠"8" | x="8"` . We may also split a formula onto many lines.

A *first-order language* is any language whose formulae are built from atomic propositions like above. The constants or literals, terms, and atomic propositions of a first-order language may be chosen as appropriate to the domain of discourse. A variable of a first-order language may only contain a value in the domain of discourse, while a variable of a higher-order language may be used more flexibly. When talking about first-order languages in general, we use the symbols $\neg$, $\forall$, and so on, and when talking about a particular first-order language specified in this paper, we use `~` , `A` , and so on.

We will need long formulae. To simplify reading them, we introduce abbreviations. The first abbreviation claims that variable `a` contains a character, that is, a string of length precisely one. The formula consists of `a="`x`"` for each encoded character x, separated by `|` and surrounded by `(` and `)` . We do not write it in full but instead write $\cdots$ to denote the missing part.

Char(`a`) :⇔ `( a="a" | a="b" |` $\cdots$ `| a="<" )`

The abbreviation was written for variable `a` , but clearly a similar abbreviation can be written for any variable. So we may use the abbreviations Char(`b`) , Char(`g75`) , and so on.

The next abbreviation claims that `x` is a substring of `y` . That is, there are strings `u` and `v` such that string `y` is the same as string `u` followed by string `x` followed by string `v` .

Sb(`x`, `y`) :⇔ `( Eu:Ev: y=uxv )`

When this abbreviation is used with `u` in the place of `x` , some other variable has to be used instead of `u` on the right hand side. That is, Sb(`u`, `y`) does not abbreviate `( Eu:Ev: y=uuv )` but, for instance, `( Ez:Ev: y=zuv )` . The incorrect interpretation `( Eu:Ev: y=uuv )` contains a *name clash*, that is, the variable `x` that is distinct from `u` in `y=uxv` in the definition of Sb(`x`, `y`) , became the same variable as `u` . In general, when interpreting an abbreviation containing a subformula of the form `A`x`:`π

or `Ex:`π , it may be necessary to replace x by some other variable, to avoid name clashes. Further information on this issue can be found in textbooks on logic, in passages that discuss "bound" and "free" variables.

Please keep in mind that abbreviations are not part of our language. They are only a tool for compactly referring to certain formulae that are too long to be written in full. Each string that uses abbreviations denotes the string that is obtained by replacing the abbreviations by their definitions, changing variable names in the definitions as necessary to avoid name clashes.

4 A Formula Expressing the Encoding of Characters

In this section we show that a formula `Q(x, y)` can be written that claims that `y` is the encoding of `x` , that is, `y` is obtained by replacing `\0` for each `\` and `\1` for each `"` in `x` . We start with a formula claiming that `y` is obtained by replacing `v` for one instance of `u` inside `x` .

```
RepOne(x, u, v, y)  :⇔  ( Ee:Ef: x=euf & y=evf )
```

The next formula claims that, under certain assumptions mentioned below, `y` is obtained by replacing `v` for every instance of `u` inside `x` . It converts `x` to `y` by making the replacements one by one. It assumes that `v` has no characters in common with `u` , so that no fake instances of `u` can occur inside or overlapping `v` . Furthermore, it assumes that different instances of `u` in `x` do not overlap, so that the result is independent of the order in which the instances are chosen for replacement. It also assumes that `p` (for punctuation) is a string that does not occur inside `x` , `y` , or any intermediate result. Furthermore, `p` cannot overlap with itself. We will later see how `p` is constructed.

The sequence of replacements is represented by `s` as a sequence of the form $\mathtt{p}x_1\mathtt{p}x_2\mathtt{p}\cdots\mathtt{p}x_n\mathtt{p}$, where $x_1 = \mathtt{x}$, $x_n = \mathtt{y}$, and $x_2, \ldots, x_{n-1}$ are the intermediate results. The requirements on `p` guarantee that `s` can be decomposed into this form in precisely one way. The parts `(Et: s=pxpt)` and `(Et: s=tpyp)` guarantee that $x_1 = \mathtt{x}$ and $x_n = \mathtt{y}$. Thanks to `~Sb(u, y)` , *every* instance of `u` is replaced. The rest of the formula picks each x_i other than the last and claims that x_{i+1} is obtained from it by making one replacement. The x_i is represented by `h` and x_{i+1} by `k` . They are distinguished by not containing `p` , being preceded by `p` , being separated from each other by `p` , and being succeeded by `p` .

```
RepAll(x, u, v, y, p)  :⇔  ( Es:
        (Et: s=pxpt) & (Et: s=tpyp) & ~Sb(u, y)
      & Ah:Ak: ( Sb(phpkp, s) & ~Sb(p, h) & ~Sb(p, k) ) -> RepOne(h, u, v, k)
)
```

To emphasize that abbreviations *are not* but *stand for* strings in our language, and that the strings they stand for are often not easy to comprehend, we now show the string that `RepAll(x, u, v, y, p)` stands for. The real string is too long to be shown on one line, so we split it on two lines.

```
(Es:(Et:s=pxpt)&(Et:s=tpyp)&~(Ez:Ev:y=zuv)&Ah:Ak:((Eu:Ev:s=uphpkpv)&
~(Eu:Ev:h=upv)&~(Eu:Ev:k=upv))->(Ee:Ef:h=euf&k=evf))
```

To obtain the punctuation string `p` , we first make variable `q` contain some sequence of `:`-characters that does not occur inside `x` . Such a string exists, because the string consisting of $n+1$ `:`-characters meets the requirements, when n is the length of `x` .

```
Punct(x, q)  :⇔  ( ( Aa: Sb(a, q) & Char(a) -> a=":" ) & ~Sb(q, x) )
```

The `p` used above is obtained as `"+"q` , that is, by adding a `+`-character to the front of the sequence of `:`-characters in variable `q` . So the value of `p` is `+:::::` or some similar sequence with a different number of `:`-characters. It clearly neither overlaps with itself nor occurs within `x` .

To guarantee that `u` and `v` do not have characters in common, we first convert each instance of `\` to `"*"q` , that is, to some string of the form `*:::···:` that does not occur inside `x` . Then each `*:::···:` is converted to `\0` , then each `"` to `*:::···:` , and finally each `*:::···:` to `\1` . In the first conversion, `u` consists of a single character, so different instances of `u` do not overlap. The same holds for the third conversion. In the second and fourth conversion, `u` is `*:::···:` , which clearly cannot overlap with itself. Furthermore, `+:::···:` does not overlap and is not inside `*:::···:` , so `p` cannot occur in `y` or any intermediate result.

We are now ready to write `Q(x, y)` . In it, the value `\` is represented by the string literal `"\0"` , `\1` by `"\01"` , and so on.

```
Q(x, y)  :⇔  ( Eq:  Punct(x, q)
             & Ex1: RepAll( x, "\0",  "*"q, x1, "+"q)
             & Ex2: RepAll(x1, "*"q, "\00", x2, "+"q)
             & Ex3: RepAll(x2, "\1",  "*"q, x3, "+"q)
             &      RepAll(x3, "*"q, "\01",  y, "+"q) )
```

5 Encoding Turing Machine Computations

Turing machines are a formal model of computation. In this section we show that for each Turing machine, there is a formula `Pvble(x)` that yields true if and only if the machine eventually halts, given `x` as the input. We call it `Pvble(x)` , because the Turing machine is thought to represent some proof system such that it halts if and only if `x` can be proven.

Details of the definition of Turing machines vary in the literature. To start our definition, we introduce a new symbol ⊔, called *blank*. Let *b-strings* be defined similarly to strings, but they may also contain blanks. So our Turing machines use 54 symbols: 53 characters and the blank. A Turing machine consists of a *control unit*, a *read/write head*, and a *tape* that consists of an infinite number of *cells* in both directions. Each cell on the tape may contain any character or ⊔. When we say that some part of the tape is blank, we mean that each cell in it contains ⊔. At any instant of time, the read/write head is on some cell of the tape. During a computation step, the read/write head rewrites the content of the cell and then possibly moves to the previous or the next cell, as dictated by the control unit and the contents of the cell before the step.

The control unit consists of *states* and *rules*. The states are numbered from 0 to r, for some positive integer r. State 0 is called the *final state*. There are $54r$ rules, one for each state q other than 0 and for each character c and ⊔. A rule is of the form $(c,q) \mapsto (c',q',d)$, where c' is any character or ⊔, q' is any state, and d is either `L` , `R` , or `N` . The meaning of the rule is that if the control unit is in state q and the tape cell under the read/write head contains c, then the machine writes c' on the cell, moves the read/write head one cell to the left or right or does not move it, and the control unit enters its state q'. If the control unit enters state 0, then computation halts.

Initially, the tape contains a finite sequence of characters, written somewhere on the tape. This finite sequence is the input to the machine. The rest of the tape is initially blank. Initially, the read/write head is on the first input character (or just anywhere, if the input is empty), and the control unit is in state 1.

At any instant of time, let the *right b-string* mean the content of the cell under the read/write head, the content of the next cell to the right, and so on, up to and including the last character on the tape. If the cell under the read/write head and all cells to the right are blank, then the right b-string is empty. So the last symbol of a non-empty right b-string is always different from $\sqcup$. Let the *left b-string* be defined similarly, but starting at the cell immediately to the left of the read/write head, and proceeding to the left until the first character on the tape is taken. Again, the left b-string may be empty, and if it is not, then its last symbol is not $\sqcup$. The contents of the tape as a whole are an infinite sequence of blanks extending to the left, then the left b-string reversed, then the right b-string, and then an infinite sequence of blanks extending to the right. Initially, the left b-string is empty and the right b-string contains the input.

A halting computation corresponds to a sequence (λ_0, q_0, ρ_0), (λ_1, q_1, ρ_1), $\ldots$, (λ_n, q_n, ρ_n), where λ_0 is the empty string, $q_0 = 1$, ρ_0 is the input string, $q_n = 0$, $q_i \neq 0$ when $0 \leq i < n$, and each (λ_i, q_i, ρ_i) for $1 \leq i \leq n$ is obtained from $(\lambda_{i-1}, q_{i-1}, \rho_{i-1})$ as follows. Here λ_i is the left b-string and ρ_i is the right b-string after i computation steps. Let $c = \sqcup$ if ρ_{i-1} is empty, and otherwise let c be the first symbol of ρ_{i-1}. There is a unique rule of the form $(c, q_{i-1}) \mapsto (c', q', d)$. We have $q_i = q'$. The b-strings λ_i and ρ_i are obtained by replacing c' for the first symbol of ρ_{i-1}, with special treatment of the case that ρ_{i-1} is empty or $c' = \sqcup$; and then possibly moving the first symbol of the resulting b-string to the front of λ_{i-1}, or moving a symbol in the opposite direction, again with some special cases. The special cases are discussed in more detail later in this section. The moving of a symbol from the right b-string to the left b-string models the movement of the read/write head one cell to the right, and the moving of a symbol in the opposite direction models the movement of the read/write head one cell to the left.

We want to model this sequence in our language on strings. The states q_i are represented simply by writing their numbers using the digits `0` , `1` , ..., `9` in the usual way. That is, state number 32768 is represented by `32768` . The b-strings λ_i and ρ_i are more difficult, because they may contain blanks, but there is no blank character in our language. So we represent $\sqcup$ with `\2` and `\` with `\0` . To simplify later constructions by remaining systematic with the encoding in Section 3, we also represent `"` with `\1` . All other characters represent themselves. To summarize, `\` , `"` , and $\sqcup$ on the tape of the Turing machine are represented in λ_i and ρ_i by the values `\0` , `\1` , and `\2` , whose string literal representations are `"\00"` , `"\01"` , and `"\02"` . In this sense, `\` and `"` become doubly encoded.

So we define an *encoded symbol* as `\0` , `\1` , `\2` , or any other character than `\` and `"` . We need not (and could not) say that $\sqcup$ is not an encoded symbol, because $\sqcup$ is not a character at all.

```
EChar(e)  :⇔
     ( e="\00" | e="\01" | e="\02" | Char(e) & e≠"\0" & e≠"\1" )
```

The next formula expresses that `y` is obtained from `x` by replacing `e` for its first encoded symbol, with special treatment of the empty strings and the blank. If `x` consists of at most one encoded symbol, `x` as a whole is overwritten. The result is the empty string if `e` is the encoded blank, and otherwise the result is `e` . If `x` consists of more than one encoded symbols, ordinary replacement occurs.

```
Write(x, e, y)  :⇔  (
        ( x="" | EChar(x) ) & ( e="\02" & y="" | e≠"\02" & y=e )
     | ( Ef:Ez: x=fz & EChar(f) & z≠"" & y=ez )
)
```

Let the encoded form of the left b-string be called *left string*, and similarly with the right b-string. The next formula expresses the removal of the first encoded symbol from one string and its addition to the front of another string, again with special treatment of the empty strings and the blank. The variables `f1` and `f2` contain the values of the from-string before and after the operation, and `t1` and `t2` contain the to-string. The encoded blank `\2` is never added to the front of an empty to-string, to maintain the rule that the b-strings never end with the blank. If the from-string is empty, then the operation behaves as if the encoded blank were extracted from it.

```
Move(f1, t1, f2, t2)   :⇔   (
        ( f1="" & t1="" & f2="" & t2="" )
      | ( f1="" & t1≠"" & f2="" & t2="\02"t1 )
      | ( f1="\02"f2 & t1="" & t2="" )
      | ( Ee: EChar(e) & f1=ef2 & (e≠"\02" | t1≠"") & t2=et1 )
)
```

Next we introduce a formula for each rule $(c,q) \mapsto (c',q',d)$. Let $\dot{c} =$ `\0` , if $c =$ `\` ; $\dot{c} =$ `\1` , if $c =$ `"` ; $\dot{c} =$ `\2` , if $c = \sqcup$; and otherwise $\dot{c} = c$. Let $\ddot{c} =$ `\00` , if $c =$ `\` ; $\ddot{c} =$ `\01` , if $c =$ `"` ; $\ddot{c} =$ `\02` , if $c = \sqcup$; and otherwise $\ddot{c} = c$. We define $\dot{c}'$ and $\ddot{c}'$ similarly. Let $\dot{q}$ denote q written using `0` , `1` , …, `9` in the usual way, and similarly with $\dot{q}'$.

We consider first the case where $d =$ `N`. The first part of the formula checks that the rule triggers, that is, the current state is q and the symbol under the read/write head is c, taking into accout the possibility that the right string is empty. The second part of the formula gives the state of the control unit, the right string, and the left string new values as dictated by the rule.

```
Rule$_{c,q}^{c',q',\mathrm{N}}$(l1, r1, q1, l2, r2, q2)   :⇔   (
        q1="$\dot{q}$" & ( r1="" & "$\ddot{c}$"="\02" | Ex: r1="$\ddot{c}$"x )
      & q2="$\dot{q}'$" & Write(r1, "$\ddot{c}'$", r2) & l2=l1
)
```

Rules with $d =$ `R` or $d =$ `L` are similar, but the moving of the read/write head is also represented.

```
Rule$_{c,q}^{c',q',\mathrm{R}}$(l1, r1, q1, l2, r2, q2)   :⇔   (
        q1="$\dot{q}$" & ( r1="" & "$\ddot{c}$"="\02" | Ex: r1="$\ddot{c}$"x )
      & q2="$\dot{q}'$" & Er: Write(r1, "$\ddot{c}'$", r) & Move(r, l1, r2, l2)
)
Rule$_{c,q}^{c',q',\mathrm{L}}$(l1, r1, q1, l2, r2, q2)   :⇔   (
        q1="$\dot{q}$" & ( r1="" & "$\ddot{c}$"="\02" | Ex: r1="$\ddot{c}$"x )
      & q2="$\dot{q}'$" & Er: Write(r1, "$\ddot{c}'$", r) & Move(l1, r, l2, r2)
)
```

Let $\ddot{\lambda}_i$ and $\ddot{\rho}_i$ be obtained from λ_i and ρ_i by replacing each symbol c in them with $\ddot{c}$. The computation of the Turing machine is represented as a string `c` of the form

`"`$\ddot{\lambda}_0$`"`$\ddot{\rho}_0$`\3`$\dot{q}_0$`"`$\ddot{\lambda}_1$`"`$\ddot{\rho}_1$`\3`$\dot{q}_1$`"`$\cdots$`"`$\ddot{\lambda}_n$`"`$\ddot{\rho}_n$`\3`$\dot{q}_n$`"` .

Because the $\dot{q}_i$ consist of just digits and the $\ddot{\lambda}_i$ and $\ddot{\rho}_i$ have been encoded, `"` and `\3` cannot occur inside them. So they can be used for separating the $\ddot{\lambda}_i$, $\ddot{\rho}_i$, and $\dot{q}_i$ from each other.

We are ready to write the formula that claims that the Turing machine halts on input `x` . It says that there is a sequence `c` that models the computation. First, `c` starts with the empty left string, the encoded input string as the right string, and 1 as the state. Second, `c` ends with 0 as the state. Finally, each `"`$\ddot{\lambda}_i$`"`$\ddot{\rho}_i$`\3`$\dot{q}_i$`"`$\ddot{\lambda}_{i+1}$`"`$\ddot{\rho}_{i+1}$`\3`$\dot{q}_{i+1}$`"` satisfies some rule. That `q1` and `q2` do not pick more from `c` than they should follows from the fact that the rules check that they consist of digits only.

```
Pvble(x)   :⇔   ( Ec:
          ( Et:Ey: Q(x, y) & c="\1\1"y"\031\1"t ) & ( Et: c=t"\030\1" )
        & ( Al1:Ar1:Aq1: Al2:Ar2:Aq2:
                  ~Sb("\1", l1) & ~Sb("\1", r1) & ~Sb("\1", l2) & ~Sb("\1", r2)
                & Sb("\1"l1"\1"r1"\03"q1"\1"l2"\1"r2"\03"q2"\1", c)
            -> (          Rule$_1$   (l1, r1, q1, l2, r2, q2)
                  | ⋯ |  Rule$_{54r}$ (l1, r1, q1, l2, r2, q2) ) )
)
```

6 Incompleteness of Theories of Finite Character Strings

In this section we prove that any recursively enumerable proof system for our language on strings either fails to prove some true sentence, or proves some false sentence.

Please remember that `Q(x, y)` and `Pvble(x"\1"y"\1")` are abbreviations used in this paper to improve readability, and not as such strings in our language. They stand for some strings in our language that are too long to be written explicitly in this paper. Each of these two long strings has a corresponding encoded string, which is obtained by replacing `\0` for each `\` and `\1` for each `"` . We denote them with `Q̇(x, y)` and `Ṗvble(x"\1"y"\1")` . Also remember that `Q(x, y)` claims that `y` is the encoded form of `x` . Therefore,

`Q( Q(x, y), Q̇(x, y) )` and `Q( Pvble(x"\1"y"\1"), Ṗvble(x"\1"y"\1") )` hold.

We can now write Gödel's famous self-referential sentence in our framework as follows.

```
 Ex:Ey: Q(x, y) & ~Pvble(x"\1"y"\1") & x=
"Ex:Ey: Q̇(x, y) & ~Ṗvble(x"\1"y"\1") & x="
```

Let α be any string and β be its encoded form. Then `x="`β`"` says that the variable `x` has the value α. Therefore, the last part of Gödel's sentence says that the value of `x` is the following string, with `Q(x, y)` and `Pvble(x"\1"y"\1")` replaced by the strings they stand for:

```
Ex:Ey: Q(x, y) & ~Pvble(x"\1"y"\1") & x=
```

This and `Q(x, y)` together say that the value of `y` is the following string, with `Q̇(x, y)` and `Ṗvble(x"\1"y"\1")` replaced by the strings they stand for:

```
Ex:Ey: Q̇(x, y) & ~Ṗvble(x"\1"y"\1") & x=
```

The string literal `"\1"` denotes the string `"` . Thus the value of `x"\1"y"\1"` is the value of `x` followed by `"` followed by the value of `y` followed by `"` . Remembering that spaces and division to lines are only for simplifying reading and not part of the real string, we see that the value of `x"\1"y"\1"` is Gödel's sentence. Furthermore, `~Pvble(x"\1"y"\1")` claims that `x"\1"y"\1"` is not provable. To summarize, the other parts of Gödel's sentence make `x"\1"y"\1"` be Gödel's sentence, and `~Pvble(x"\1"y"\1")` says that it is not provable. Altogether, Gödel's sentence claims that Gödel's sentence is not provable.

The formula `Pvble()` specifies a proof system for strings. Gödel's sentence is not a single sentence, instead, each proof system for strings has its own `Pvble()` and thus its own Gödel's sentence. Gödel's sentence of a proof system for strings claims that Gödel's sentence of that system is not provable in that system.

The Turing machine that halts immediately independently of the input represents a proof system for strings that proves every sentence. This proof system is useless, because for any sentence that it proves, it also proves its negation. So it proves many false sentences. However, it serves as an example of a proof system that proves its own Gödel's sentence.

Consider now any proof system for strings that proves its own Gödel's sentence. Because the sentence claims that the system does not prove it, the system has proven a false sentence. Consider then any proof system for strings that does not prove its own Gödel's sentence. Its Gödel's sentence thus expresses a true claim, and is thus a true sentence that the system does not prove.

We have proven the following.

Theorem 1 *Each recursively enumerable proof system for the first-order language on finite character strings with string literals, concatenation, and =, either proves a false sentence or fails to prove a true sentence.*

That is, there is no recursively enumerable proof system for strings that proves precisely the true sentences and nothing else. No recursively enumerable proof system for strings can precisely capture the true claims on strings that can be expressed in our language. This is the incompleteness theorem for strings.

7 Incompleteness of Natural Number Arithmetic

In this section we show that natural number arithmetic can simulate strings and their concatenation, and conclude that also natural number arithmetic is incomplete.

Our language on natural number arithmetic uses the same characters as our language on strings in Section 3. A *number literal* is either `0` or any non-empty finite sequence of digits that does not start with `0`. A *variable* is any string that starts with a lower case letter and then consists of zero or more digits. The value of a variable is a natural number. A *term* is a variable, a number literal, or any of the following, where t and u are terms: `(`t`)`, t`+`u, or t`*`u. The parentheses are used in the familiar way, `+` denotes addition, and `*` denotes multiplication. Furthermore, `*` has higher precedence than `+`, that is, t`+`u`*`v denotes the same as t`+(`u`*`v`)`. Atomic propositions and formulae are defined like in Section 3.

We now introduce a one-to-one correspondence between strings and natural numbers. Let $p = 53$, and let the 53 characters in the character set be given numbers from 1 to 53. If c'_i is a character, then let its number be denoted with c_i. The string $c'_1c'_2\cdots c'_n$ has the number

$$num(c'_1c'_2\cdots c'_n) \quad = \quad c_1p^{n-1}+c_2p^{n-2}+\ldots+c_{n-2}p^2+c_{n-1}p+c_n \quad .$$

So the empty string has the number 0, and the number of any string consisting of precisely one character is the number of that character. Let ι_n denote the string of length n whose every character has number 1. We have $num(\iota_n) = p^{n-1}+\ldots+p+1$, and ι_0 is the empty string.

We have to show that this mapping is indeed one-to-one. To do that, for each string $c'_1c'_2\cdots c'_n$ we introduce a *successor* and prove that the number of the successor is always one bigger than the number of the string itself. If $c_i = p$ for every $1 \leq i \leq n$, then the successor is defined as ι_{n+1}. We have

$$\begin{array}{r} num(\iota_{n+1}) \\ -\,num(c'_1c'_2\cdots c'_n) \end{array} = \begin{array}{rrrr} p^n + & p^{n-1} + & \ldots + & p+1 \\ -\,pp^{n-1} - & pp^{n-2} - & \ldots - & p\cdot 1 \end{array} = 1 \,.$$

In the opposite case, at least one of $c_1, \ldots, c_n$ is not p. Let j be the last such index, that is, $1 \leq j \leq n$, $c_j \neq p$, and $c_i = p$ when $j < i \leq n$. The successor is defined as the string $d'_1d'_2\cdots d'_n$, where $d_i = c_i$ when $1 \leq i < j$, $d_j = c_j+1$, and $d_i = 1$ when $j < i \leq n$. We have

$$\begin{array}{l} num(d'_1d'_2\cdots d'_n) - num(c'_1c'_2\cdots c'_n) = \\ \quad\; c_1p^{n-1} + \ldots + c_{j-1}p^{n-j+1} + (c_j+1)p^{n-j} \qquad\qquad + \; p^{n-j-1} + \ldots + \; p+1 \\ \; - c_1p^{n-1} - \ldots - c_{j-1}p^{n-j+1} - \qquad c_jp^{n-j} - pp^{n-j-1} - pp^{n-j-2} - \ldots - p\cdot 1 \\ = 1 \,. \end{array}$$

We see that the empty string, its successor, the successor of that string, and so on are in one-to-one correspondence with the natural numbers 0, 1, 2, and so on. It remains to be proven that this sequence of strings covers all strings. It does not contain any string twice, because the corresponding natural numbers are all distinct. So it contains infinitely many distinct strings. For any n, there is only a finite number of strings of length n. So the sequence cannot get stuck at any length n. The only case where the successor is of different length than the string itself is when the successor is ι_{n+1}. So the sequence covers at least ι_0, ι_1, ι_2, and so on. Between ι_n and ι_{n+1}, including ι_n but not ι_{n+1}, the sequence goes through

$num(\iota_{n+1}) - num(\iota_n) = p^n$ strings of length n. The number of strings of length n is p^n, so the sequence goes through all of them.

We have shown that our correspondence between strings and natural numbers is one-to-one.

Our next task is to represent concatenation of strings as a formula on their numbers. The definition of *num* yields immediately

$$num(c'_1 \cdots c'_n d'_1 \cdots d'_m) \;=\; p^m num(c'_1 \cdots c'_n) + num(d'_1 \cdots d'_m) \quad .$$

To present this in our language, we have to extract p^m from $num(d'_1 \cdots d'_m)$ only using the language. Let $y = num(d'_1 \cdots d'_m)$. We have $num(\iota_m) \leq y < num(\iota_{m+1})$, that is, $p^{m-1} + \ldots + 1 \leq y < p^m + \ldots + 1$. Multiplying this by $p-1$ we get $p^m - 1 \leq y(p-1) < p^{m+1} - 1$, to which adding $y+1$ yields $p^m + y \leq yp + 1 < p^{m+1} + y$. If $m' > m$, then $p^{m'} + y \leq yp + 1$ does not hold, and if $m' < m$, then $yp+1 < p^{m'+1} + y$ does not hold. Therefore, $k = p^m$ if and only if k is a power of p and $k + y \leq yp + 1 < pk + y$.

A prime number is a natural number greater than 1 that cannot be represented as a product of two natural numbers greater than 1. If p is a prime number and $p^m = xy$, then, for some $0 \leq i \leq m$, $x = p^i$ and $y = p^{m-i}$. Therefore, and because 53 is a prime number, the property that k is a power of 53 can be formulated as follows.

```
Pow53(k)  :⇔  ( Ax:Ay: k=x*y -> x=1 | Ez: x=53*z )
```

That $x < y$ can be expressed as follows.

```
lt(x, y)  :⇔  ( Ei: y=x+i+1 )
```

Based on these considerations, if `x` and `y` are the numbers of two strings, then the number of the concatenation of the strings is obtained as follows.

```
Cat(x, y, z)  :⇔
      ( Ek: Pow53(k) & lt(y*53+1, 53*k+y) & ~lt(y*53+1, k+y) & z=k*x+y )
```

Atomic propositions in our language on strings are of the form $t_1 \cdots t_m = u_1 \cdots u_n$ or $t_1 \cdots t_m \neq u_1 \cdots u_n$, where $t_1, \ldots, t_m, u_1, \ldots, u_n$ are variables or string literals. They can be replaced as shown below for `=`, where `t` and `u` are two variable names that are different from the t_i and u_j.

```
( Et:Eu: t=u
      & ( Eu: Cat(u, $t_m$, t) & Et: Cat(t, $t_{m-1}$, u) & Eu: Cat(u, $t_{m-2}$, t) & ... )
      & ( Et: Cat(t, $u_n$, u) & Eu: Cat(u, $u_{n-1}$, t) & Et: Cat(t, $u_{n-2}$, u) & ... )
)
```

There is a Turing machine T_1 that inputs a sentence in the language on strings, replaces each string literal by its number, and replaces each atomic proposition as shown above. (We could have made this easier for the Turing machine but harder for the reader by, in Section 3, not allowing more than one character in any string literal, not allowing more than one variable and/or string literal in a term, and instead declaring that `<x+y:z>` expresses that `xy=z`.) If there is a Turing machine T_2 that halts on the true sentences in the language on natural numbers and fails to halt on false sentences, then there is a Turing machine T that first runs T_1 and then runs T_2 on the result. By construction, T halts if and only if its input string is a true sentence in the language on strings. But we proved in Section 6 that such a Turing machine does not exist. Therefore, T_2 does not exist. We have proven the following.

Theorem 2 *Each recursively enumerable proof system for the first-order language on natural numbers with* 0, 1, $+$, $\cdot$, *and* $=$, *either proves a false sentence or fails to prove a true sentence.*

8 Versions of Gödel's First Incompleteness Theorem

We proved that any recursively enumerable theory of natural numbers with zero, one, addition, multiplication, equality, logical and, logical not, and the universal quantifier either proves a false sentence or fails to prove a true sentence. Although this theorem is widely called Gödel's first incompleteness theorem, it falls short of what Gödel presented in [2]. It assumes that the truth or falsehood of a sentence can be reasonably talked about, even if the theory does not prove either. (When we say that a theory proves a sentence false, we mean that the theory proves the negation of the sentence.) This assumption has been criticized. Perhaps for this reason, Gödel went beyond this version. To discuss this, we first make the following observation.

In the presence of truth and falsehood as we usually consider them, a sentence and its negation cannot both be true. Furthermore, for each sentence, either it or its negation is true. A theory is *consistent* if and only if in no case it proves both a sentence and its negation. A theory is *complete* if and only if in each case it proves the sentence or its negation. (The word "complete" is used in more than one meaning in mathematical logic. This is the meaning we use here.) Therefore, if a theory proves only true sentences and proves all of them, then it is consistent and complete.

The notions of consistency and completeness do not rely on a pre-defined notion of truth of a sentence. However, they do not together mean the same as "proves only true sentences and proves all of them", because it may be that the theory fails to prove a true sentence and instead proves its false negation. Indeed, there are consistent and complete theories whose language is the same as the language of natural number arithmetic. An example is obtained by letting 0 denote false, 1 denote true, $+$ denote $\vee$, and $\cdot$ denote $\wedge$, and by adopting the usual axioms and inference rules of $=$ and propositional logic together with two special rules: "$\forall x : P(x)$ is equivalent to $P(0) \cdot P(1)$" and "$\exists x : P(x)$ is equivalent to $P(0) + P(1)$". This theory could well be called a first-order theory of truth values. It proves sentences that are false from the point of view of natural number arithmetic, such as $1+1=1$ (which represents $\mathsf{true} \vee \mathsf{true} = \mathsf{true}$). It is a consistent and complete theory, but a wrong theory for natural number arithmetic although it has the same language.

Another way to look at this is that the replacement of the notions of truth and falsehood by completeness and consistency disconnect the language from natural numbers, leaving only two uninterpreted constant symbols 0 and 1, and two uninterpreted binary operator symbols $+$ and $\cdot$. The mere fact that the symbols look familiar does not give them any formal properties. Instead, to make them again have a link with natural number arithmetic, some axioms and inference rules are needed.

In conclusion, the right liberation of the incompleteness result from the notion of pre-defined truth is that no "sufficiently strong" theory of natural number arithmetic is consistent and complete. Here "sufficient strength" has two aspects. First, the notion of first-order theories has a standard set of logical axioms and inference rules. It is assumed. Second, enough properties of natural numbers are assumed in the form of axioms, to ensure that the theory indeed is a theory of natural numbers instead of, say, the theory of truth values sketched above. Not much is needed. A rather weak axiom system known as *Robinson arithmetic* suffices [10, 12]. It is otherwise the same as the well-known Peano arithmetic, but the induction axiom has been replaced by the axiom "each natural number is either 0 or the result of adding 1 to some natural number."

Gödel did not prove the theorem in the form stated above. Instead of consistency, he used the stronger notion called *ω-consistency*. A theory is not ω-consistent if and only if it is not consistent or, for some formula P with one free variable x, it proves both $\exists x : P(x)$ and each one of $\neg P(0)$, $\neg P(1)$, $\neg P(2)$, ... That a theory proves each one of $\neg P(0)$, $\neg P(1)$, $\neg P(2)$, ... does not necessarily imply that it proves their conjunction $\forall x : \neg P(x)$, because no proof can go through an infinite number of cases one by one (proofs must be finite), and a common pattern that would facilitate proving them simultaneously in a single proof does not necessarily exist. Even so, intuition says that if none of $P(0)$, $P(1)$, $P(2)$, and so on holds, then there is no x such that $P(x)$ holds, that is, $\neg\exists x : P(x)$ holds. So a healthy theory of natural number arithmetic must be not only consistent, but also ω-consistent.

Gödel's result was that such a theory cannot be complete. Let $\mathsf{Prf}(x,y)$ denote the claim that natural number x is the encoding of a proof of the sentence encoded by natural number y. This claim can be formulated in natural number arithmetic. Furthermore, if $\mathsf{Prf}(x,y)$ holds, then $\mathsf{Prf}(x,y)$ can be proven, and if $\neg\mathsf{Prf}(x,y)$ holds, then $\neg\mathsf{Prf}(x,y)$ can be proven. Together with the requirement of "effectivity", this implies that the proof system must be recursive in the sense of Section 2. That is, there is a mechanical test which, for any string, tells whether it is a valid proof, where also the answer "no" is given explicitly instead of just never answering anything. Although this assumption is strictly stronger than recursive enumerability, proof systems typically satisfy it.

The proof system must also facilitate the simple reasoning steps in the sequel.

Gödel's self-referential sentence is $\neg\exists x : \mathsf{Prf}(x,g)$, where g is its own encoding as a natural number. Assume first that the proof system proves $\neg\exists x : \mathsf{Prf}(x,g)$. Then there is a natural number p that is the encoding of some proof of $\neg\exists x : \mathsf{Prf}(x,g)$. By the strength assumption above, the proof system proves $\mathsf{Prf}(p,g)$. From it the proof system can conclude $\exists x : \mathsf{Prf}(x,g)$. So it proves both $\exists x : \mathsf{Prf}(x,g)$ and its negation, and is thus not consistent. The case remains where the proof system does not prove $\neg\exists x : \mathsf{Prf}(x,g)$. Then no natural number is the encoding of a proof of $\neg\exists x : \mathsf{Prf}(x,g)$. By the strength assumption above, the proof system proves $\neg\mathsf{Prf}(0,g)$, $\neg\mathsf{Prf}(1,g)$, and so on. If the proof system is ω-consistent, then it does not prove $\exists x : \mathsf{Prf}(x,g)$. So it leaves both $\exists x : \mathsf{Prf}(x,g)$ and $\neg\exists x : \mathsf{Prf}(x,g)$ without a proof, and is thus incomplete.

Later Rosser found a modification to the proof that allows to replace ω-consistency with consistency [11]. We call his self-referential sentence R. It is $\forall x : (\mathsf{Prf}(x,r) \rightarrow \exists y : y \leq x \wedge \mathsf{Prf}(y,\overline{r}))$, where r is the encoding of R and $\overline{r}$ is the encoding of $\neg$R. If p is the encoding of a proof of R, then the system proves R, $\mathsf{Prf}(p,r)$, and $\exists y : y \leq p \wedge \mathsf{Prf}(y,\overline{r})$. If such an y indeed exists, then the proof whose encoding is y yields $\neg$R, so the system proves a contradiction. Otherwise, the system proves $\neg\mathsf{Prf}(0,\overline{r}) \wedge \neg\mathsf{Prf}(1,\overline{r}) \wedge \cdots \wedge \neg\mathsf{Prf}(p,\overline{r})$, yielding $\neg\exists y : y \leq p \wedge \mathsf{Prf}(y,\overline{r})$, a contradiction again. ω-consistency is not needed, because $\neg\mathsf{Prf}(0,\overline{r}) \wedge \cdots \wedge \neg\mathsf{Prf}(p,\overline{r})$ is a finite expression and thus a sentence.

If p is the encoding of a proof of $\neg$R, then the system proves $\mathsf{Prf}(p,\overline{r})$ and $\exists x : (\mathsf{Prf}(x,r) \wedge \neg\exists y : y \leq x \wedge \mathsf{Prf}(y,\overline{r}))$, yielding $\exists x : \mathsf{Prf}(x,r) \wedge \neg(p \leq x)$. Like above, the system proves one or another contradiction, depending on whether any of 0, 1, ..., $p-1$ is the encoding of a proof of R.

In conclusion, if the system is consistent, then it proves neither R nor $\neg$R, and is thus incomplete.

The above proof of Rosser's theorem uses the symbol $\leq$ that is not part of the first-order language on arithmetic. It can be expressed as mentioned in Section 2. The crucial property is that if c is a natural number constant, then no other natural numbers than 0, 1, ..., c have the properties that $x \leq c$ and $\neg(c+1 \leq x)$. This can be proven from Peano arithmetic, but in the case of other axiom systems, specific axioms on $\leq$ may be needed.

In his original publication [2] Gödel also sketched a proof of a corollary that is now known as Gödel's second incompleteness theorem. It says that natural number arithmetic does not prove its own consistency, if it indeed is consistent. What is more, no recursive consistent theory that contains natural number arithmetic proves its own consistency. The significance of this result is the following. Some mathematical principles are easy to accept, while some others have raised doubts. The questionable principles would become more acceptable, if, with a proof that only uses easily acceptable principles, they were proven to not yield contradictions. Gödel's second incompleteness theorem rules out perhaps not all, but at least the most obvious approaches to such proofs.

9 Related Work and Conclusions

We have shown that any recursively enumerable first-order theory of finite character strings with concatenation and equality either proves a false sentence or fails to prove a true sentence. From this we derived a similar result about natural number arithmetic with addition, multiplication, and equality, obtaining the "either proves a false sentence or fails to prove a true sentence" version of Gödel's first incompleteness theorem.

A *halting tester* is a Turing machine that reads any Turing machine M together with its input I and tells whether M halts, if executed on I. In addition to inventing his machines, Alan Turing proved that there is no halting tester [13]. (The modern version of this proof is very simple.)

Our proof of Gödel's theorem is based on encoding each claim of the form "M halts on I" as a sentence in natural number arithmetic. It is not the first such proof. If each encoded sentence "M halts on I" or its negation were provable by a system that only proves true sentences, then a halting tester would be obtained by letting a Turing machine test all finite character strings until it finds a proof of halting or non-halting. Therefore, the system has an unprovable true sentence. Essentially the same reasoning can be expressed in another words by pointing out that the set of provable sentences is recursively enumerable, but the set of true sentences is not recursively enumerable, because the true sentences of the form "M does not halt on I" cannot be enumerated. So truth and provability do not match. This proof is given in, e.g., [4, p. 354] (leaving the (A) mentioned below as a doubly starred exercise!) and [7, pp. 288–291], underlies the proof in [8, p. 134], and is at least hinted at in [6, p. 64].

Two major difficult technicalities in Gödel's proof are (A) to show that reasoning or computation can be encoded as properties of natural numbers (so-called *Gödel numbering*), and (B) to give a formula access to its own number. The proof based on non-existence of halting testers makes (B) trivial. It makes it necessary to check or believe that, given M and I, a Turing machine can perform the construction in (A). Fortunately, it is rather obvious.

Because natural number arithmetic has no direct construct for expressing finite sequences of natural numbers, (A) is surprisingly difficult. To do (A), often the Chinese Remainder Theorem is used. That brings discussion so far from the main topic that some expositions simply skip the issue. Alternatively, one may add the exponentiation operator n^m to the theory, as was done in [8, p. 135]. In [7], Dexter Kozen made (A) relatively easy by using a slightly less straighforward representation for halting computations than we did in Section 5, and treating natural numbers essentially as finite sequences of p-ary digits, where p is a prime. Numbers that were known to be powers of p, but not known which power, were used to extract individual digits. Thanks to padding with blanks, the representation of each configuration during a computation used the same number of digits. As a consequence, there was a number c such that if y extracts a digit in a configuration, then yc extracts the corresponding digit in the next configuration.

Our proof made (A) and (B) easy by doing them in a formalism that is very amenable to them. The most advanced number-theoretic property needed in the whole proof is that if a number is a power of prime p, then all its factors other than 1 are divisible by p. The Chinese Remainder Theorem was not used and the exponentiation operator was not added to the language. Turing machines were referred to twice: as the basis of the definition of "recursively enumerable proof system", and as devices that can perform a simple syntactic transformation.

It seems obvious that the incompleteness of theories of finite character strings can also be proven with the approach in [4, 7, 8]. Then one may continue like in Section 7. In this combined approach, the formula $\mathtt{Q(x,\ y)}$ would not be needed (but the computability of (A) by a Turing machine would).

Neil D. Jones has proven the incompleteness of first-order theories of nested lists with concatenation [5, p. 202]. Also this proof is based on the non-existence of halting testers. The counterpart of (A) is trivial, because the formalism supports it directly. Nested lists are a strong formalism that can easily express natural numbers, so this result is not surprising.

Finite character strings with concatenation may at first sight seem a poor formalism: a data type with infinitely many distinct values could not be much simpler. On the other hand, all computation reduces to the manipulation of finite character strings both in theory (Turing machines) and in practice (files are finite sequences of bytes). So the incompleteness of first-order theories of finite character strings with concatenation seems too obvious to be a new result. To prove it, it suffices to cite Gödel and then show that strings can simulate arithmetic. Such simulations have been studied at least in [1, 9]. However, neither publication explicitly mentions the incompleteness of strings and, indeed, the author has failed to find any mention of it in the literature. What is more, in this paper the proof was simplified by simulating in the opposite direction, that is, by proving the incompleteness of strings directly and then deriving the incompleteness of arithmetic as a corollary. This idea seems to be new.

Even if it turns out that the approach of this paper is not novel, we hope that the paper helps the readers understand Gödel's famous result.

Acknowledgements. This version of the paper has benefited from the good comments given by the anonymous reviewers.

References

[1] J. Corcoran, W. Frank & M. Maloney (1974): *String Theory*. *J. Symb. Log.* 39(4), pp. 625–637, doi:10.2307/2272846.

[2] K. Gödel (1931): *Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I*. *Monatsh. Math. Phys.* 38(1), pp. 173–198, doi:10.1007/BF01700692. In German.

[3] D.R. Hofstadter (1979): *Gödel, Escher, Bach: An Eternal Golden Braid*. Basic Books.

[4] J.E. Hopcroft & J.D. Ullman (1979): *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley.

[5] N.D. Jones (1997): *Computability and Complexity – From a Programming Perspective*. Foundations of computing series, MIT Press.

[6] S.C. Kleene (1943): *Recursive Predicates and Quantifiers*. *Trans. Amer. Math. Soc.* 53(1), pp. 41–73, doi:10.1090/S0002-9947-1943-0007371-8.

[7] D. Kozen (1997): *Automata and Computability*. Undergraduate texts in computer science, Springer, doi:10.1007/978-1-4612-1844-9.

[8] C.H. Papadimitriou (1994): *Computational Complexity*. Addison-Wesley.

[9] W.V. Quine (1946): *Concatenation as a Basis for Arithmetic*. *J. Symb. Log.* 11(4), pp. 105–114, doi:10.2307/2268308. Available at http://projecteuclid.org/euclid.jsl/1183395170.

[10] R.M. Robinson (1950): *An Essentially Undecidable Axiom System*. In: *Proceedings of the International Congress of Mathematics 1950*, pp. 729–730.

[11] J.B. Rosser (1936): *Extensions of Some Theorems of Gödel and Church*. *J. Symb. Log.* 1(3), pp. 87–91, doi:10.2307/2269028. Available at http://projecteuclid.org/euclid.jsl/1183142131.

[12] A. Tarski, A. Mostowski & R.M. Robinson (1953): *Undecidable Theories*. North Holland.

[13] A. Turing (1936): *On Computable Numbers, With an Application to the Entscheidungsproblem*. *Proceedings of the London Mathematical Society* 42, pp. 230–265. Correction in 43 (1937), 544–546.

Subset Synchronization of Transitive Automata[*]

Vojtěch Vorel
Charles University in Prague, Czech Republic
vorel@ktiml.mff.cuni.cz

We consider the following generalized notion of synchronization: A word is called a reset word of a subset of states of a deterministic finite automaton if it maps all states of the set to a unique state. It is known that the minimum length of such words is superpolynomial in worst cases, namely in a series of substantially nontransitive automata. We present a series of transitive binary automata with a strongly exponential minimum length. This also constitutes a progress in the research of composition sequences initiated by Arto Salomaa, because reset words of subsets are just a special case of composition sequences. Deciding about the existence of a reset word for given automaton and subset is known to be a PSPACE-complete problem, we prove that this holds even if we restrict the problem to transitive binary automata.

1 Introduction

A *deterministic finite automaton* is a triple $A = (Q, X, \delta)$, where Q and X are finite sets and δ is an arbitrary mapping $Q \times X \to Q$. Elements of Q are called *states*, X is the *alphabet*. The *transition function* δ can be naturally extended to $Q \times X^{\star} \to Q$, still denoted by δ. We extend it also by defining

$$\delta(S,w) = \{\delta(s,w) \mid s \in S, w \in X^{\star}\}$$

for each $S \subseteq Q$. An automaton (Q, X, δ) is said to be *transitive* if

$$(\forall r, s \in Q)\,(\exists w \in X^{\star})\,\delta(r,w) = s.$$

A state $s \in Q$ is a *sink state* if

$$(\forall x \in X)\,\delta(s,x) = s.$$

Clearly, if a nontrivial automaton has some sink state, it is impossible for the automaton to be transitive. For a given automaton $A = (Q, X, \delta)$, we call $w \in X^{\star}$ a *reset word* if $|\delta(Q,w)| = 1$. If such a word exists, we call the automaton *synchronizing*. Note that each word having a reset word as a factor is also a reset word.

The *Černý conjecture*, a longstanding open problem, claims that each synchronizing automaton has a reset word of length at most $(|Q|-1)^2$. There is a series of automata due to Černý that reaches this bound [3], but all known upper bounds lie in $\Omega\left(|Q|^3\right)$, see [15] for the best one[1]. A tight bound has been established for various special classes of automata, see a survey in [23] or some recent advances e.g. in [1, 6, 8, 20].

[*]Research supported by the Czech Science Foundation grant GA14-10799S.
[1]An improved bound published by Trakhtman [22] in 2011 has turned out to be proved incorrectly.

1.1 Synchronization of Subsets

Even if an automaton is not synchronizing, there could be various subsets $S \subseteq Q$ such that $|\delta(S,w)| = 1$ for some word $w \in X^\star$. We say that such S is *synchronizable* in A and in the opposite case we say it is *blind* in A. The word w is called a *reset word of* S in A. Such words are of our interest. They lack some of elegant properties of classical reset words (i.e. reset words of all Q), particularly a word w having a factor v which is a reset word of S need not to be itself a reset word of S. In fact, if we choose a subset S and a word w, it is possible for the set $\delta(S,w)$ to be blind even if the set S was synchronizable.

Suppose $A = (Q,X,\delta)$ and $S \subseteq Q$. We denote by $\mathrm{CS}(A,S)$ the length of the shortest reset word of S in A. If S is blind, we set $\mathrm{CS}(A,S) = 0$. Let $\mathcal{M}$ be a class of automata. For each n let $\mathcal{M}_{\leq n}$ be the class of all automata lying in $\mathcal{M}$ and having at most n states. We denote

$$\mathrm{CS}_n^{\mathcal{M}} = \max_{\substack{A \in \mathcal{M}_{\leq n} \\ S \subseteq Q}} \mathrm{CS}(A,S).$$

If $\mathcal{M}$ is the class of *all* automata, we write just CS_n instead of $\mathrm{CS}_n^{\mathcal{M}}$.

Such values we informally call *subset synchronization thresholds.* The class of all transitive automata and the class of all automata with a k-letter alphabet are denoted by $\mathcal{TR}$ and $\mathcal{AL}_k$ respectively. Automata from $\mathcal{AL}_2$ are called *binary*.

As we describe below, it was proven independently by [10] and [17] that $\mathrm{CS}_n \geq (\sqrt[3]{n})!$, and a construction from [12] implies that $\mathrm{CS}_n \geq 2^{\Omega(n)}$, but the proofs use automata with multiple sink states and growing alphabets. Use of sink states is a very strong tool for designing automata having given properties, but in practice such automata seem very special. They represent unstable systems balancing between different deadlocks. The very opposite are the transitive automata. Does the threshold remain so high if we consider only transitive automata? Unfortunately, we show below that it does, even if we restrict the alphabet size to a constant. We prove that

$$\mathrm{CS}_n^{\mathcal{AL}_2 \cap \mathcal{TR}} = 2^{\Omega(n)},$$

which substantially raises also the general lower bounds of each $\mathrm{CS}_n^{\mathcal{AL}_k}$, because their former lower bounds (following from [12]) lie in $2^{o(n)}$. The new bound is tight since $\mathrm{CS}_n = 2^{\mathcal{O}(n)}$.

1.2 Minimum Length of Compositions

It has been repeatedly pointed out by Arto Salomaa [17, 18] that very little is known about minimum length of a composition needed to generate a function by a given set of generators. To be more precise, let us adopt and slightly extend the notation used in [17, 18]. We denote by $\mathcal{T}_n$ the semigroup of all functions from $\{1,\ldots,n\}$ to itself. Given $\mathbf{G} \subseteq \mathcal{T}_n$, we denote by $\langle \mathbf{G} \rangle$ the subsemigroup generated by $\mathbf{G}$. Given $\mathbf{F} \subseteq \mathcal{T}_n$ we denote by $\mathrm{D}(\mathbf{G},\mathbf{F})$ the length k of a shortest sequence $g_1,\ldots,g_k$ of functions from $\mathbf{G}$ such that $g_1 \ldots g_k \in \mathbf{F}$. Finally, denote

$$\mathrm{D}_n = \max_{\overline{n} \leq n} \max_{\substack{\mathbf{F},\mathbf{G} \subseteq \mathcal{T}_{\overline{\mathbf{n}}} \\ \mathbf{F} \cap \langle \mathbf{G} \rangle \neq \emptyset}} \mathrm{D}(\mathbf{G},\mathbf{F}). \tag{1}$$

From the well-known connection between automata and transformation semigroups it follows that the value CS_n could be also defined by (1) if we just restrict $\mathbf{F}$ to be some of the sets

$$\mathbf{F}_S = \{ f \in \mathcal{T}_n \mid (\forall r,s \in S)\, f(r) = f(s) \}$$

for $S \subseteq \{1, \ldots, n\}$. Therefore it holds trivially that

$$\mathrm{D}_n \geq \mathrm{CS}_n .$$

Arto Salomaa refers to a single nontrivial bound of D_n, namely $\mathrm{D}_n \geq (\sqrt[3]{n})!$, which is a consequence of the above-mentioned variant for CS_n. In fact he omits a much older construction of Kozen [9, Theorem 3.2.7] which deals with lengths of *proofs* rather than compositions but witnesses easily that $\mathrm{D}_n = 2^{\Omega\left(\frac{n}{\log n}\right)}$. Since 2013 it follows from [12] that $\mathrm{D}_n = 2^{\Omega(n)}$. Our result shows that this lower bound holds also if we restrict $\mathbf{G}$ to any nontrivial fixed size.

In Group Theory, thresholds like D_n are studied in the scope of permutations, see [7].

2 Lower Bounds of Subset Synchronization Thresholds

We first formulate the two former lower bounds of CS_n. Let p_i stand for the i-th prime.

Theorem 1 ([10, 17]). *For each k there is an automaton $A_k = (Q_k, \{a,b\}, \delta_k)$ and a subset S_k such that $|Q_k| = 2 + \sum_{i=1}^{k} p_i$ and $\mathrm{CS}(A_k, S_k) = \prod_{i=1}^{k} p_i$.*

The proof of the theorem uses an automaton A_k that consists of k cyclic parts of prime sizes and two sink states, so it is essentially non-transitive. The theorem implies that $\mathrm{CS}_n \geq (\sqrt[3]{n})!$, because $\prod_{i=1}^{k} p_i \geq k!$ and $|Q_k| \leq k^3$, using the estimation $p_i \leq i^2$. By the terminology of [10] such bound is *exponential*, but using canonical estimations of p_i it is not hard to show that the bound is exceeded by $n \mapsto \varepsilon^n$ for any $\varepsilon > 1$.

Theorem 2 ([12]). *It holds that $\mathrm{CS}_n = 2^{\Omega(n)}$ and $\mathrm{CS}_n^{\mathcal{AL}_2} = 2^{\Omega\left(\frac{n}{\log n}\right)}$.*

The paper [12] studies *careful synchronization* of partial automata, but the lower bounds can be adapted for our setting. The proofs of [12, Theorem 1] and [12, Theorem 3] can be modified (by adding one state) so that the constructed automata have sink states. Then we can add another sink state D which becomes the target of all undefined transitions. Then all reset words for the subset $Q \backslash \{\mathrm{D}\}$ are careful reset words of the original partial automaton and we can use the corresponding lower bounds.

Let us introduce three key methods used in the present paper. The first is quite simple and has been already used in the literature [2]. It modifies an automaton in order to decrease the alphabet size with preserving high synchronization thresholds:

Lemma 3. *For each automaton $A = (Q, X, \delta)$ and $S \subseteq Q$ there is an automaton $A' = (Q', X', \delta')$ and $S' \subseteq Q'$ such that*

1. *S is synchronizable in $A \Rightarrow S'$ is synchronizable in A'*
2. $\mathrm{CS}(A', S') \geq \mathrm{CS}(A, S)$
3. $|Q'| = |Q| \cdot |X|$
4. $|X'| = 2$
5. *A' and A have equal number of strongly connected components*

Proof. Suppose that $X = \{a_0, \ldots, a_m\}$. We set $Q' = Q \times X$, $X' = \{\alpha, \beta\}$,

$$\begin{aligned} \delta'((s,a_i),\alpha) &= (\delta(s,a_i), a_0) \\ \delta'((s,a_i),\beta) &= (s, a_{i+1 \bmod m}) . \end{aligned}$$

Informally, a transition $r \xrightarrow{a_i} s$ of A is simulated in A' by

$$(r,a_0) \xrightarrow{\beta} (r,a_1) \xrightarrow{\beta} \ldots \xrightarrow{\beta} (r,a_i) \xrightarrow{\alpha} (s,a_0)\,.$$

If we set $S' = S \times \{a_0\}$, it is not hard to see that any reset word of S' in A' have to be of the form $\left(\beta^{i_1}\alpha\right)\ldots\left(\beta^{i_d}\alpha\right)$ for some $w = a_{i_1}\ldots a_{i_d}$ which is a reset word of S in A. □

The second method is original and is intended for modifying an automaton to be transitive, again with high synchronization thresholds preserved. It relies on the following concept:

Definition 4. Let $A = (Q,X,\delta)$ be an automaton and let $\rho \subseteq Q^2$ be a congruence, i.e. equivalence relation satisfying $r\rho s \Rightarrow \delta(r,x)\,\rho\,\delta(s,x)$ for each $x \in X$. We say that ρ is a *swap congruence* if, for each equivalence class C of ρ and each letter $x \in X$, the restricted function $\delta(\_,x) : C \to Q$ is injective.

Let us express the key feature of swap congruences and use it in the construction.

Lemma 5. *Let $A = (Q,X,\delta)$ be an automaton, let $\rho \subseteq Q^2$ be a swap congruence and take any $S \subseteq Q$. If there are any $r,s \in S$ with $r \neq s$ and $r\rho s$, the set S is blind.*

Proof. Because r and s lie in a common equivalence class of ρ, by the definition of swap congruence we have $\delta(r,x) \neq \delta(s,x)$ for any $x \in X$. It follows that each set $\delta(S,w)$ for $w \in X^\star$ is of size at least 2. □

Lemma 6. *For each automaton $A = (Q,X,\delta)$ and $S \subseteq Q$ there is an automaton $A' = (Q',X',\delta')$ and $S' \subseteq Q'$ such that*

1. *S is synchronizable in $A \Rightarrow S'$ is synchronizable in A'*
2. $\mathrm{CS}(A',S') \geq \mathrm{CS}(A,S)$
3. *A' is transitive*
4. $|Q'| = 4|Q| + 2$
5. $|X'| = |X| + 2c$

where c is the number of strongly connected components of A.

Proof. Let $C_1,\ldots,C_c$ be the strongly connected components of A. Fix some $q_i \in C_i$ for each i. We set

$$\begin{aligned} Q' &= \left\{E,\overline{E}\right\} \cup \left(\left\{1,\overline{1},2,\overline{2}\right\} \times Q\right) \\ X' &= X \cup \{a_i, b_i \mid i = 1,\ldots,c\} \end{aligned}$$

and define the transition function δ' as follows. If we omit all the letters a_s and b_s from the alphabet X', we find the states $E,\overline{E}$ isolated (i.e. $E \xrightarrow{x} E, \overline{E} \xrightarrow{x} \overline{E}$ for $x \in X$) and the rest of A' consisting just of four copies of A:

$$(N,s) \quad \xrightarrow{x} \quad (N,\delta(s,x))$$

for $N \in \left\{1,\overline{1},2,\overline{2}\right\}, s \in Q, x \in X$. Let us introduce the additional letters. For any $i \in 1,\ldots,c$ and $s \in Q$ such that $s \neq q_i$ we set

$$\begin{array}{ll} (1,s) \xrightarrow{a_i,b_i} E & \left(\overline{1},s\right) \xrightarrow{a_i,b_i} \overline{E} \\ (2,s) \xrightarrow{a_i,b_i} E & \left(\overline{2},s\right) \xrightarrow{a_i,b_i} \overline{E} \end{array}$$

and it remains to see Figure 1, which describes for each $i \in 1,\dots,c$ the action of a_i and b_i on the six states $E, \overline{E}, (1,q_i), (\overline{1},q_i), (2,q_i), (\overline{2},q_i)$.

Observe that the equivalence ρ having the classes $\{E,\overline{E}\}$ and $\{(1,s),(\overline{1},s)\}, \{(2,s),(\overline{2},s)\}$ for each $s \in Q$ is a swap congruence of A'. We claim that the automaton A' and the set

$$S' = \{1\} \times S \cup \{\overline{2}\} \times S$$

fulfill our requirements on synchronizability, the synchronization threshold and transitivity:

- If the set S is synchronizable in A, there are $r \in Q$ and $w \in X^\star$ such that $\delta(S,w) = \{r\}$. The state r lies in some C_i, so there is a word $u \in X^\star$ such that $\delta(r,u) = q_i$. We claim that the word wua_i synchronizes S' in A'. Indeed, we have $\delta'(S',w) = \{(1,r),(\overline{2},r)\}$ and therefore $\delta'(S',wu) = \{(1,q_i),(\overline{2},q_i)\}$ and $\delta'(S',wua_i) = \{(1,q_i)\}$.
- Let S' be synchronized in A' by a word $w \in (X')^\star$. There must occur some letter a_r or b_r in w, because S' contains states from two different copies of A. Thus we can write

$$w = uxv$$

for some $u \in X^\star$, $i \in 1,\dots,c$, $x \in \{a_i,b_i\}$ and $v \in (X')^\star$. If $\delta'(S',u)$ contains unique state from $\{1\} \times Q$, the word u synchronizes S in A, we are done. Otherwise there is some state $(1,s) \in \delta'(S',u)$ such that $s \neq q_i$. Because $u \in X^\star$, it holds also that $(\overline{2},s) \in \delta'(S',u)$. But

$$(1,s) \xrightarrow{a_i,b_i} E \qquad (\overline{2},s) \xrightarrow{a_i,b_i} \overline{E},$$

so the blind subset $\{E,\overline{E}\}$ is contained in $\delta'(S',ux)$, which is a contradiction.
- In order to verify that A' is transitive we first find a path between any pair of distinct states $(N,s),(N,r)$ from a common copy of A. Let $r \in C_i$ and $\delta(q_i,u) = r$. If $s = q_i$, the path is labeled by u. Otherwise we have:

$$\begin{array}{llll}(1,s) \xrightarrow{a_i} E \xrightarrow{a_i} (1,q_i) \xrightarrow{u} (1,r) & & (\overline{1},s) \xrightarrow{a_i} \overline{E} \xrightarrow{a_i} (\overline{1},q_i) \xrightarrow{u} (\overline{1},r) \\ (2,s) \xrightarrow{a_i} E \xrightarrow{b_i} (2,q_i) \xrightarrow{u} (2,r) & & (\overline{2},s) \xrightarrow{a_i} \overline{E} \xrightarrow{b_i} (\overline{2},q_i) \xrightarrow{u} (\overline{2},r).\end{array}$$

The paths above also guarantee that there are no more than two strongly connected components:

$$C = \{E\} \cup \{1,2\} \times Q, \qquad \overline{C} = \{\overline{E}\} \cup \{\overline{1},\overline{2}\} \times Q.$$

It remains to connect C with $\overline{C}$: For any i we have $(1,q_i) \xrightarrow{b_i} (\overline{2},q_i) \xrightarrow{a_i} (1,q_i)$.

□

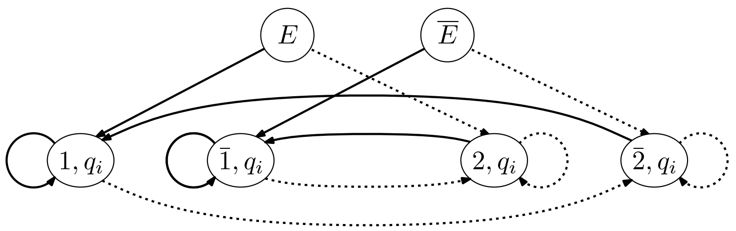


Figure 1: Action of the letters a_i (solid arrows) and b_i (dotted arrows) on certain states of A'.

Let us present the main construction of the present paper, a series of automata with strictly exponential subset synchronization threshold, constant alphabet size and constant number of strongly connected components. We use some informal principles that occur in [12] as well.

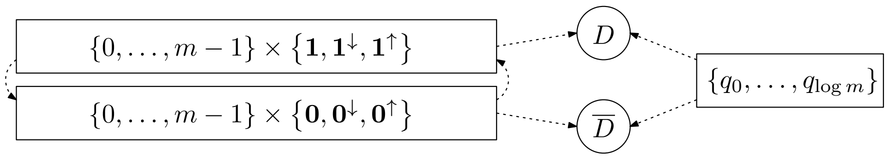

$\{0,\ldots,m-1\}\times\{\mathbf{1},\mathbf{1}^{\downarrow},\mathbf{1}^{\uparrow}\}$
$\{0,\ldots,m-1\}\times\{\mathbf{0},\mathbf{0}^{\downarrow},\mathbf{0}^{\uparrow}\}$
D
$\overline{D}$
$\{q_0,\ldots,q_{\log m}\}$

Figure 2: Main parts of A. The arrows depict the connectivity pattern of A.

Lemma 7. *For infinitely many $m\in\mathbb{N}$ there is an automaton $A=(Q,X,\delta)$ and $S\subseteq Q$ such that*

1. $\mathrm{CS}(A,S)=2^m(\log m+1)+1$
2. $|Q|=6m+\log m+3$
3. $|X|=4$
4. A has 4 strongly connected components.

Proof. Suppose $m=2^k$. For each $t\in 0,\ldots,m-1$ we denote by $\tau=\mathrm{bin}(t)$ the standard k-digit binary representation of t. By a classical result proved in [5] there is a *De Bruijn sequence* $\xi=\xi_0\ldots\xi_{m-1}$ consisting of binary letters $\xi_i\in\{\mathbf{0},\mathbf{1}\}$ such that each word $\tau\in\{\mathbf{0},\mathbf{1}\}^k$ appears exactly once as a cyclic factor of ξ (i.e. it is a factor or begins by a suffix of ξ and continues by a prefix of ξ). Let us fix such ξ. By $\pi(i)$ we denote the number t, whose binary representation $\mathrm{bin}(t)$ starts in ξ from the i-th position. Note that π is a permutation of $\{0,\ldots,m-1\}$. Set

$$\begin{aligned}
Q &= \left(\{0,\ldots,m-1\}\times\left\{\mathbf{0},\mathbf{0}^{\downarrow},\mathbf{0}^{\uparrow},\mathbf{1},\mathbf{1}^{\downarrow},\mathbf{1}^{\uparrow}\right\}\right)\cup\{q_0,\ldots,q_{\log m},D,\overline{D}\}\\
X &= \{\mathbf{0},\mathbf{1},\kappa,\omega\}\\
S &= (\{0,\ldots,m-1\}\times\{\mathbf{0}\})\cup\{q_0,D\}.
\end{aligned}$$

Figure 2 visually distinguishes main parts of the automaton. The states D and $\overline{D}$ are sinks. Together with $D\in S$ it implies that any reset word of S takes the states of S to D and that the state $\overline{D}$ must not become active during the synchronization (i.e. lie in $\delta(S,v)$ for a prefix v of a reset word). The states $\{q_0,\ldots,q_{\log m}\}$ guarantee that any reset word of S lies in

$$\left(\{\mathbf{0},\mathbf{1}\}^k\kappa\right)^{\star}\omega X^{\star}. \tag{2}$$

Indeed, as defined by Figure 3, any other word takes q_0 to $\overline{D}$. Let the letter ω act as follows:

$$\begin{aligned}
\{0,\ldots,m-1\}\times\{\mathbf{1}\}\ ,\ q_0\ ,\ D &\overset{\omega}{\longrightarrow} D\\
\{0,\ldots,m-1\}\times\left\{\mathbf{0},\mathbf{0}^{\downarrow},\mathbf{0}^{\uparrow},\mathbf{1}^{\downarrow},\mathbf{1}^{\uparrow}\right\}\ ,\ q_1,\ldots,q_{\log m}\ ,\ \overline{D} &\overset{\omega}{\longrightarrow} \overline{D}.
\end{aligned}$$

We see that ω maps each state to D or $\overline{D}$. This implies that once ω occurs in a reset word of S, it must complete the synchronization. In order to map q_0 to D, the letter ω *must* occur, so any shortest reset word of S is exactly of the form

$$w=(\tau_1\kappa)\ldots(\tau_d\kappa)\,\omega, \tag{3}$$

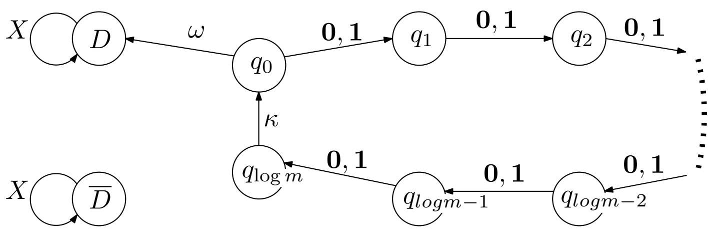


Figure 3: A part of A. Except for the depicted ones, all transitions here lead to $\overline{D}$.

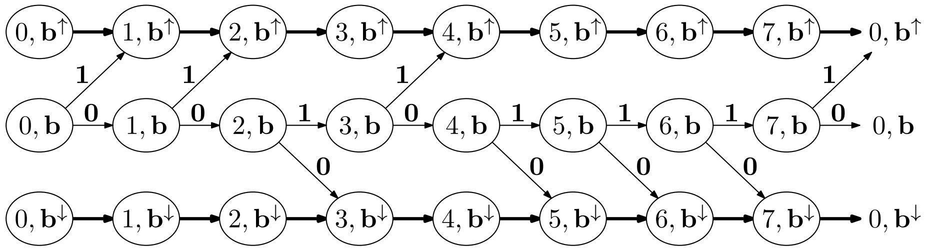
where $\tau_j \in \{\mathbf{0},\mathbf{1}\}^k$ for each j.

The two biggest parts depicted by Figure 2 contain $3m$ states each and are the same up to the letters κ and ω. On both of them (take $\mathbf{b} \in \{\mathbf{0},\mathbf{1}\}$) let the letters $\mathbf{0}$ and $\mathbf{1}$ act as follows:

$$(i,\mathbf{b}) \xrightarrow{\mathbf{0}} \begin{cases} (i+1,\mathbf{b}) & \text{if } \xi_i = \mathbf{0} \\ (i+1,\mathbf{b}^{\downarrow}) & \text{if } \xi_i = \mathbf{1} \end{cases} \qquad (i,\mathbf{b}) \xrightarrow{\mathbf{1}} \begin{cases} (i+1,\mathbf{b}^{\uparrow}) & \text{if } \xi_i = \mathbf{0} \\ (i+1,\mathbf{b}) & \text{if } \xi_i = \mathbf{1} \end{cases}$$

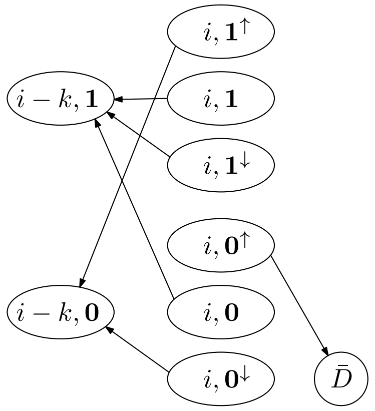
$$\left(i,\mathbf{b}^{\uparrow}\right) \xrightarrow{\mathbf{0},\mathbf{1}} \left(i+1,\mathbf{b}^{\uparrow}\right) \qquad \left(i,\mathbf{b}^{\downarrow}\right) \xrightarrow{\mathbf{0},\mathbf{1}} \left(i+1,\mathbf{b}^{\downarrow}\right)$$

where we perform the addition modulo m. For example, Figure 4 depicts such part of A for $m = 8$ and a particular De Bruijn sequence ξ. Figure 5 defines the action of κ on the states $\{i\} \times \left\{\mathbf{0},\mathbf{0}^{\downarrow},\mathbf{0}^{\uparrow},\mathbf{1},\mathbf{1}^{\downarrow},\mathbf{1}^{\uparrow}\right\}$ for any i, so the automaton A is completely defined.



$0,\mathbf{b}^{\uparrow}$
$1,\mathbf{b}^{\uparrow}$
$2,\mathbf{b}^{\uparrow}$
$3,\mathbf{b}^{\uparrow}$
$4,\mathbf{b}^{\uparrow}$
$5,\mathbf{b}^{\uparrow}$
$6,\mathbf{b}^{\uparrow}$
$7,\mathbf{b}^{\uparrow}$
$0,\mathbf{b}^{\uparrow}$
$\mathbf{1}$
$\mathbf{1}$
$\mathbf{1}$
$\mathbf{1}$
$0,\mathbf{b}$
$\mathbf{0}$
$1,\mathbf{b}$
$\mathbf{0}$
$2,\mathbf{b}$
$\mathbf{1}$
$3,\mathbf{b}$
$\mathbf{0}$
$4,\mathbf{b}$
$\mathbf{1}$
$5,\mathbf{b}$
$\mathbf{1}$
$6,\mathbf{b}$
$\mathbf{1}$
$7,\mathbf{b}$
$\mathbf{0}$
$0,\mathbf{b}$
$\mathbf{0}$
$\mathbf{0}$
$\mathbf{0}$
$\mathbf{0}$
$0,\mathbf{b}^{\downarrow}$
$1,\mathbf{b}^{\downarrow}$
$2,\mathbf{b}^{\downarrow}$
$3,\mathbf{b}^{\downarrow}$
$4,\mathbf{b}^{\downarrow}$
$5,\mathbf{b}^{\downarrow}$
$6,\mathbf{b}^{\downarrow}$
$7,\mathbf{b}^{\downarrow}$
$0,\mathbf{b}^{\downarrow}$

Figure 4: A part of A assuming $m = 8$ and $\xi = \mathbf{00101110}$. Bold arrows represent both $\mathbf{0},\mathbf{1}$.



$i,\mathbf{1}^{\uparrow}$
$i-k,\mathbf{1}$
$i,\mathbf{1}$
$i,\mathbf{1}^{\downarrow}$
$i,\mathbf{0}^{\uparrow}$
$i-k,\mathbf{0}$
$i,\mathbf{0}$
$i,\mathbf{0}^{\downarrow}$
$\bar{D}$

Figure 5: Action of the letter κ. The subtraction is modulo m.

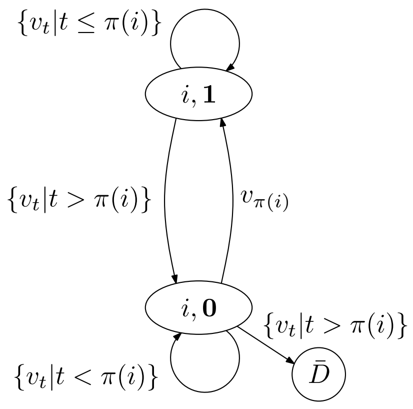


Figure 6: Action of the words $v_0,\ldots,v_{m-1}$ on the i-th switch.

Let w be a shortest reset word of S in A. It is necessarily of the form (3), so it makes sense to denote $v_t = \mathrm{bin}(t)\,\kappa$ and treat w as a word

$$w = v_{t_1} \dots v_{t_d}\, \omega \in \{v_0, \dots, v_{m-1}, \omega\}^\star . \tag{4}$$

The action of each v_t is depicted by Figure 6. It is a key step of the entire proof to confirm that Figure 6 is correct. Indeed:

- Starting from a state $(i, \mathbf{0})$, a word $\mathrm{bin}(t)$ takes us through kind of decision tree to one of the states $\left(i+k, \mathbf{0}^{\downarrow}\right), (i+k, \mathbf{0}), \left(i+k, \mathbf{0}^{\uparrow}\right)$, depending on whether t is lesser, equal, or greater than $\pi(i)$ respectively. This is guaranteed by wiring the sequence ξ into the transition function, see Figure 4. The letter κ then take us back to $\{i\} \times \{\dots\}$, namely to $(i, \mathbf{0})$ or $(i, \mathbf{1})$, or we fall to $\overline{D}$ (respectively).
- Starting from a state $(i, \mathbf{1})$, we proceed similarly and end up in $(i, \mathbf{0})$ or $(i, \mathbf{1})$ depending on whether t is greater than $\pi(i)$ or not.

It follows that after applying any prefix $v_{t_1} \dots v_{t_j}$ of w exactly one of the states $(i, \mathbf{0}), (i, \mathbf{1})$ is active for each i. We say that *the i-th switch is set to* $\mathbf{0}$ *or* $\mathbf{1}$ *in time* j. Observe that in time d all the switches are set to $\mathbf{1}$, because otherwise the state $\overline{D}$ would become active by the application of ω. On the other hand, in time 0 all the switches are set to $\mathbf{0}$. We are going to show that in fact during the synchronization of S the switches together perform a binary counting from 0 (all the switches set to $\mathbf{0}$) to $2^m - 1$ (all the switches set to $\mathbf{1}$). For each i the significance of i-th switch is given by the value $\pi(i)$. So the $\pi^{-1}(m-1)$-th switch carries the most significant digit, the $\pi^{-1}(0)$-th switch carries the least significant digit and so on. The number represented in this manner by the switches in time j is denoted by $\mathfrak{b}_j \in \{0, \dots, 2^m - 1\}$. We claim that $\mathfrak{b}_j = j$ for each j. Indeed:

- In time 0, all the switches are set to $\mathbf{0}$, we have $\mathfrak{b}_0 = 0$.
- Suppose that $\mathfrak{b}_{j'} = j'$ for each $j' \leq j-1$. We denote

  $$\overline{t_j} = \min\{\pi(i) \mid i\text{-th switch is set to } \mathbf{0} \text{ in time } j-1\} \tag{5}$$

  and claim that $t_j = \overline{t_j}$. Note that $\overline{t_j}$ is defined to be the least significance level at which there occurs a $\mathbf{0}$ in the binary representation of $\mathfrak{b}_{j-1}$. Suppose for a contradiction that $t_j > \overline{t_j}$. By the definition of $\overline{t_j}$ the state $\left(\pi^{-1}(\overline{t_j}), \mathbf{0}\right)$ lies in $\delta\left(S, v_{t_1} \dots v_{t_{j-1}}\right)$. But v_{t_j} takes this state to $\overline{D}$, which is a contradiction. Now suppose that $t_j < \overline{t_j}$. In such case the application of v_{t_j} does not turn any switch from $\mathbf{0}$ to $\mathbf{1}$, so $\mathfrak{b}_j \leq \mathfrak{b}_{j-1}$ and thus in time j the configuration of switches is the same at it was in time $\mathfrak{b}_j$. This contradicts the assumption that w is a shortest reset word. We have proved that $t_j = \overline{t_j}$ and it remains only to show that the application of v_{t_j} performs an addition of 1 and so makes the switches represent the value $\mathfrak{b}_{j-1} + 1$.

  - Consider an i-th switch with $\pi(i) < t_j$. By the definition of $\overline{t_j}$ it is set to $\mathbf{1}$ in time $j-1$ and the word v_{t_j} set it to $\mathbf{0}$ in time j. This is what we need because such switches represent a continuous segment of $\mathbf{1}$s at the least significant positions of the binary representation of $\mathfrak{b}_{j-1}$.
  - The $\pi^{-1}(t_j)$-th switch is set from $\mathbf{0}$ to $\mathbf{1}$ by the word v_{t_j}.
  - Consider an i-th switch with $\pi(i) > t_j$. The switch represents a digit of $\mathfrak{b}_{j-1}$ which is more significant than the $\overline{t_j}$-th digit. As we expect, the word v_{t_j} leave such switch unchanged.

Because $\mathfrak{b}_d = 2^m$, we deduce that $d = 2^m$ and thus $|w| = 2^m(\log m + 1) + 1$ if such (shortest) reset word exists. But in fact we have also shown that there is only one possibility for such w and that it is a true reset word for S: The unique w is of the form (4), where t_j is the position of the least significant $\mathbf{0}$ in the binary representation of $j - 1$. □

Now it remains to put the three lemmas together and so construct a binary transitive automaton with a strictly exponential subset synchronization threshold.

Theorem 8. *It holds that* $\mathrm{CS}_n^{\mathcal{TR}\cap\mathcal{AL}_2} = 2^{\Omega(n)}$.

Proof. The series CS_n is non-decreasing, so it is enough to work with some infinitely many values of n. Let us take any $m \in \mathbb{N}$ and use it to build the automaton $A = (Q, X, \delta)$ and the subset S as described by Lemma 7. We apply Lemma 6 to get transitive $A' = (Q', X', \delta')$ and S' with

$$\begin{aligned} \left|Q'\right| &= 24m + 4\log m + 14 \\ \left|X'\right| &= 12 \\ \mathrm{CS}\left(A', S'\right) &\geq 2^m \end{aligned}$$

and then apply Lemma 3 to get transitive $A'' = (Q'', X'', \delta'')$ and S'' with

$$\begin{aligned} \left|Q''\right| &= 288m + 48\log m + 168 \\ \left|X'\right| &= 2 \\ \mathrm{CS}\left(A'', S''\right) &\geq 2^m \end{aligned}$$

Denoting $n = 288m + 48\log m + 168$ we get that

$$\mathrm{CS}_n^{\mathcal{TR}\cap\mathcal{AL}_2} = \Omega\left(2^{\frac{n}{289}}\right).$$

□

Simpler variants of the constructions imply some more subtle results for less restricted classes:

Theorem 9. *It holds that*

1. $\mathrm{CS}_n = \Omega\left(2^{\frac{n}{2}}\right)$
2. $\mathrm{CS}_n^{\mathcal{TR}} = \Omega\left(2^{\frac{n}{4}}\right)$
3. $\mathrm{CS}_n^{\mathcal{AL}_4} = \Omega\left(2^{\frac{n}{7}}\right)$
4. $\mathrm{CS}_n^{\mathcal{AL}_2} = \Omega\left(2^{\frac{n}{25}}\right)$

A series witnessing the first claim arises from the proof of Lemma 7 if we just consider actual alphabet consisting of the letters $\{\omega, v_0, \ldots, v_{m-1}\}$ and realize the idea of Figure 6 so there remain only the states $D, \overline{D}$ and $(i, \mathbf{0}), (i, \mathbf{1})$ for each i. There is no more need to deal with a De Bruijn sequence. The construction presented in Lemma 7 results from an effort to make this simple variant binary with keeping the size of Q in $\mathcal{O}(m)$. A construction needed to prove the second claim depends on a careful use of swap congruences and appears in the extended version of this paper. The third claim follows directly from Lemma 7 and the last one we get if we then just apply Lemma 3.

3 Deciding about Synchronizability

It is well known that the decision about classical synchronizability of a given automaton (i.e. assuming $S = Q$) is a polynomial time task, even if we also require an explicit reset word on the output. A relatively simple algorithm could be traced back to [3] and since that time a lot of work has been done on various improvements. Besides decreasing the running time of the algorithm there is an effort to decrease the length of reset words produced [16, 21]. It has been proven that it is both NP-hard and coNP-hard to find a *shortest* reset word for given automaton (it is actually DP-complete [14]). Moreover, it remains NP-hard to bound the length of shortest reset words only from above by a given value [4] or approximate its length with a constant factor [2]. Such problems has been studied also with various additional requirements on the automaton, e.g. cyclicity, Eulerian property, commutativity and others, but in most cases also the restricted problem turns up to be hard, see [11, 24].

On the other hand, there have not been done much research in computational complexity of problems concerning synchronization of subsets, although they does not seem to have less chance to emerge in practice. Namely, the first natural problem in this direction is

SUBSYNITY
Input: n-state automaton $A = (Q, X, \delta)$, $S \subseteq Q$
Output: is there some $w \in X^{\star}$ such that $|\delta(S, w)| = 1$?

This problem, in contrast to the similar problem of classical synchronizability, is known to be PSPACE-complete. Note that such hardness is not a consequence of any lower bound of synchronization threshold, because an algorithm need not to produce an explicit reset word.

Theorem 10 ([13, 19])**.** SUBSYNITY *is a PSPACE-complete problem.*

The proofs of the theorem above make use of a result of Kozen [9], which establishes that it is PSPACE-complete to decide if given finite acceptors with a common alphabet accept any common word. This problem is polynomially reduced to SUBSYNITY using the idea of two sink states, which is used also in the automata with prime-length cycles and in Lemma 7. Is it possible to avoid the non-transitivity here? We have proved that the subset synchronization threshold may be exponential even in automata from $\mathcal{AL}_2 \cap \mathcal{TR}$, but this does not imply that there is no trick for tractable decision about their synchronizability. However, the methods we used are general enough to reduce SUBSYNITY to the restricted version:

Theorem 11. SUBSYNITY RESTRICTED TO BINARY TRANSITIVE AUTOMATA *is a PSPACE-complete problem.*

Proof. There is a polynomial reduction from the general problem SUBSYNITY: Perform the construction from Lemma 6 and then the one from Lemma 3. □

Though the results of this paper does not sound very optimistically, there are still many interesting and practical restrictions which could hypothetically make our decision problem tractable or at least decrease the subset synchronization threshold, preferably to a polynomial. Such restrictions, which all have been already studied in terms of classical synchronization, concern monotonic and aperiodic automata, cyclic and one-cluster automata, Eulerian automata and others.

References

[1] Marie-Pierre Béal, Mikhail V. Berlinkov, and Dominique Perrin. A quadratic upper bound on the size of a synchronizing word in one-cluster automata. *Int. J. Found. Comput. Sci.*, 22(2):277–288, 2011. doi:10.1142/S0129054111008039.

[2] Mikhail V. Berlinkov. Approximating the minimum length of synchronizing words is hard. *Theory of Computing Systems*, 54(2):211–223, 2014. doi:10.1007/s00224-013-9511-y.

[3] Ján Černý. Poznámka k homogénnym experimentom s konečnými automatmi. *Matematicko-fyzikálny časopis*, 14(3):208–216, 1964.

[4] David Eppstein. Reset sequences for monotonic automata. *SIAM J. Comput.*, 19(3):500–510, 1990. doi:10.1137/0219033.

[5] Camille Flye Sainte-Marie. Solution to question nr. 48. *L'intermédiaire des Mathématicians*, 1:107–110, 1894.

[6] Mariusz Grech and Andrzej Kisielewicz. The Černý conjecture for automata respecting intervals of a directed graph. *Discrete Mathematics & Theoretical Computer Science*, 15(3):61–72, 2013.

[7] Harald A. Helfgott and Ákos Seress. On the diameter of permutation groups. *Annals of Mathematics, to appear*.

[8] Jakub Kowalski and Marek Szykula. The Černý conjecture for small automata: experimental report. *CoRR*, abs/1301.2092, 2013.

[9] D. Kozen. Lower bounds for natural proof systems. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 254–266, 1977. doi:10.1109/SFCS.1977.16.

[10] D. Lee and Mihalis Yannakakis. Testing finite-state machines: state identification and verification. *Computers, IEEE Transactions on*, 43(3):306–320, 1994. doi:10.1109/12.272431.

[11] Pavel Martyugin. Complexity of problems concerning reset words for cyclic and eulerian automata. In Béatrice Bouchou-Markhoff, Pascal Caron, Jean-Marc Champarnaud, and Denis Maurel, editors, *Implementation and Application of Automata*, volume 6807 of *Lecture Notes in Computer Science*, pages 238–249. Springer Berlin Heidelberg, 2011. doi:10.1007/978-3-642-22256-6_22.

[12] Pavel V. Martyugin. Careful synchronization of partial automata with restricted alphabets. In Andrei A. Bulatov and Arseny M. Shur, editors, *Computer Science - Theory and Applications*, volume 7913 of *Lecture Notes in Computer Science*, pages 76–87. Springer Berlin Heidelberg, 2013. doi:10.1007/978-3-642-38536-0_7.

[13] B. K. Natarajan. An algorithmic approach to the automated design of parts orienters. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, SFCS '86, pages 132–142, Washington, DC, USA, 1986. IEEE Computer Society. doi:10.1109/SFCS.1986.5.

[14] Jörg Olschewski and Michael Ummels. The complexity of finding reset words in finite automata. In *Proceedings of the 35th international conference on Mathematical foundations of computer science*, MFCS'10, pages 568–579, Berlin, Heidelberg, 2010. Springer-Verlag. doi:10.1007/978-3-642-15155-2_50.

[15] Jean-Eric Pin. On two combinatorial problems arising from automata theory. *Annals of Discrete Mathematics*, 17:535–548, 1983.

[16] Adam Roman. Synchronizing finite automata with short reset words. *Applied Mathematics and Computation*, 209(1):125–136, 2009. doi:10.1016/j.amc.2008.06.019.

[17] Arto Salomaa. Composition sequences for functions over a finite domain. *Theoret. Comput. Sci.*, 292:263–281, 2000. doi:10.1016/S0304-3975(01)00227-4.

[18] Arto Salomaa. A half-century of automata theory. chapter Compositions over a Finite Domain: From Completeness to Synchronizable Automata, pages 131–143. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 2001. doi:10.1142/9789812810168_0007.

[19] Sven Sandberg. Homing and synchronizing sequences. In Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors, *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*, pages 5–33. Springer Berlin Heidelberg, 2005. `doi:10.1007/11498490_2`.

[20] Benjamin Steinberg. The Černý conjecture for one-cluster automata with prime length cycle. *Theoret. Comput. Sci.*, 412(39):5487 – 5491, 2011. `doi:10.1016/j.tcs.2011.06.012`.

[21] A. N. Trahtman. An efficient algorithm finds noticeable trends and examples concerning the Cerny conjecture. In *MFCS'06*, pages 789–800, 2006. `doi:10.1007/11821069_68`.

[22] A. N. Trahtman. Modifying the upper bound on the length of minimal synchronizing word. In *FCT*, pages 173–180, 2011. `doi:10.1007/978-3-642-22953-4_15`.

[23] MikhailV. Volkov. Synchronizing automata and the Cerný conjecture. In Carlos Martín-Vide, Friedrich Otto, and Henning Fernau, editors, *Language and Automata Theory and Applications*, volume 5196 of *Lecture Notes in Computer Science*, pages 11–27. Springer Berlin Heidelberg, 2008. `doi:10.1007/978-3-540-88282-4_4`.

[24] Vojtìch Vorel. Complexity of a problem concerning reset words for eulerian binary automata. In Adrian-Horia Dediu, Carlos Martín-Vide, José-Luis Sierra-Rodríguez, and Bianca Truthe, editors, *Language and Automata Theory and Applications*, volume 8370 of *Lecture Notes in Computer Science*, pages 576–587. Springer International Publishing, 2014. `doi:10.1007/978-3-319-04921-2_47`.