**Noname manuscript No.**
(will be inserted by the editor)

# Walking on Data Words

**Amaldev Manuel · Anca Muscholl · Gabriele Puppis**

Received: date / Accepted: date

**Abstract** Data words are words with additional edges that connect pairs of positions carrying the same data value. We consider a natural model of automaton walking on data words, called Data Walking Automaton, and study its closure properties, expressiveness, and the complexity of some basic decision problems. Specifically, we show that the class of deterministic Data Walking Automata is closed under all Boolean operations, and that the class of non-deterministic Data Walking Automata has decidable emptiness, universality, and containment problems. We also prove that deterministic Data Walking Automata are strictly less expressive than non-deterministic Data Walking Automata, which in turn are captured by Class Memory Automata.

**Keywords** data languages, walking automata

## 1 Introduction

Data words generalize strings over finite alphabets, where the term 'data' denotes the presence of elements from an infinite domain. Formally, *data words* are modeled as finite sequences of elements chosen from a set of the form $\Sigma \times \mathbb{D}$, where $\Sigma$ is a finite alphabet and $\mathbb{D}$ is an infinite alphabet. Elements of $\Sigma$ are called *letters*, while elements of $\mathbb{D}$ are called *data values*. Sets of data words are called *data languages*.

This research has received funding from the ANR project 2010 BLANC 0202 01 FREC and from the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement n. 259454.

Amaldev Manuel
LIAFA, University of Paris Diderot, France
E-mail: amal@liafa.jussieu.fr

Anca Muscholl
LaBRI, University of Bordeaux, France
E-mail: anca@labri.fr

Gabriele Puppis
CNRS / LaBRI, University of Bordeaux, France
E-mail: gabriele.puppis@labri.fr

It comes natural to investigate reasonable mechanisms (e.g., automata, logics, algebras) for specifying languages of data words. Some desirable features of such mechanisms are the decidability of the paradigmatic problems (i.e., emptiness, universality, containment) and effective closures of the recognized languages under Boolean operations and projection. A natural idea is to enhance a finite state machine with data structures to provide some ability to handle data values. Examples of these structures include registers to store data values [8,10], pebbles to mark positions in the data word [13], hash tables to store partitions of the data domain [1]. In [4] Data Automata are introduced and shown to capture the class of data languages definable in two-variable first-order logic over data words. Class Memory Automata [1] provide an alternative view of Data Automata. For all models, except Pebble Automata and Two-way Register Automata, the non-emptiness problem is decidable; universality and, by extension, equivalence and inclusion are undecidable for all non-deterministic models.

In this work we consider data words as sequences of letters with additional edges that connect pairs of positions carrying the same data value. This idea is consistent with the fact that as far as a data word is concerned the actual data value at a position is not relevant, but only the relative equality and inequality of positions with respect to data values. It is also worth noting that none of the above automaton models makes any distinction between permutations of the data values inside data words. Our model of automaton, called Data Walking Automaton, is naturally two-way: it can roughly be seen as a finite state device whose head moves along successor and predecessor positions, as well as along the edges that connect any position to the closest one having the same data value, either to the right or to the left. Remarkably, emptiness, universality, and containment are decidable problems for Data Walking Automata. Our automata capture, up to functional renaming of letters, all data languages recognized by Data Automata. The deterministic subclass of Data Walking Automata is shown to be closed under all Boolean operations (closure under complementation is not immediate since the machines may loop). We can also deduce from previous results on Tree Walking Automata [2,3] that deterministic Data Walking Automata are strictly less powerful than non-deterministic Data Walking Automata, which in turn are subsumed by Data Automata.

Our contributions can be summarized as follows:

1. We adapt the model of walking automaton, originally introduced for trees, to data words.
2. We study closure properties of the classes of data languages recognized by deterministic and non-deterministic walking automata under the operations of union, intersection, complementation, and projection.
3. We analyze the relative expressive power of the deterministic and non-deterministic models of walking automata, comparing them with other classes of automata from the literature, most notably, Data Automata. We also show that deterministic walking automata recognize all data languages definable in the two-variable fragment of first-order logic with access to the global and class successor predicates.
4. We study the complexity of fundamental problems on data languages recognized by non-deterministic walking automata; in particular, we prove that the

problems of word acceptance, emptiness, universality, and containment are decidable.
5. We prove that extending the model of walking automaton with alternation results in an undecidable emptiness problem.

**Organization.** In Section 2 we give some preliminary definitions concerning the standard models of Data Automata and Tiling Automata. In Section 3 we introduce the deterministic and non-deterministic models of walking automata on data words and we prove some basic closure properties. In Section 4 we analyze the expressive power of Data Walking Automata, in comparison with Data Automata, and we prove a series of separation results analogous to those for walking automata on trees. In Section 5 we identify a fragment of first-order logic, precisely, the two-variable fragment with access to the global and class successor predicates, that is captured by the class of deterministic Data Walking Automata. In Section 6 we study the complexity of some fundamental problems involving Data Walking Automata, most notably, word acceptance, emptiness, universality, and containment. In Section 7 we consider the alternating model of Data Walking Automaton and we show that the emptiness problems becomes undecidable in this case. Section 8 provides an assessment of the results and discusses future work.

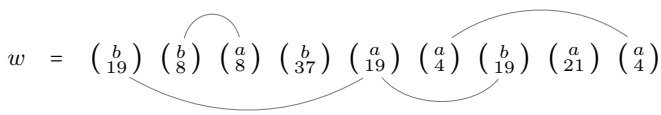
## 2 Preliminaries

Throughout this paper we will tacitly assume that all data words are non-empty – this assumption will simplify some definitions, such as that of Tiling Automaton. Given a data word $w = (a_1, d_1) \cdots (a_n, d_n)$, a *class* of $w$ is a maximal set of positions with identical data value. The set of classes of $w$ forms a partition of the set of positions and is naturally defined by the equivalence relation $i \sim j$ iff $d_i = d_j$.

The *global successor* and *global predecessor* of a position $i$ in a data word $w$ are the positions $i+1$ and $i-1$ (if they exist). The *class successor* of a position $i$ is the leftmost position after $i$ in its class (if it exists) and is denoted by $i \oplus 1$. The *class predecessor* of a position $i$ is the rightmost position before $i$ in its class (if it exists) and is denoted by $i \ominus 1$. The global and class successors of a position are collectively called *successors*, and similarly for the *predecessors*. The successors and predecessors of a position are called its *neighbors*.

Using the above definitions we can identify any data word $w \in (\Sigma \times \mathbb{D})^*$ with a directed graph whose vertices are the positions of $w$, each one labelled by a letter from $\Sigma$, and whose edges are given by the successor and predecessor functions $+1$, $-1$, $\oplus 1$, $\ominus 1$. This graph can be represented in space $\Theta(|w|)$, where $|w|$ denotes the length of $w$. For $1 \le i \le |w|$ we denote by $w(i) \in \Sigma \times \mathbb{D}$ the label of the $i$th position of $w$.

For example, the following is the graph representation of a data word $w$ over the alphabet $\{a, b\} \times \mathbb{N}$:



### 2.1 Local types

Given a data word $w$ and a position $i$ in it, we introduce the local type $\overrightarrow{\mathsf{type}}_w(i)$ (resp., $\overleftarrow{\mathsf{type}}_w(i)$) to specify whether the global and class successors (resp., predecessors) of $i$ exist and whether they coincide or not. Formally, when considering the successors of a position $i$, four scenarios are possible:

1. the position $i$ is the rightmost one in $w$ and hence no successors exist; we denote this by $\overrightarrow{\mathsf{type}}_w(i) = \mathsf{max}$;
2. the position $i$ is not the rightmost position of $w$, but it is the rightmost in its class, in which case the global successor exists but not the class successor; we denote this by $\overrightarrow{\mathsf{type}}_w(i) = \mathsf{cmax}$;
3. both global and class successors of $i$ are defined in $w$ and they coincide, i.e. $i+1 = i \oplus 1$; we denote this by $\overrightarrow{\mathsf{type}}_w(i) = \mathsf{1succ}$;
4. both successors of $i$ are defined in $w$ and they are different, i.e. $i+1 \neq i \oplus 1$; we denote this by $\overrightarrow{\mathsf{type}}_w(i) = \mathsf{2succ}$.

We define $\overrightarrow{\mathsf{Types}} = \{\mathsf{max}, \mathsf{cmax}, \mathsf{1succ}, \mathsf{2succ}\}$ to be the set of possible right types of positions of data words. The symmetric cases for the predecessors of $i$ are signified by the left type $\overleftarrow{\mathsf{type}}_w(i) \in \overleftarrow{\mathsf{Types}} = \{\mathsf{min}, \mathsf{cmin}, \mathsf{1pred}, \mathsf{2pred}\}$. Finally, we define $\mathsf{type}_w(i) = \left(\overleftarrow{\mathsf{type}}_w(i), \overrightarrow{\mathsf{type}}_w(i)\right) \in \mathsf{Types} = \overleftarrow{\mathsf{Types}} \times \overrightarrow{\mathsf{Types}}$.

### 2.2 Class Memory Automata

We will rely on results on Data Automata [4] for our decidability results. However, for convenience we will use an equivalent model called Class Memory Automata [1]. We use $[n]$ to denote the subset $\{1, ..., n\}$ of the natural numbers. Intuitively, a Class Memory Automaton is a finite state automaton enhanced with hash functions that assigns a *memory value* from a finite set $[k]$ to each data value in $\mathbb{D}$. On encountering a pair $(a, d)$, a transition is non-deterministically chosen from a set that depends on the current state of the automaton, the memory value $f(d)$, and the input letter $a$. When a transition on $(a, d)$ is executed, the current state and the memory value of $d$ are updated. Below we give a formal definition of a Class Memory Automaton. Later we will show that this model is also similar to that of Tiling Automata [17].

**Definition 1** A *Class Memory Automaton* (*CMA* for short) is a tuple $\mathcal{C} = (Q, \Sigma, k, \Delta, I, F, K)$, where:

- $Q$ is the finite set of states,
- $\Sigma$ is the finite alphabet,
- $[k]$ is the set of memory values,
- $\Delta \subseteq Q \times \Sigma \times [k] \times Q \times [k]$ is the transition relation,
- $I \subseteq Q$ is the set of initial states,
- $F \subseteq Q$ is the set of final states,
- $K \subseteq [k]$ is the set of final memory values.

*Configurations* are pairs $(q, f)$, with $q \in Q$ and $f \in [k]^{\mathbb{D}}$ – i.e. pairs consisting of a control state and a function from $\mathbb{D}$ to $[k]$. *Transitions* are of the form

$$(q, f) \xrightarrow{(a,d)} (q', f')$$

with $\big(q, a, f(d), q', h'\big) \in \Delta$, $f'(d) = h'$, and $f'(e) = f(e)$ for all $e \in \mathbb{D} \setminus \{d\}$. Sequences of transitions with matching configurations are called *runs*. The *initial configurations* are the pairs $(q_0, f_0)$, with $q_0 \in I$ and $f_0(d) = 1$ for all $d \in \mathbb{D}$; the *final configurations* are the pairs $(q, f)$, with $q \in F$ and $f(d) \in K$ for all $d \in \mathbb{D}$. The *recognized language* $\mathscr{L}(\mathcal{C})$ consists of the data words $w = (a_1, d_1) \cdots (a_n, d_n) \in (\Sigma \times \mathbb{D})^*$ that admit runs of the form $(q_0, f_0) \xrightarrow{(a_1,d_1)} \cdots \xrightarrow{(a_n,d_n)} (q_n, f_n)$, starting in an initial configuration and ending in a final configuration.

It is known that data languages recognized by CMA are effectively closed under union and intersection, but not under complementation. Their emptiness problem is decidable and reduces to reachability in vector addition systems, which is decidable but not known to be of elementary complexity. Inclusion and universality problems for CMA are undecidable.

The following result, paired with closure under intersection, allows us to assume that the information about local types of positions of a data word is available to CMA:

**Proposition 1 (Björklund and Schwentick [1])** *Let $L$ be the set of all data words $w \in (\Sigma \times \mathsf{Types} \times \mathbb{D})^*$ such that, for all positions $i$, $w(i) = (a, \tau, d)$ implies $\tau = \mathsf{type}_w(i)$. The language $L$ is recognized by a CMA.*

### 2.3 Tiling automata

Here we briefly recall the definitions of another class of automata, called *Tiling Automata* or *Graph Automata* [17]. Such automata receive acyclic directed graphs of bounded degree as input and they capture the expressiveness of the existential fragment of monadic second-order logic. In order to accept an input graph, a Tiling Automaton associates, in a non-deterministic way, a *color* (or state) to each node and then checks that the resulting colored spheres satisfy some specific constraints. Here, by colored sphere centered at a node $v$ we mean precisely the subgraph induced by the set of nodes at distance at most $r$ from $v$, for a fixed number $r$ which is a parameter of the automaton (note that this set has bounded size because the input graph has bounded degree). Accordingly, the constraints of a Tiling Automaton are encoded by a finite set of graphs, hereafter called *tiles*, that describe the admitted spheres in an input graph marked with colors. Below, we give a definition of Tiling Automaton that is tailored for graphs representing data words – we refer to [17] for a more general definition and an account of the basic properties. In particular, we fix the the radius of the spheres to be 1, as it is usually done with graphs that represent finite words, trees, or pictures over a finite alphabet. However, we remark that considering only spheres of radius 1 limits the expressiveness of Tiling Automata as acceptors of data words, since the restriction will capture only data languages definable in a strict fragment of existential monadic second-order logic over the relations $+1$ and $\oplus 1$. Subsequently, we will show that CMA and Tiling Automata are equally expressive when operating on the subclass of graphs representing data words.

We fix a finite set $\Gamma$ of colors that are used to color the positions in the input data word. We also reuse the notion of type that we gave in Subsection 2.1. We define a *tile* as a tuple of the form

$$t = (a, \tau, \gamma_0, \gamma_{-1}, \gamma_{\ominus 1}, \gamma_{+1}, \gamma_{\oplus 1})$$

that specifies the possible label $a$ and the possible type $\tau$ of a position $i$, as well as the possible colors $\gamma_0$, $\gamma_{-1}$, $\gamma_{\ominus 1}$, $\gamma_{+1}$, $\gamma_{\oplus 1}$ that can be associated with $i$ and its neighboring positions $i-1$, $i \ominus 1$, $i+1$, $i \oplus 1$. For the sake of brevity, an element $\alpha$ among $0, -1, \ominus 1, +1, \oplus 1$ is called an *axis* and correspondingly the color $\gamma_\alpha$ is denoted by $t[\alpha]$. Clearly, we assume that $t[\alpha]$ is undefined (denoted $t[\alpha] = \perp$) for all and only those axes (i.e., successors or predecessors) that are missing, as indicated by the type $\tau$. Similarly, we assume that $t[-1] = t[\ominus 1]$ (resp., $t[+1] = t[\oplus 1]$) whenever $\tau \in \{\mathsf{1pred}\} \times \overrightarrow{\mathsf{Types}}$ (resp., $\tau \in \overleftarrow{\mathsf{Types}} \times \{\mathsf{1succ}\}$).

For example, if $\tau = (\mathsf{cmin}, \mathsf{1succ})$, then the tuple $t = (a, \tau, \gamma_0, \gamma_{-1}, \gamma_{\ominus 1}, \gamma_{+1}, \gamma_{\oplus 1})$ is a tile only if $\gamma_0, \gamma_{-1} \neq \perp$, $\gamma_{\ominus 1} = \perp$, and $\gamma_{+1} = \gamma_{\oplus 1} \neq \perp$.

**Definition 2** A *Tiling Automaton* is a triple $\mathcal{T} = (\Sigma, \Gamma, T)$ consisting of a finite alphabet $\Sigma$, a finite set $\Gamma$ of colors, and a finite set $T$ of tiles over $\Sigma$ and $\Gamma$. A *tiling* by $\mathcal{T}$ of a data word $w = (a_1, d_1) \ldots (a_n, d_n)$ is a function $\widetilde{w} : [n] \to \Gamma$ such that, for all positions $i$ in $w$, the tile

$$\big( a_i,\ \mathsf{type}_w(i),\ \widetilde{w}(i),\ \widetilde{w}(i-1),\ \widetilde{w}(i \ominus 1),\ \widetilde{w}(i+1),\ \widetilde{w}(i \oplus 1) \big)$$

belongs to $T$. The language recognized by the Tiling Automaton $\mathcal{T}$ consists of all data words that admit a valid tiling by $\mathcal{T}$.

The result below follows from simple translations of automata and depends on the fact that CMA can compute the types of the positions in a data word.

**Proposition 2** *CMA and Tiling Automata on data words are equivalent. Moreover, there exist polynomial-time translations between the two models.*

*Proof.* Let $\mathcal{C} = (Q, k, \Sigma, \Delta, I, F, K)$ be a CMA. Intuitively, the runs of $\mathcal{C}$ can be seen as labellings satisfying the constraints of a suitable Tiling Automaton. More precisely, we introduce the set of colors $\Gamma = Q \times [k]$, where each color is meant to describe the state and the memory value of the data value that appear in a possible run of $\mathcal{C}$ immediately after a given position. To construct an equivalent Tiling Automaton $\mathcal{T} = (\Sigma, \Gamma, T)$, it suffices to describe the set $T$ of admitted tiles. Formally, this set consists of those tuples

$$t \ = \ (a, \tau, \gamma_0, \gamma_{-1}, \gamma_{\ominus 1}, \gamma_{+1}, \gamma_{\oplus 1})$$

that satisfy the following conditions:

- if $\tau \in \{\mathsf{min}\} \times \overrightarrow{\mathsf{Types}}$, then $\gamma_0 = (q', h')$ for some transition $(q_0, a, 1, q', h') \in \Delta$ and some initial state $q_0 \in I$;
- if $\tau \in \{\mathsf{cmin}\} \times \overrightarrow{\mathsf{Types}}$, then $\gamma_{-1} = (q, h)$ for some memory value $h \in [k]$ and $\gamma_0 = (q', h')$ for some transition $(q, a, 1, q', h') \in \Delta$;
- if $\tau \in \{\mathsf{1pred}\} \times \overrightarrow{\mathsf{Types}}$, then $\gamma_{-1} = (q, h)$ and $\gamma_0 = (q', h')$ for some transition $(q, a, h, q', h') \in \Delta$;
- if $\tau \in \{\mathsf{2pred}\} \times \overrightarrow{\mathsf{Types}}$, then $\gamma_{-1} = (q, h)$ for some memory value $h \in [k]$, $\gamma_{\ominus 1} = (q'', h'')$ for some state $q'' \in Q$ and some memory value $h'' \in [k]$, and $\gamma_0 = (q', h')$ for some transition $(q, a, h'', q', h') \in \Delta$;
- if $\tau \in \overleftarrow{\mathsf{Types}} \times \{\mathsf{cmax}\}$, then $\gamma_0 = (q, h)$ for some state $q \in Q$ and some final memory value $h \in K$;

– if $\tau \in \overleftarrow{\mathsf{Types}} \times \{\mathsf{max}\}$, then $\gamma_0 = (q, h)$ for some final state $q \in F$ and some final memory value $h \in K$

(note that we do not need to specify additional conditions on $t$ by considering the case where $\tau \in \overleftarrow{\mathsf{Types}} \times \{\mathsf{1succ}, \mathsf{2succ}\}$, as the required constraints will be enforced when considering the possible tiles associated with the class successor, which have type $\tau' \in \{\mathsf{1pred}, \mathsf{2pred}\} \times \overrightarrow{\mathsf{Types}}$).

Given a data word $w = (a_1, d_1)\,(a_2, d_2)\,\ldots\,(a_n, d_n)$ and a run of the CMA $\mathcal{C}$ on $w$ of the form

$$\rho \;=\; (q_0, f_0) \xrightarrow{(a_1,d_1)} (q_1, f_1) \xrightarrow{(a_2,d_2)} \ldots \xrightarrow{(a_n,d_n)} (q_n, f_n)$$

we let $h_i = f_i(d_i)$ for all $i = 1, ..., n$ and we observe that the following is a valid tiling of $w$ by $\mathcal{T}$ (we succinctly represent it by a string over $Q \times [k]$):

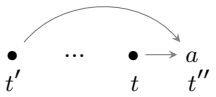
$$\widetilde{w} \;=\; (q_1, h_1)\,(q_2, h_2)\,\ldots\,(q_n, h_n).$$

Conversely, any tiling $\widetilde{w}$ like the above one can be completed into a valid run of $\mathcal{C}$ by defining the hash functions $f_i$ inductively for all $i = 0, ..., n$, as follows:

$$f_i(d) \;=\; \begin{cases} 1 & \text{if } i = 0, \\ h_i & \text{if } d = d_i, \\ f_{i^\star}(d) & \text{if } i > 0,\ d \neq d_i, \text{ and } i^\star = \mathsf{max}\{0\} \cup \{j \mid 1 \leq j < i,\ d = d_j\}. \end{cases}$$

It follows that the Tiling Automaton $\mathcal{T}$ defines the same data language recognized by the CMA $\mathcal{C}$.

For the converse translation, suppose that we are given a Tiling Automaton $\mathcal{T} = (\Sigma, \Gamma, T)$ recognizing a data language $L \subseteq (\Sigma \times \mathbb{D})^*$. In view of Proposition 1, it is sufficient to construct a CMA $\mathcal{C} = (Q, k, \Sigma \times \mathsf{Types}, \Delta, I, F, K)$ that accepts the data words $w \in L$ augmented with the information about the local types. The states of $\mathcal{C}$ are the tiles in $T$, plus a distinguished initial state $q_0$, i.e. $Q = T \uplus \{q_0\}$. Moreover, we let $k = |Q|$ and we identify the memory values in $[k]$ with the states in $Q$; in particular, we identify the memory value 1 with the initial state $q_0$. We now turn towards defining the transition relation $\Delta$ of $\mathcal{C}$. Recall that we identified memory values in $[k]$ with states in $Q$. In particular, this means that a generic transition rule of $\mathcal{C}$ is a tuple of the form $\big(t, t', (a, \tau), t'', t'''\big)$ that, on the basis of the input symbol $(a, \tau)$ and the states $t$ and $t'$ associated, respectively, with the global predecessor and the class predecessor, specifies a possible state $t''$ that could be associated with the current position, as succinctly described by the diagram




(as usual, assume $t' = q_0$ when there is no class predecessor and $t = q_0$ when there is no global predecessor). Hence it suffices to define $\Delta$ as the set of tuples of the form $\big(t, t', (a, \tau), t'', t'''\big)$, with $(a, \tau) \in \Sigma \times \mathsf{Types}$ and $t, t', t'' \in Q$, such that

1. if $\tau \in \{\mathsf{min}\} \times \overrightarrow{\mathsf{Types}}$, then $t = t' = q_0$;
2. if $\tau \in \{\mathsf{cmin}\} \times \overrightarrow{\mathsf{Types}}$, then $t' = q_0$, $t[+1] = t''[0]$, and $t''[-1] = t[0]$;
3. if $\tau \in \{\mathsf{1pred}\} \times \overrightarrow{\mathsf{Types}}$, then $t = t'$, $t[+1] = t[\oplus 1] = t''[0]$, and $t''[-1] = t[0]$;

4. if $\tau \in \{\mathsf{2pred}\} \times \overrightarrow{\mathsf{Types}}$, then $t'[\oplus 1] = t[+1] = t''[0]$, $t''[-1] = t[0]$, and $t''[\ominus 1] = t'[0]$.

Finally, the sets $F$ and $K$ of final states and final memory values contain those tiles $t = (a, \tau, \gamma_0, \ldots, \gamma_{\oplus 1})$ whose type $\tau$ belong to $\overleftarrow{\mathsf{Types}} \times \{\mathsf{max}, \mathsf{cmax}\}$.

Let us now consider a data word $w = (a_1, d_1)\,(a_2, d_2)\,\ldots\,(a_n, d_n)$ and define $w' = (a_1, \tau_1, d_1)\,(a_2, \tau_2, d_2)\,\ldots\,(a_n, \tau_n, d_n)$, where $\tau_i = \mathsf{type}_w(i)$ for all positions $i \in [n]$. Any tiling of $w$ by $\mathcal{T}$ can be turned into a valid run of $\mathcal{C}$ on $w'$ by simply prepending the initial configuration $(q_0, f_0)$ to the sequence of tiles. Conversely, any run of $\mathcal{C}$ on $w'$ devoid of the initial configuration can be seen as a tiling of $w$ by $\mathcal{T}$. □

## 3 Automata walking on data words

An automaton walking on data words is a finite state acceptor that processes a data word by moving its head along the successors and predecessors of positions. We let $\mathsf{Axis} = \{0, +1, \oplus 1, -1, \ominus 1\}$ be the set of the five possible directions of navigation in a data word (0 stands for 'stay in the current position').

**Definition 3** A *Data Walking Automaton* (*DWA* for short) is defined as a tuple $\mathcal{A} = (Q, \Sigma, \Delta, I, F)$, where $Q$ is the finite set of states, $\Sigma$ is the finite alphabet, $\Delta \subseteq Q \times \Sigma \times \mathsf{Types} \times Q \times \mathsf{Axis}$ is the transition relation, $I \subseteq Q$ is the set of initial states, $F \subseteq Q$ is the set of final states.

Let $w = (a_1, d_1) \cdots (a_n, d_n) \in (\Sigma \times \mathbb{D})^*$ be a data word. Given $i \in [n]$ and $\alpha \in \mathsf{Axis}$, we denote by $\alpha(i)$ the position that is reached from $i$ by following the axis $\alpha$ (for instance, if $\alpha = 0$ then $\alpha(i) = i$, if $\alpha = \oplus 1$ then $\alpha(i) = i \oplus 1$, provided that $i$ is not the last element in its class). A configuration of $\mathcal{A}$ is a pair consisting of a state $q \in Q$ and a position $i \in [n]$. A transition is a tuple of the form $(p, i) \xrightarrow{w} (q, j)$ such that $(p, a_i, \tau, q, \alpha) \in \Delta$, with $\tau = \mathsf{type}_w(i)$ and $j = \alpha(i)$. The initial configurations are the pairs $(q_0, i_0)$, with $q_0 \in I$ and $i_0 = 1$. The halting configurations are those pairs $(q, i)$ on which no transition is enabled; such configurations are said to be *final* if $q \in F$. The language $\mathscr{L}(\mathcal{A})$ recognized by $\mathcal{A}$ is the set of all data words $w \in (\Sigma \times \mathbb{D})^*$ that admit a run of $\mathcal{A}$ that starts in an initial configuration and halts in a final configuration.

We will also consider *deterministic* versions of DWA, in which the set $I$ of initial states is a singleton and the transition relation $\Delta$ is a partial function from $Q \times \Sigma \times \mathsf{Types}$ to $Q \times \mathsf{Axis}$.

*Example 1* Let $L_1$ be the set of all data words that contain at most one occurrence of each data value (this language is equally defined by the formula $\forall x \forall y\ x \sim y \to x = y$). A deterministic DWA can recognize $L_1$ by reading the input data word from left to right (along axis +1) and by checking that all positions except the last one have type $(\mathsf{cmin}, \mathsf{cmax})$. When a position with type $(\mathsf{cmin}, \mathsf{max})$ or $(\mathsf{min}, \mathsf{max})$ is reached, the machine halts in an accepting state.

*Example 2* Let $L_2$ be the set of all data words in which every occurrence of $a$ is followed by an occurrence of $b$ in the same class (this is expressed by the formula $\forall x\ a(x) \to \exists y\ b(y) \wedge x < y \wedge x \sim y$). A deterministic DWA can recognize $L_2$ by scanning the input data word along the axis +1. On each position $i$ with left type

cmin, the machine starts a subcomputation that scans the entire class of $i$ along the axis $\oplus 1$, and verifies that every $a$ is followed by a $b$. The subcomputation terminates when a position with right type cmax is reached, after which the machines traverses back the class, up to the position $i$ with left type cmin, and then resumes the main computation from the successor $i+1$. Intuitively, the automaton traverses the data word from left to right in a 'class-first' manner.

*Example 3* Our last example deals with the set $L_3$ of all data words in which every occurrence of $a$ is followed by an occurrence of $b$ that is *not* in the same class (this is expressed by the formula $\forall x.\ a(x) \rightarrow \exists y.\ b(y) \wedge x < y \wedge x \not\sim y$). This language is recognized by a deterministic DWA, although not in an obvious way. Fix a data word $w$. It is easy to see that $w \in L_3$ iff one following cases holds:

1. there is no occurrence of $a$ in $w$,
2. $w$ contains a rightmost occurrence of $b$, say in position $\ell_b$, and all occurrences of $a$ are before $\ell_b$; in addition, we require that either the class of $\ell_b$ does not contain an $a$, or the class of $\ell_b$ contains a rightmost occurrence of $a$, say in position $\ell_a$, and another $b$ appears after $\ell_a$ but outside the class of $\ell_b$.

We show how to verify the second case by a deterministic DWA. For this, the automaton reaches the rightmost position of $w$ and searches backward, following the axis $-1$, the first occurrence of $b$: this puts the head of the automaton in position $\ell_b$. From position $\ell_b$ the automaton searches along the axis $\ominus 1$ an occurrence of $a$. If no occurrence of $a$ is found before seeing the left type cmin, then the automaton halts and accepts. Otherwise, as soon as an $a$ is seen (necessarily at position $\ell_a$), a second phase starts that tries to find another occurrence of $b$ after $\ell_a$ and outside the class of $\ell_b$ (we call such an occurrence a *$b$-witness*). To do this, the automaton moves along the axis $+1$ until it sees a $b$, say at position $i$. After that, it scans the class of $i$ along the axis $\oplus 1$. If the right type cmax is seen before seeing a $b$, this means that $i$ was the position of the last $b$ in the class of $i$: in this case, the automaton goes back to position $i$ (which is now the first position along axis $\ominus 1$ that contains a $b$) and accepts iff another $b$ is seen along the axis $+1$ (thanks to the previous test, that occurrence of $b$ must be outside the class of $\ell_b$ and hence a $b$-witness). Otherwise, if a $b$ is seen in position $j > i$ the automaton backtracks to position $i$ and resumes the search for another occurrence of $b$ along the axis $+1$ (note that if $i$ is a $b$-witness, then $j$ is also a $b$-witness, which will be processed by the automaton eventually).

### 3.1 Closure properties

We show here some basic closure properties for the class of non-deterministic DWA and the class of deterministic DWA under the set theoretic operations of union, intersection, and complementation. We defer to Section 6 a study of (non)closure properties of DWA under projection; there we will be able to build up on a number of results involving the classes of deterministic DWA, non-deterministic DWA, and CMA.

**Proposition 3** *The class of non-deterministic DWA is effectively closed under union and intersection.*

*Proof.* Closure under union for the class of non-deterministic DWA is easily shown by taking a disjoint union of the state space of the two automata. Closure under intersection is shown by assuming without loss of generality that one of the two automata accepts only by halting in the leftmost position and by coupling its final states with the initial states of the other automaton. □

Analogous closure properties hold for the class of deterministic DWA, but now rely on the fact that one can remove loops from deterministic computations.

**Proposition 4** *Given a deterministic DWA $\mathcal{A}$, one can construct in linear time a deterministic DWA $\mathcal{A}'$ equivalent to $\mathcal{A}$ that always halts.*

*Proof.* This proof is an adaptation of Sipser's construction for eliminating loops from deterministic space-bounded Turing machines [16]. We fix for the rest of the proof a deterministic DWA $\mathcal{A} = (Q, \Sigma, \Delta, q_0, F)$ and an input data word $w = (a_1, d_1) \cdots (a_n, d_n) \in (\Sigma \times \mathbb{D})^*$ of length $n$. We define the *configuration graph* of $\mathcal{A}$ on $w$ as the directed graph $\mathcal{G}(\mathcal{A}, w)$ with vertices $Q \times [n]$ and edges of the form $(p, i) \xrightarrow{w} (q, j)$. The *reverse configuration graph* $\mathcal{G}^{\mathsf{rev}}(\mathcal{A}, w)$ is the graph obtained from $\mathcal{G}(\mathcal{A}, w)$ by reversing the edges. The basic argument behind the construction is that the reverse configuration graph $\mathcal{G}^{\mathsf{rev}}(\mathcal{A}, w)$ of a *deterministic* DWA is a forest. If not, there would be two distinct paths from a vertex $(p, i)$ to a vertex $(q, j)$ in $\mathcal{G}^{\mathsf{rev}}(\mathcal{A}, w)$ contradicting the fact that $\mathcal{A}$ is deterministic.

As in the case of Turing machines, without loss of generality we can assume that $\mathcal{A}$ has a unique final state $q_f$ and in the case of a successful run the automaton $\mathcal{A}$ finishes at the last position in state $q_f$. The data word $w$ is in $\mathscr{L}(\mathcal{A})$ if there is a path in $\mathcal{G}^{\mathsf{rev}}(\mathcal{A}, w)$ from the configuration $(q_f, n)$ to the configuration $(q_0, 1)$. We construct a deterministic DWA $\mathcal{A}'$ that searches for such a path by performing a depth-first traversal of the tree rooted at $(q_f, n)$. The idea is implemented in the following way. We fix an arbitrary order on the transitions in $\Delta$. In particular, this allows us to identify the first, second, ... edge leaving a certain node $(q, i)$ in the graph $\mathcal{G}^{\mathsf{rev}}(\mathcal{A}, w)$. The states of the automaton $\mathcal{A}'$ are the transitions of $\mathcal{A}$ and $\mathcal{A}'$ starts at the last position in state corresponding to the first transition with target $q_f$. Traversing the edge from a vertex $(q, j)$ to a child $(p, i)$ in the graph $\mathcal{G}^{\mathsf{rev}}(\mathcal{A}, w)$ is simulated by applying the transition contained in the state of $\mathcal{A}'$ backwards. When a node has no children or all its children have been traversed, the automaton goes to the parent by taking the unique possible transition at that node and computes the next transition for the parent node. At any point during the simulation, if the node $(q_0, 1)$ is visited, then the automaton halts and accepts. Otherwise the simulation terminates eventually at the root $(q_f, n)$ and the input is rejected. □

**Proposition 5** *The class of deterministic DWA is effectively closed under union, intersection, and complementation.*

*Proof.* Thanks to Proposition 4 we can assume, without loss of generality, that deterministic DWA never loop, and always halt in the first position of the input word. Under this assumption, closure under complementation simply amounts at swapping the final and the non-final states. Similarly, unions and intersections of deterministic DWA are computed by chaining the automata, that is, by coupling the halting states of one automaton to the initial states of the other. □

## 4 Deterministic vs non-deterministic DWA

This section is devoted to prove the following separation results:

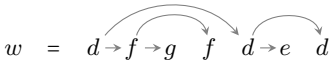
**Theorem 1** *There exist data languages recognized by non-deterministic DWA that cannot be recognized by deterministic DWA. There also exist data languages recognized by CMA that cannot be recognized by non-deterministic DWA.*

Intuitively, the proof of the theorem exploits the fact that one can encode binary trees by suitable data words and think of deterministic DWA (resp. non-deterministic DWA, CMA) as deterministic Tree Walking Automata (resp. non-deterministic Tree Walking Automata, classical bottom-up tree automata). One can then use the results from [2,3] that show that (i) Tree Walking Automata cannot be determinised and (ii) Tree Walking Automata, even non-deterministic ones, cannot recognize all regular tree languages. We develop these ideas in the following subsections.

### 4.1 Encodings of trees

Hereafter we use the term 'tree' (resp. 'forest') to denote a generic finite tree (resp. forest) where each node is labelled with a symbol from a finite alphabet $\Sigma$ and has either 0 or 2 children. To encode trees/forests by data words, we will represent the node-to-left-child and the node-to-right-child relationships by means of the successor functions $+1$ and $\oplus 1$, respectively. In particular, a *leaf* will correspond to a position of the data word with *no class successor*, an *internal node* will correspond to a position where *both class and global successors* are defined (and are distinct), and a *root* will be represented either by the *leftmost position* in the word or by a position with no class predecessor that is immediately preceded by a position with no class successor.

As an example, given pairwise different data values $d, e, f, g$, the complete binary tree of height 2 can be encoded by the following data word:




(to ease the understanding, we only drew the instances of the successor functions $\oplus 1$ and $+1$ that represent left and right edges in the encoded tree).

A formal definition of encoding of a tree or forest follows.

**Definition 4** We say that a data word $w \in (\Sigma \times \mathbb{D})^+$ is a *forest encoding* if there is no position $i$ such that $\overrightarrow{\mathsf{type}}_w(i) = \mathsf{1succ}$ and no pair of consecutive positions $i$ and $i+1$ such that $\overrightarrow{\mathsf{type}}_w(i) = \mathsf{2succ}$ and $\overleftarrow{\mathsf{type}}_w(i+1) = \mathsf{2pred}$.

Given a forest encoding $w$, we denote by $\mathsf{forest}(w)$ the directed graph that has for nodes the positions of $w$, labelled over $\Sigma$, and for edges the pairs

$$(i, i+1) \qquad \text{and} \qquad (i, i \oplus 1)$$

whenever $\overrightarrow{\mathsf{type}}_w(i) = \mathsf{2succ}$.

The fact that $\mathsf{forest}(w)$ is indeed a forest, for every data word $w$ satisfying the above definition, follows from two basic observations: (i) the edges of $\mathsf{forest}(w)$ follow the ordering on the positions of $w$, and hence $\mathsf{forest}(w)$ is a directed *acyclic* graph, and (ii) for all pairs of distinct positions $i, j$ in $w$, if $\overrightarrow{\mathsf{type}}(i) = \overrightarrow{\mathsf{type}}(j) = \mathsf{2succ}$, then $i + 1 \neq j \oplus 1$ (otherwise, we would have $j < i$, $\overrightarrow{\mathsf{type}}(i) = \mathsf{2succ}$, and $\overleftarrow{\mathsf{type}}(i + 1) = \mathsf{2pred}$, contradicting Definition 4), and hence nodes in $\mathsf{forest}(w)$ have *in-degree at most* 1.

In particular, we can identify left and right children, leaves, and roots in $\mathsf{forest}(w)$, based on the following case distinction:

- if $\overrightarrow{\mathsf{type}}_w(i) = \mathsf{2succ}$, then $i + 1$ and $i \oplus 1$ are the targets of two edges departing from $i$; we say that $i+1$ and $i \oplus 1$ are the *left* and *right children* of $i$, respectively;
- if $\overrightarrow{\mathsf{type}}_w(i) \in \{\mathsf{max}, \mathsf{cmax}\}$, then $i$ has no edge departing from it, in which case $i$ is a *leaf*;
- if $\overleftarrow{\mathsf{type}}_w(i) = \mathsf{min}$ or $\overleftarrow{\mathsf{type}}_w(i) = \mathsf{cmin}$ and $\overrightarrow{\mathsf{type}}_w(i - 1) = \mathsf{cmax}$, then $i$ has no edge entering it, and hence we call it *a root*.

Moreover, if $\mathsf{forest}(w)$ contains a single root, then it is a tree and we accordingly define $\mathsf{tree}(w) = \mathsf{forest}(w)$; otherwise, we simply let $\mathsf{tree}(w)$ be undefined. Note that every tree of the form $\mathsf{tree}(w)$ is a *full binary tree*, namely, the internal nodes have always two children.

We remark that there exist several encodings of the same tree/forest that are not isomorphic, even up to permutations of the data values. For instance, the two data words below encode the same complete binary tree of height 2:


Among all possible encodings of a tree/forest, we identify special ones, called *canonical encodings*, in which the nodes are listed following the *pre-order* visit. For example, the above data word $w$ corresponds to a canonical encoding, while $w'$ does not. Clearly, each tree $t$ has a unique canonical encoding, up to permutations of the data values, which we denote by $\mathsf{enc}(t)$.

*Remark 1* We conclude this part by observing that the data language consisting of all forest encodings is recognized by a deterministic DWA: for this it suffices to scan the input data word once from left to right and check that the local types satisfy Definition 4. If in addition the DWA checks that there are no occurrences of the local type $(\mathsf{cmin}, \mathsf{cmax})$, then the recognized language consists of the valid encodings of full binary trees, namely, those data words $w$ such that $\mathsf{tree}(w)$ is defined. On the other hand, the language of the *canonical* encodings of forests/trees is not recognizable by any DWA (even non-deterministic ones).

### 4.2 Separations of tree automata

We will work in this section with full binary trees, hereafter called simply *trees*. We briefly recall the definition of a tree walking automaton and the separation results from [2,3]. In a way similar to DWA, we first introduce local types of nodes inside trees. These can be seen as pairs of labels from the finite sets $\mathsf{Types}^{\downarrow} = \{\mathsf{leaf}, \mathsf{internal}\}$ and $\mathsf{Types}^{\uparrow} = \{\mathsf{root}, \mathsf{leftchild}, \mathsf{rightchild}\}$, and they allow us to distinguish between a leaf and an internal node as well as between a root, a left child, and a right child.

We use a set $\mathsf{TAxis} = \{0, \uparrow, \swarrow, \searrow\}$ of four navigational directions inside a tree: 0 is for staying in the current node, $\uparrow$ is for moving to the parent, $\swarrow$ is for moving to the left child, and $\searrow$ is for moving to the right child.

**Definition 5** A *non-deterministic Tree Walking Automaton* (*TWA*) is a tuple $\mathcal{A} = (Q, \Sigma, \Delta, I, F)$, where:
- $\Sigma$ is the finite alphabet,
- $Q$ is the finite set of states,
- $\Delta \subseteq Q \times \Sigma \times \mathsf{Types}^{\downarrow} \times \mathsf{Types}^{\uparrow} \times Q \times \mathsf{TAxis}$ is the transition relation,
- $I \subseteq Q$ is the set of initial states,
- $F \subseteq Q$ is the sets of final states.

*Runs* of TWA are defined in a way similar to the runs of DWA and begin with the initial state marking the root. The subclass of *deterministic* TWA is obtained by replacing the transition relation $\Delta$ with a partial function from $Q \times \Sigma \times \mathsf{Types}^{\downarrow} \times \mathsf{Types}^{\uparrow}$ to $Q \times \mathsf{TAxis}$ and by letting $I$ consist of a single initial state $q_0$.

**Theorem 2 (Bojańczyk and Colcombet [2,3])** *There exist languages recognized by non-deterministic TWA that cannot be recognized by deterministic TWA. There also exist regular languages of trees that cannot be recognized by non-deterministic TWA.*

### 4.3 Translations between TWA and DWA

Given a tree language $L$, we denote by $L^{\mathsf{enc}}$ the language of all data words that encode (possibly in a non-canonical way) the trees in $L$:

$$L^{\mathsf{enc}} = \big\{ w \;\big|\; \mathsf{tree}(w) \in L \big\}.$$

To derive from Theorem 2 analogous separation results for data languages, we need to provide suitable translations between TWA and DWA, as well as from tree automata to CMA:

**Lemma 1** *Given a deterministic (resp. non-deterministic) TWA $\mathcal{A}$ recognizing $L$, one can construct a deterministic (resp. non-deterministic) DWA $\mathcal{A}^{\mathsf{enc}}$ recognizing $L^{\mathsf{enc}}$. Conversely, given a deterministic (resp. non-deterministic) DWA $\mathcal{A}$, one can construct a deterministic (resp. non-deterministic) TWA $\mathcal{A}^{\mathsf{tree}}$ such that, for any tree $t$, $\mathcal{A}^{\mathsf{tree}}$ accepts $t$ iff $\mathcal{A}$ accepts the canonical encoding $\mathsf{enc}(t)$.*

*Proof.* We prove the first claim for a deterministic TWA $\mathcal{A}$ (the case of a non-deterministic TWA is similar). We recall from Remark 1 that the language consisting of all (possibly non-canonical) encodings of full binary trees is recognized by a deterministic DWA, which we denote by $\mathcal{U}^{\mathsf{enc}}$. We then construct a deterministic DWA $\mathcal{A}'$ such that, given any tree $t$ and any encoding $w$ of $t$, we have $t \in \mathscr{L}(\mathcal{A})$ iff $w \in \mathscr{L}(\mathcal{A}')$ (note that we do not specify the behaviour of $\mathcal{A}'$ on the inputs that are not valid encodings of trees). Formally, given the TWA $\mathcal{A} = (Q, \Sigma, \Delta, q_0, F)$,

we define $\mathcal{A}' = (Q, \Sigma, \Delta', q_0, F)$, where

$$\Delta'\big(p, a, (\overleftarrow{\tau}, \overrightarrow{\tau})\big) = \begin{cases} (q, 0) & \text{if } \Delta\big(p, a, (\tau^{\downarrow}, \tau^{\uparrow})\big) = (q, 0), \\ (q, +1) & \text{if } \overrightarrow{\tau} = \mathsf{2succ} \text{ and } \Delta\big(p, a, (\mathsf{internal}, \tau^{\uparrow})\big) = (q, \swarrow), \\ (q, \oplus 1) & \text{if } \overrightarrow{\tau} = \mathsf{2succ} \text{ and } \Delta\big(p, a, (\mathsf{internal}, \tau^{\uparrow})\big) = (q, \searrow), \\ (q, -1) & \text{if } \overleftarrow{\tau} = \mathsf{cmin} \text{ and } \Delta\big(p, a, (\tau^{\downarrow}, \mathsf{leftchild})\big) = (q, \uparrow), \\ (q, \ominus 1) & \text{if } \overleftarrow{\tau} = \mathsf{2pred} \text{ and } \Delta\big(p, a, (\tau^{\downarrow}, \mathsf{rightchild})\big) = (q, \uparrow) \end{cases}$$

and where $\tau^{\downarrow}$ and $\tau^{\uparrow}$ are obtained from $\overrightarrow{\tau}$ and $\overleftarrow{\tau}$ as follows: either $\tau^{\downarrow} = \mathsf{internal}$ or $\tau^{\downarrow} = \mathsf{leaf}$, depending on whether $\overrightarrow{\tau} = \mathsf{2succ}$ or $\overrightarrow{\tau} \in \{\mathsf{max}, \mathsf{cmax}\}$, and either $\tau^{\uparrow} = \mathsf{root}$, or $\tau^{\uparrow} = \mathsf{leftchild}$, or $\tau^{\uparrow} = \mathsf{rightchild}$, depending on whether $\overleftarrow{\tau} = \mathsf{min}$, or $\overleftarrow{\tau} = \mathsf{cmin}$, or $\overleftarrow{\tau} = \mathsf{2pred}$.

We let the reader check that, for all trees $t$ and all data word encodings $w$ of $t$, $\mathcal{A}$ accepts $t$ iff $\mathcal{A}'$ accepts $w$. We conclude the proof by exploiting the closure properties of DWA and by defining $\mathcal{A}^{\mathsf{enc}}$ as the intersection of $\mathcal{U}^{\mathsf{enc}}$ and $\mathcal{A}'$.

We turn now to the second claim. We fix a deterministic DWA $\mathcal{A}$ (again, the case of a non-deterministic DWA is similar) and we show how to construct a deterministic TWA $\mathcal{A}^{\mathsf{tree}}$ whose behaviour is the same as the behaviour of $\mathcal{A}$ when restricted to *canonical* encodings of trees. For the latter property to make sense, we need to make sure that the behaviour of $\mathcal{A}$ is invariant on the possible different canonical encodings of trees: this is however easy to see, since canonical encodings are unique up to permutation of the data values, and, similarly, computations of DWA are invariant under permutation of the data values.

We recall that the standard definition of a TWA envisages three possible directions of navigation in a tree: $\uparrow$, $\swarrow$, and $\searrow$. For the sake of presentation, we introduce two new axis, denoted $\leftarrow$ and $\rightarrow$, that allow the automaton to move from a certain node $i$ respectively to the predecessor $\leftarrow(i)$ and to the successor $\rightarrow(i)$ of $i$, according to the total ordering induced by the pre-order visit of the tree. We will use these new directions of navigation to mimic the moves of $\mathcal{A}$ between consecutive positions of a canonical encoding. For instance, a move of $\mathcal{A}$ from position $i$ to position $i-1$ in a canonical encoding $w$ of $t$ will be simulated by a corresponding move of $\mathcal{A}^{\mathsf{tree}}$ from node $i$ to the node that immediately precedes $i$ in the pre-order visit of $t$, even in the case when $i$ is not a left child (so $\leftarrow(i) \neq \uparrow(i)$). We also observe that allowing moves along the axis $\leftarrow$ and $\rightarrow$ does not increase the expressive power of TWA. Indeed, when a transition is executed that takes the automaton from node $i$ to node $j = \leftarrow(i)$, then two cases can happen depending on the local type of node $i$ in $t$: either $i$ is a left child, in which case $j$ is simply the parent of $i$, or $i$ is a right child, in which case $j$ is the rightmost leaf in the left subtree of the parent of $i$, i.e. $j = \searrow^n(\swarrow(\uparrow(i)))$ f from what followsor $n$ sufficiently large, and thus the transition can be simulated by a finite sequence of moves along the axis $\uparrow$, $\swarrow$, $\searrow$, ..., $\searrow$. Analogous arguments hold for the transitions that take the automaton from node $i$ to node $j = \rightarrow(i)$.

We also modify our TWA model in order to be able to check simple node properties at each transition – again, this modification does not affect the expressive power. Specifically, we assume that the guards of the transitions of a TWA can distinguish, using refined local types, the last (rightmost) leaf in the pre-order visit of the entire tree from all the other leaves (this feature can be easily implemented

via deterministic subcomputations that start in a leaf and look for the deepest ancestor that is not a right child). We thus refine the local type $\mathsf{leaf} \in \mathsf{Types}^{\downarrow}$ into two new local types $\mathsf{rightmostleaf}$ and $\mathsf{otherleaf}$.

Thanks to the above arguments, we can easily transform the deterministic DWA $\mathcal{A} = (Q, \Sigma, \Delta, q_0, F)$ into a deterministic TWA $\mathcal{A}^{\mathsf{tree}} = (Q, \Sigma, \Delta', q_0, F)$, where

$$\Delta'\big(p, a, (\tau^{\uparrow}, \tau^{\downarrow})\big) = \begin{cases} (q, 0) & \text{if } \Delta\big(p, a, (\overleftarrow{\tau}, \overrightarrow{\tau})\big) = (q, 0), \\ (q, \searrow) & \text{if } \Delta\big(p, a, (\overleftarrow{\tau}, \mathsf{2succ})\big) = (q, \oplus 1), \\ (q, \rightarrow) & \text{if } \tau^{\downarrow} \neq \mathsf{rightmostleaf} \text{ and } \Delta\big(p, a, (\overleftarrow{\tau}, \overrightarrow{\tau})\big) = (q, +1), \\ (q, \uparrow) & \text{if } \Delta\big(p, a, (\mathsf{2pred}, \overrightarrow{\tau})\big) = (q, \ominus 1), \\ (q, \leftarrow) & \text{if } \tau^{\uparrow} \neq \mathsf{root} \text{ and } \Delta\big(p, a, (\overleftarrow{\tau}, \overrightarrow{\tau})\big) = (q, -1) \end{cases}$$

and where $\overleftarrow{\tau}$ and $\overrightarrow{\tau}$ are obtained from $\tau^{\uparrow}$ and $\tau^{\downarrow}$ as follows: $\overleftarrow{\tau}$ = $\mathsf{min}$ / $\mathsf{cmin}$ / $\mathsf{2pred}$ depending on $\tau^{\uparrow}$ = $\mathsf{root}$ / $\mathsf{leftchild}$ / $\mathsf{rightchild}$, and $\overrightarrow{\tau}$ = $\mathsf{max}$ / $\mathsf{cmax}$ / $\mathsf{2succ}$ depending on $\tau^{\downarrow}$ = $\mathsf{rightmostleaf}$ / $\mathsf{otherleaf}$ / $\mathsf{internal}$.

By a slight abuse of notation, we can identify the nodes in a tree $t$ with the corresponding positions in the canonical encoding $\mathsf{enc}(t)$. Under this assumption, it becomes easy to verify that every transition $(p, i) \xrightarrow{t} (q, j)$ of the TWA $\mathcal{A}^{\mathsf{tree}}$ on a tree $t$ can be seen as a transition $(p, i) \xrightarrow{\mathsf{enc}(t)} (q, j)$ of the DWA $\mathcal{A}$ on the canonical encoding $\mathsf{enc}(t)$ and, symmetrically, every transition of $\mathcal{A}$ on the canonical encoding $\mathsf{enc}(t)$ can be seen as a transition of $\mathcal{A}^{\mathsf{tree}}$ on the tree $t$. Analogous properties for the runs of $\mathcal{A}$ and $\mathcal{A}^{\mathsf{tree}}$ follow by a simple inductive argument. This shows that $\mathcal{A}^{\mathsf{tree}}$ accepts precisely those trees whose canonical encodings are accepted by $\mathcal{A}$ starting from the rightmost position. □

**Lemma 2** *Given a tree automaton $\mathcal{A}$ recognizing a regular language $L$, one can construct a CMA $\mathcal{A}^{\mathsf{enc}}$ recognizing $L^{\mathsf{enc}}$.*

*Proof.* The tree automaton $\mathcal{A}$ can be seen as a Tiling Automaton on trees or, equivalently, as a Tiling Automaton on encodings of trees (recall that, by Definition 4, nodes that are neighbors inside a tree $t$ correspond, in any data word that encodes $t$, to positions that are also neighbors). The data language of all valid encodings of trees is also recognized by a Tiling Automaton. The claim follows from Proposition 2 and the fact that CMA are closed under intersection. □

We are now ready to transfer the separation results to data languages:

*Proof of Theorem 1.* Let $L_1$ be a language recognized by a non-deterministic TWA $\mathcal{A}_1$ that cannot be recognized by deterministic TWA (recall that such a language exists thanks to the first claim of Theorem 2). Using the first claim of Lemma 1, we construct a non-deterministic DWA $\mathcal{A}_1^{\mathsf{enc}}$ such that $\mathscr{L}(\mathcal{A}_1^{\mathsf{enc}}) = L_1^{\mathsf{enc}}$. Suppose by way of contradiction that there is a deterministic DWA $\mathcal{B}_1$ that also recognizes $L_1^{\mathsf{enc}}$. We apply the second claim of Lemma 1 and we obtain a deterministic TWA $\mathcal{B}_1^{\mathsf{tree}}$ that accepts all and only the trees whose canonical encodings are accepted by $\mathcal{B}_1$. Since $L_1^{\mathsf{enc}} = \{w \mid \mathsf{tree}(w) \in L_1\}$ is invariant under equivalent encodings of trees (that is, $w \in L_1^{\mathsf{enc}}$ iff $w' \in L_1^{\mathsf{enc}}$ whenever $\mathsf{tree}(w) = \mathsf{tree}(w')$), we have that $t \in L_1$ iff $\mathsf{enc}(t) \in L_1^{\mathsf{enc}}$, iff $t \in \mathscr{L}(\mathcal{B}_1^{\mathsf{tree}})$. We have just shown that the deterministic TWA $\mathcal{B}_1^{\mathsf{tree}}$ recognizes the language $L_1$, which contradicts the assumption on $L_1$.

By applying similar arguments to a regular tree language $L_2$ that is not recognizable by non-deterministic TWA (cf. second claim of Theorem 2), one can separate CMA from non-deterministic DWA. □

We conclude this section with a couple of remarks. We know from the previous results that if non-deterministic TWA were not closed under complementation, as one reasonably expects, then by Lemma 1 non-deterministic DWA would not be closed under complementation either. Unfortunately, we are not able to show that the class of non-deterministic DWA is not closed under complementation. We conjecture however that the following language cannot be complemented within the class of non-deterministic DWA:

$$L_{\mathsf{bridges}} \;=\; \Big\{\, w_1\; \overset{\frown}{d\; w_2\; d}\; w_3\; \overset{\frown}{e\; w_4\; e}\; w_5\; \overset{\frown}{f\; w_6\; f}\; w_7 \;\;\Big|\;\; d,e,f \in \mathbb{D},\; w_1,\ldots,w_7 \in \mathbb{D}^* \,\Big\}.$$

Finally, we observe that the language $L_{\mathsf{bridges}}$ is definable in the two-variable fragment of first-order logic with access to the linear order $<$ on positions and either the class successor predicate $\oplus 1$ or the data equality predicate $\sim$. It is also definable in Basic Data LTL, a linear temporal logic with 2-sorted operators, working over the string projection and the data classes, see [9].

## 5 A fragment of first-order logic captured by DWA

Two-variable fragments of first-order logics have been extensively studied in the literature, especially in connection with data languages. For example, in [4] the logic $\mathsf{FO}^2[\Sigma, +1, \leq, \sim]$, which uses the global successor, the total ordering relation, and a third predicate $\sim$ for comparing data values, has been considered and proved decidable by reduction to emptiness of Data Automata. Other examples of logical formalisms that use at most two variables and some binary predicates were studied in [5, 11, 15].

In this section we consider the two-variable fragment of first-order logic with global successor and class successor predicates, and we prove that sentences in this logic can be translated to equivalent deterministic DWA. More precisely, the logic under consideration is denoted $\mathsf{FO}^2[\Sigma, +1, \oplus 1]$ and consists of first-order formulas that use at most two variable names, unary predicates corresponding to the letters in the finite alphabet $\Sigma$, and the global and class successors predicates $+1$ and $\oplus 1$. Data words can be naturally seen as models of $\mathsf{FO}^2[\Sigma, +1, \oplus 1]$ sentences.

Intuitively, the fact that deterministic DWA recognize all data languages definable in $\mathsf{FO}^2[\Sigma, +1, \oplus 1]$ follows from two basic observations:

1. every $\mathsf{FO}^2[\Sigma, +1, \oplus 1]$ sentence can be rewritten into a Boolean combination of locally threshold testable conditions of the form *"local property $\alpha(x)$ is satisfied on $k$ distinct positions"*, where "local property" roughly means a formula that can be evaluated over a small neighborhood of the position;
2. deterministic DWA can easily count, up to some given bound, the number of positions $x$ in a data word where a certain local property $\alpha(x)$ holds; since they are closed under unions and intersections, deterministic DWA can thus evaluate Boolean combinations of locally threshold testable conditions.

The first observation can be seen as a variant of Gaifman locality theorem [12] in the specific setting of data words and two-variable first-order formulas. Even though the proof of Gaifman locality theorem for first-order logic is usually given in terms of Ehrenfeucht-Fraïssé games and graph-theoretic notions such as that of neighborhood-type, here we prefer to give a more direct translation from $\mathsf{FO}^2[\Sigma, +1, \oplus 1]$ formulas to Boolean combinations of locally threshold testable conditions. This choice is also motivated by the fact that two-variable formulas cannot describe precisely the isomorphism types of subgraphs induced by neighboring positions, as it is the case for instance with full first-order formulas.

In the following, it is convenient to think of a data word $w \in (\Sigma \times \mathbb{D})^*$ as a directed labeled graph $G_w = (V, \rightrightarrows, \not\rightrightarrows, \rightrightarrows\!\!\!\!\!\!\not\;\;)$, where:
- $V = (V_a)_{a \in \Sigma}$ is the partition of the domain of $w$ into sets $V_a = \{i \mid w(i) = a\}$,
- $i \rightrightarrows j$ iff $j = i + 1 = i \oplus 1$ (i.e. $j$ is both a successor and a class successor of $i$),
- $i \not\rightrightarrows j$ iff $j = i + 1$ and either $i \oplus 1$ is undefined or $j \neq i \oplus 1$,
- $i \rightrightarrows\!\!\!\!\!\!\not\;\; j$ iff $j = i \oplus 1$ and either $i + 1$ is undefined or $j \neq i + 1$.

We denote by $\mathsf{dist}_w(i, j)$ the length of the shortest path between $i$ and $j$ in the undirected graph obtained from $G_w$.

We will freely use shorthands of formulas such as $x \rightrightarrows y$ for $y = x + 1 \wedge y = x \oplus 1$, $x \not\rightrightarrows y$ for $y = x + 1 \wedge y \neq x \oplus 1$, $x \rightrightarrows\!\!\!\!\!\!\not\;\; y$ for $y \neq x + 1 \wedge y = x \oplus 1$, and $\mathsf{dist}(x, y) > 1$ for $y \neq x + 1 \wedge y \neq x \oplus 1 \wedge x \neq y + 1 \wedge x \neq y \oplus 1$. Moreover, we will assume that all existential quantifications in $\mathsf{FO}^2[\Sigma, +1, \oplus 1]$ are of the form

$$\exists y \left(\varphi(y) \ \wedge \ \tau(x, y)\right) \tag{$\dagger$}$$

where $\varphi(y)$ does not contain any free occurrence of the variable $x$ and $\tau(x, y)$ is a formula among $x \rightrightarrows y$, $x \leftleftarrows y$, $x \not\rightrightarrows y$, $x \not\leftleftarrows y$, $x \rightrightarrows\!\!\!\!\!\!\not\;\; y$, $x \not\leftleftarrows y$, $\mathsf{dist}(x, y) > 1$. We can do so, without loss of generality, because every atomic relation between $x$ and $y$ (e.g. $y = x \oplus 1$) can be seen as disjunction of formulas $\tau(x, y)$ of the previous form and because existential quantification commutes with disjunction.

We are interested in special forms of $\mathsf{FO}^2[\Sigma, +1, \oplus 1]$ formulas with one free variable $x$, which can be evaluated over small neighborhoods of $G_w$. Formally, we define *$\ell$-local formulas* by induction on $\ell \in \mathbb{N}$, as follows:
- $a(x)$ is 0-local for all letters $a \in \Sigma$,
- $\alpha(x) \ \wedge \ \beta(x)$ is $\ell$-local if both $\alpha(x)$ and $\beta(x)$ are $\ell$-local,
- $\neg\alpha(x)$ is $\ell$-local if $\alpha(x)$ is $\ell$-local,
- $\exists y \left(\alpha(y) \wedge \tau(x, y)\right)$ is $(\ell{+}1)$-local if $\alpha(y)$ is $\ell$-local and $\tau(x, y)$ entails $\mathsf{dist}(x, y) = 1$, namely, $\tau(x, y) \in \{x \rightrightarrows y, \ y \leftleftarrows x, \ x \not\rightrightarrows y, \ y \not\leftleftarrows x, \ x \rightrightarrows\!\!\!\!\!\!\not\;\; y, y \not\leftleftarrows x\}$.

It is not surprising that deterministic DWA can evaluate $\ell$-local formulas on data words via computations of bounded length that start and end in the same position:

**Lemma 3** *Given an $\ell$-local formula $\alpha(x)$, one can construct a deterministic DWA $\mathcal{A}$ such that, for all data words $w$ and all positions $i$, if $\mathcal{A}$ starts in $i$, then it halts again in $i$ and it accepts or rejects depending on whether $(w, i) \vDash \alpha(x)$.*

*Proof.* The construction of the automaton $\mathcal{A}$ follows the syntactic structure of the local formula and exploits basic closure properties of the considered subclass of deterministic DWA.

The base case consists of translating a predicate $a(x)$ into a single-transition automaton that moves from the initial state to a halting state that is either accepting or rejecting depending on the label of the current position.

For the inductive step, the cases of Boolean combinations $\alpha(x) \wedge \beta(x)$ and $\neg\alpha(x)$ are dealt with by using the standard constructions of concatenation of runs and complementation of automata. Finally, the translation of an $(\ell+1)$-local formula $\exists y \left(\alpha(y) \wedge \tau(x,y)\right)$ is done by a simple case distinction based on the form of $\tau(x,y)$. For instance, if $\tau(x,y) = (x \rightrightarrows y)$, then the automaton tests that the current position has type $\mathsf{1succ}$ (if not it halts and rejects), then moves to the successor of the current position, simulates the automaton for the $\ell$-local formula $\alpha(y)$, moves back to the original position, and halts in an accepting or rejecting state depending on the result of the subcomputation for $\alpha(y)$. Analogous constructions can be given for the remaining cases. □

Thanks to the above lemma and to the fact that, up to logical equivalence, there exist only finitely many $\ell$-local formulas, we can treat $\ell$-local formulas in the same way as we treat letters from the finite alphabet $\Sigma$.

In order to show that $\mathsf{FO}^2[\Sigma, +1, \oplus 1]$ definable data languages are recognized by deterministic DWA, we will first give a translation of $\mathsf{FO}^2[\Sigma, +1, \oplus 1]$ sentences towards Boolean combinations of constraints that count, up to a certain threshold, the number of positions satisfying some local formulas. For this we introduce an intermediate logical language, denoted $\mathsf{FO}^2_{\mathsf{count}}[\Sigma, +1, \oplus 1]$, that extends $\mathsf{FO}^2[\Sigma, +1, \oplus 1]$ by adding sentences with counting quantifiers of the form $\exists^{\geq k} y\ \alpha(y)$, where $k \in \mathbb{N}$ and $\alpha(y)$ is an $\ell$-local formula for some $\ell \in \mathbb{N}$. These new sentences are interpreted on data words in the following natural way:

$$w \vDash \exists^{\geq k} y\ \alpha(y) \qquad \text{iff} \qquad \begin{array}{l} w \text{ contains } k \text{ distinct positions } i_1, \ldots, i_k \\ \text{such that, for all } j = 1, \ldots, k,\ (w, i_j) \vDash \alpha(y). \end{array}$$

The general idea is to transform inductively, starting from the innermost subformulas, any $\mathsf{FO}^2_{\mathsf{count}}[\Sigma, +1, \oplus 1]$ formula $\varphi(x)$ into an equivalent Boolean combination of $\ell$-local formulas $\alpha(x)$, for a suitable $\ell \in \mathbb{N}$, and global constraints of the form $\exists^{\geq k} y\ \alpha(y)$. The following lemma provides the inductive translation in the most interesting case (i.e. quantification over points that are far from the instance of the free variable):

**Lemma 4** *Let $\varphi(x) = \exists y \left(\alpha(y) \wedge \mathsf{dist}(x,y) > 1\right)$ be an $\mathsf{FO}^2[\Sigma, +1, \oplus 1]$ formula, where $\alpha(y)$ is $\ell$-local. Let $E = \{ \rightrightarrows, \leftleftarrows, \not\rightrightarrows, \not\leftleftarrows, \rightrightarrows, \leftleftarrows \}$ be the set of all edge relations witnessing distance 1. We have that $\varphi(x)$ is logically equivalent to the following Boolean combination of $(\ell+1)$-local and global constraints:*

$$\bigvee_{I \subseteq E \cup \{0\}} \left( \bigwedge_{e \in I} \alpha^e(x) \right) \wedge \left( \bigwedge_{e \in E \cup \{0\} \setminus I} \neg \alpha^e(x) \right) \wedge \exists^{\geq |I|+1} y\ \alpha(y)$$

*where $\alpha^0(x) = \alpha(x)$ and $\alpha^e(x) = \exists y. \left(\alpha(y) \wedge x\ e\ y\right)$ for all $e \in E$ (note that $\alpha^e(x)$ is an $(\ell+1)$-local formula).*

*Proof.* The proof of this lemma is a case distinction based on which positions $y$ at distance at most 1 from $x$ satisfy the local formula $\alpha(y)$. Precisely, we consider some data word $w$ and a position $i$ in it. For each $e \in E$, we denote by $j_e$ the unique position in $G_w$ such that $i\ e\ j_e$ (note that $\mathsf{dist}(i, j_e) = 1$). For convenience, we also let $j_0 = i$. We then define $I$ to be the set of all indices $e \in E \cup \{0\}$ such that $(w, j_e) \vDash \alpha(y)$. By construction, $w$ contains exactly $|I|$ positions at distance

at most 1 from $i$ that satisfy $\alpha(y)$. We conclude that $(w,i) \vDash \varphi(x)$ iff there is a position $y$ at distance more than 1 from $x$ that satisfies $\alpha(y)$, iff $w$ contains at least $|I|+1$ positions that satisfy $\alpha(y)$. □

We can now show how to turn an $\mathsf{FO}^2[\Sigma,+1,\oplus 1]$ sentence into a Boolean combination of constraints of the form $\exists^{\geq k} y\ \alpha(y)$, with $\alpha(y)$ local:

**Theorem 3** *Every $\mathsf{FO}^2[\Sigma,+1,\oplus 1]$ sentence is logically equivalent to a Boolean combination of global constraints of the form $\exists^{\geq k} y\ \alpha(y)$, where $k \in \mathbb{N}$ and $\alpha(y)$ is $\ell$-local for some $\ell \in \mathbb{N}$.*

*Proof.* We prove the following stronger claim: every $\mathsf{FO}^2_{\mathsf{count}}[\Sigma,+1,\oplus 1]$ formula (or sentence) $\Psi$ can be transformed into a *normal form* that consists of a Boolean combination of $\ell$-local formulas, for some $\ell \in \mathbb{N}$, and global constraints $\exists^{\geq k} y\ \alpha(y)$. To prove this claim we use an induction on the number $N$ of subformulas of $\Psi$ that have a single free variable and are not yet normalized. The base case $N = 0$ is vacuously true. As for the inductive step, we consider an *innermost* subformula $\phi(x)$ of $\Psi$ that is not yet normalized and we show how to normalize it. Since all proper subformulas of $\phi(x)$ are normalized, we know that $\phi(x)$ cannot be local, nor can start with a Boolean connective (otherwise $\phi(x)$ would have been already in normal form). Moreover, recall that every universally quantified formula $\forall y\ \varphi(y)$ can be seen as a shorthand for $\neg\exists y \neg\varphi(y)$, and that existentially quantified formulas are assumed to be in the form defined by Equation (†). Based on these arguments, we know that $\phi(x)$ is of the form

$$\phi(x) \;=\; \exists y \left(\varphi(y) \;\wedge\; \mathsf{dist}(x,y) > 1\right)$$

where $\varphi(y)$ is normalized and contains no free occurrence of the variable $x$. We then consider the global constraints that occur as maximal subformulas of $\varphi(y)$: since these are sentences with no free variable, they commute with the existential quantification on $y$. In particular, $\phi(x)$ is logically equivalent to a Boolean combination of formulas of the form

$$\phi'(x) \;=\; \gamma'_i \;\wedge\; \exists y \left(\alpha'_i(y) \;\wedge\; \mathsf{dist}(x,y) > 1\right)$$

where $\gamma'_i$ is a global constraint and $\alpha'_i(y)$ is a local formula. Finally, we can apply Lemma 4 to transform each subformula $\exists y \left(\alpha'_i(y) \;\wedge\; \mathsf{dist}(x,y) > 1\right)$ in $\phi'(x)$ to an equivalent Boolean combination of local and global constraints. In this way we obtain a normalized formula $\phi''(x)$ equivalent to $\phi(x)$. □

**Corollary 1** *Deterministic DWA recognize all data languages that are definable in $\mathsf{FO}^2[\Sigma,+1,\oplus 1]$ (or even in $\mathsf{FO}^2_{\mathsf{count}}[\Sigma,+1,\oplus 1]$).*

*Proof.* Thanks to Theorem 3 every $\mathsf{FO}^2[\Sigma,+1,\oplus 1]$ sentence $\Psi$ is equivalent to a Boolean combination $\Psi'$ of global constraints $\gamma_1,\ldots,\gamma_n$, where $\gamma_j \;=\; \exists^{\geq k_j} y\ \alpha_j(y)$ for all $1 \leq j \leq n$, with $k_1,\ldots,k_n \in \mathbb{N}$ and $\alpha_1(y),\ldots,\alpha_n(y)$ local formulas. We can use Lemma 3 to turn each local formula $\alpha_j(y)$ into an equivalent deterministic DWA $\mathcal{A}_j$. Moreover, we can introduce a new alphabet $\Gamma = \mathscr{P}(\{c_1,\ldots,c_n\})$ and construct a deterministic finite state automaton $\mathcal{B}$ that scans any word $\hat{w} \in \Gamma^*$, storing in its control state the number $h_j$ of occurrences of each predicate $c_j$, up to threshold $k_j$, and accepting iff the formula $\Psi'$ is satisfied when we substitute each constraint

$\gamma_j = \exists^{\geq k_j} y\ \alpha_j(y)$ with the condition $h_j \geq k_j$. Now, we let $L$ be the data language defined by $\Psi$. By construction, $\mathcal{B}$ recognizes the following language over $\Gamma$:

$$\hat{L} = \left\{ \hat{w} \mid \exists w \in L,\ \forall 1 \leq i \leq |w| = |\hat{w}|,\ \hat{w}(i) = \{c_j \mid (w,i) \vDash \alpha_j\} \right\}.$$

Using standard constructions in automata theory, one can show that the substitution in $\mathcal{B}$ of each predicate $c_j$ with the subautomaton $\mathcal{A}_j$ results in a deterministic DWA that recognizes the language $L$. □

## 6 Decision problems on DWA

We analyze in detail the complexity of the decision problems on DWA. We start by considering the simpler membership problem, which consists of deciding whether $w \in \mathscr{L}(\mathcal{A})$ for a DWA $\mathcal{A}$ and a data word $w$, both given as input. Subsequently, we move to the emptiness and universality problems, which consist of deciding, respectively, whether a given DWA accepts at least one data word and whether a given DWA accepts all data words. We will show that these problems are decidable, as well as the more general problems of containment and equivalence.

### 6.1 Membership

Compared to other classes of automata on data words (e.g. CMA, Register Automata), deterministic DWA have a membership problem of very low time/space complexity. Moreover, the complexity of the membership problem does not get much worse if we consider non-deterministic DWA. We assume the reader to be familiar with circuit complexity and, in particular, with constant-depth (e.g. $\mathrm{AC}^0$) reductions [18].

**Proposition 6** *The membership problem for a deterministic DWA $\mathcal{A}$ and a data word $w$ is decidable in time $\mathcal{O}(|w| \cdot |\mathcal{A}|)$ and is* LogSpace*-complete under $\mathrm{AC}^0$ reductions. Similarly, the membership problem for non-deterministic DWA is* NLogSpace*-complete.*

*Proof.* To decide in deterministic linear time whether a given deterministic DWA $\mathcal{A}$ accepts a given data word $w$, it is not just sufficient to simulate the run of $\mathcal{A}$ on $w$, since $\mathcal{A}$ may reject $w$ by entering an infinite loop. We use instead Proposition 4 to compute a non-looping deterministic DWA $\mathcal{A}'$ equivalent to $\mathcal{A}$. Recall that $\mathcal{A}'$ can be computed from $\mathcal{A}$ in linear time and hence $|\mathcal{A}'| = \mathcal{O}(|\mathcal{A}|)$. Then we simulate the run of $\mathcal{A}'$ on $w$. Overall, this requires time $\mathcal{O}(|\mathcal{A}| + |\mathcal{A}'| \cdot |w|) = \mathcal{O}(|\mathcal{A}| \cdot |w|)$ and space $\mathcal{O}(\log|\mathcal{A}| + \log|w|)$. For hardness, we note that the membership problem is LogSpace-hard under $\mathrm{AC}^0$ reductions already for deterministic finite state automata (see, for example, [7]).

As for non-deterministic DWA, it suffices to observe that a non-deterministic logarithmic-space Turing machine can easily guess and simulate a run of a given DWA $\mathcal{A}$ on a given data word $w$. This shows that the membership problem for non-deterministic DWA is in NLogSpace. Moreover, the membership problem is known to be NLogSpace-hard already for non-deterministic finite state automata. □

### 6.2 Emptiness

We start by reducing the emptiness of CMA to the emptiness of deterministic DWA (or, equivalently, to universality of deterministic DWA). For this purpose, it is convenient to first translate the input CMA $\mathcal{A}$ into an equivalent Tiling Automaton $\mathcal{T} = (\Sigma, \Gamma, T)$, using Proposition 2. We denote by $\mathsf{Tilings}(\mathcal{T})$ the set of data words over $\Sigma \times \mathbb{D}$ expanded by valid tilings on them – we think of the latter set as a data language over the alphabet $\Gamma \times \Sigma \times \mathbb{D}$. Now, given a data word $\widetilde{w} \in (\Gamma \times \Sigma \times \mathbb{D})^*$, checking whether $\widetilde{w}$ belongs to $\mathsf{Tilings}(\mathcal{T})$ reduces to checking constraints on neighborhoods of positions. Since this can be done by a deterministic DWA, we get the following result:

**Proposition 7** *Given a Tiling Automaton $\mathcal{T}$, one can construct in polynomial time a deterministic DWA $\mathcal{T}^{\mathsf{tiling}}$ that recognizes the data language $\mathsf{Tilings}(\mathcal{T})$.*

Three important corollaries follow from the above proposition. The corollaries concern the operation of *functional projection*, formally specified by a function $f : \Sigma \to \Sigma'$ and mapping any data word $w = (a_1, d_1) \dots (a_n, d_n)$ over $\Sigma \times \mathbb{D}$ to the data word $f(w) = (f(a_1), d_1) \dots (f(a_n), d_n)$ over $\Sigma' \times \mathbb{D}$.

**Corollary 2** *Data languages recognized by CMA are functional projections of data languages recognized by deterministic DWA.*

**Corollary 3** *The class of non-deterministic DWA and that of deterministic DWA are not closed under functional projections.*

*Proof.* If non-deterministic DWA would capture functional projections of deterministic DWA, then, by the previous result, they would also capture the languages recognized by CMA, which would contradict Theorem 1. □

**Corollary 4** *Emptiness and universality of deterministic DWA is at least as hard as emptiness of CMA, which in turn is at least as hard as reachability in Petri nets [4].*

We now turn to showing that languages recognized by non-deterministic DWA are also recognized by CMA, and hence emptiness of DWA is reducible to emptiness of CMA. Let $\mathcal{A} = (Q, \Sigma, \Delta, I, F)$ be a non-deterministic DWA. Without loss of generality, we can assume that $\mathcal{A}$ has a single initial state $q_0$ and a single final state $q_f$. We can also assume that whenever $\mathcal{A}$ accepts a data word $w$, it does so by halting in the rightmost position of $w$. For the sake of brevity, given a transition $\delta = (p, a, \tau, q, \alpha) \in \Delta$, we define $\mathsf{source}(\delta) = p$, $\mathsf{target}(\delta) = q$, $\mathsf{letter}(\delta) = a$, $\mathsf{type}(\delta) = \tau$, and $\mathsf{reach}(\delta) = \alpha$. Below, we introduce the concept of min-flow, which can be thought of as a special form of tiling that witnesses acceptance of a data word $w$ by $\mathcal{A}$. Min-flows are similar to crossing sequences, which were used by Rabin and Scott in [14] to transform two-way finite state automata to equivalent one-way automata – a difference here is that we cannot avoid, or easily detect, the presence of disconnected components in a min-flow.

**Definition 6** Let $w = (a_1, d_1) \dots (a_n, d_n)$ be a data word of length $n$. A *min-flow on $w$* is any map $\mu : [n] \to 2^{\Delta}$ that satisfies the following conditions:
1. There is a transition $\delta \in \mu(1)$ such that $\mathsf{source}(\delta) = q_0$;
2. There is a transition $\delta \in \mu(n)$ such that $\mathsf{target}(\delta) = q_f$;

3. For all positions $i \in [n]$, if $\delta \in \mu(i)$, then $\mathsf{letter}(\delta) = a_i$ and $\mathsf{type}(\delta) = \mathsf{type}_w(i)$;
4. For each $i \in [n]$ and each $q \in Q$, there is at most one transition $\delta \in \mu(i)$ such that $\mathsf{source}(\delta) = q$;
5. For each $i \in [n]$ and each $q \in Q$, there is at most one position $j \in [n]$ for which there is $\delta \in \mu(j)$ such that $\mathsf{target}(\delta) = q$ and $i = \mathsf{reach}(\delta)(j)$;
6. For each $i \in [n]$, let $\mathsf{exiting}(i)$ be the set of all states of the form $\mathsf{source}(\delta)$ for some $\delta \in \mu(i)$; similarly, let $\mathsf{entering}(i)$ be the set of all states of the form $\mathsf{target}(\delta)$ for some $\delta \in \mu(j)$ and some $j \in [n]$ such that $i = \mathsf{reach}(\delta)(j)$; our last condition states that for all positions $i \in [n]$,
   (a) if $i = 1$, then $\mathsf{entering}(i) = \mathsf{exiting}(i) \setminus \{q_0\}$,
   (b) if $i = n$, then $\mathsf{exiting}(i) = \mathsf{entering}(i) \setminus \{q_f\}$,
   (c) otherwise, $\mathsf{exiting}(i) = \mathsf{entering}(i)$.

**Lemma 5** *$\mathcal{A}$ accepts $w$ iff there is a min-flow $\mu$ on $w$.*

*Proof.* Let $w = (a_1, d_1) \cdots (a_n, d_n)$ be a data word of length $n$ and let $\rho$ be a successful run of $\mathcal{A}$ on $w$ of the form $(q_0, i_0) \xrightarrow{w} (q_1, i_1) \xrightarrow{w} \dots (q_m, i_m)$ obtained by the sequence of transitions $\delta_1, \dots, \delta_m$. Without loss of generality, we can assume that no position in $\rho$ is visited twice with the same state (indeed, if $i_k = i_h$ and $q_k = q_h$ for some $k \neq h$, then $\rho$ would contain a loop that can be eliminated without affecting acceptance). We associate with each position $i \in [n]$ the set $\mu(i) = \{\delta_k \mid 1 \leq k \leq m,\ i_k = i\}$. One can easily verify that $\mu$ is a min-flow on $w$.

For the other direction, we assume that there is a min-flow $\mu$ on $w$. We construct the edge-labeled graph $G_\mu$ with vertices in $Q \times [n]$ and edges of the form $\big((p, i), (q, j)\big)$ labeled by a transition $\delta$, where $i \in [n]$, $\delta \in \mu(i)$, $p = \mathsf{source}(\delta)$, $q = \mathsf{target}(\delta)$, and $j = \mathsf{reach}(\delta)(i)$. By construction, every vertex of $G_\mu$ has the same in-degree as the out-degree (either 0 or 1), with the only exceptions being the vertex $(q_0, 1)$ of in-degree 0 and out-degree 1, and the vertex $(q_f, n)$ of in-degree 1 and out-degree 0. One way to construct a successful run of $\mathcal{A}$ on $w$ is to repeatedly choose the *only* vertex $x$ in $G_\mu$ with in-degree 0 and out-degree 1, execute the transition $\delta$ that labels the *only* edge departing from $x$, and remove that edge from $G_\mu$. This procedure terminates when no edge of $G_\mu$ can be removed and it produces a successful run on $w$. □

Since min-flows are special forms of tilings, CMA can guess them and hence:

**Theorem 4** *Given a DWA, one can construct an equivalent CMA. In particular, emptiness of DWA is a decidable problem.*

### 6.3 Universality

Here we show that the complement of the language recognized by a DWA is also recognized by a CMA, and hence universality of DWA is reducible to emptiness of CMA. As usual, we fix a DWA $\mathcal{A} = (Q, \Sigma, \Delta, I, F)$, with $I = \{q_0\}$ and $F = \{q_f\}$, and we assume that $\mathcal{A}$ halts only on rightmost positions. Below we define max-flows, which, dually to min-flows, can be seen as a special forms of tilings witnessing non-acceptance.

**Definition 7** Let $w = (a_1, d_1) \dots (a_n, d_n)$ be a data word of length $n$. A *max-flow on $w$* is any map $\nu : [n] \to 2^Q$ that satisfies the following conditions:

1. $q_0 \in \nu(1)$ and $q_f \notin \nu(n)$,
2. for all positions $i \in [n]$ and all transitions $\delta \in \Delta$, if $\mathsf{source}(\delta) \in \nu(i)$, $\mathsf{letter}(\delta) = a_i$, and $\mathsf{type}(\delta) = \mathsf{type}_w(i)$, then $\mathsf{target}(\delta) \in \nu(\mathsf{reach}(\delta)(i))$.

**Lemma 6** *$\mathcal{A}$ rejects $w$ iff there is a max-flow $\nu$ on $w$.*

*Proof.* Let $\rho = (q_0, i_0) \xrightarrow{w} (q_1, i_1) \xrightarrow{w} \dots (q_m, i_m)$ be a partial run of $\mathcal{A}$ on $w$ starting in the initial state. It is easy to verify, e.g. by induction the length $m$ of $\rho$, that every max-flow $\nu$ on $w$ contains $\rho$ in the sense that $q_k \in \nu(i_k)$ for all indices $0 \leq k \leq m$. This means that if $\mathcal{A}$ has a successful run on $w$, then there is no max-flow on $w$.

Next assume that $\mathcal{A}$ has no successful run on $w$. Consider the smallest max-flow $\nu$ containing all the runs of $\mathcal{A}$ on $w$. This witnesses the left-to-right direction of the proposition. □

We obtain that CMA capture complements of languages recognized by DWA:

**Theorem 5** *Given a non-deterministic DWA $\mathcal{A}$ recognizing $L$, one can construct a CMA $\mathcal{A}'$ that recognizes the complement of $L$. In particular, universality of DWA is a decidable problem.*

### 6.4 Containment and other problems

We conclude by mentioning a few interesting decidability results that follow directly from Theorems 4 and 5 and from the closure properties of CMA under union and intersection. The first result concerns the decidability of containment/equivalence of DWA. The second result concerns the property of language of being *invariant under tree encodings*, namely, of being of the form $L^{\mathsf{enc}}$ for some language $L$ of trees.

**Corollary 5** *Given two non-deterministic DWA $\mathcal{A}$ and $\mathcal{B}$, one can decide whether $\mathscr{L}(\mathcal{A}) \subseteq \mathscr{L}(\mathcal{B})$. More generally, one can decide emptiness of every Boolean combination of languages recognized by non-deterministic DWA.*

*Proof.* Let $L$ be a Boolean combination of languages recognized by non-deterministic DWA. Without loss of generality, we can assume that

$$L = \bigcup_{1 \leq i \leq k} \bigcap_{1 \leq j \leq h} \left( L_{i,j} \cap \bar{L}_{i,j} \right)$$

where each $L_{i,j}$ (resp. $\bar{L}_{i,j}$) is a language recognized by a non-deterministic DWA $\mathcal{A}_{i,j}$ (resp. the complement of a language recognized by a non-deterministic DWA $\bar{\mathcal{A}}_{i,j}$). In view of Theorems 4 and 5, one can construct suitable CMA $\mathcal{C}_{i,j}$ and $\bar{\mathcal{C}}_{i,j}$ recognizing $L_{i,j}$ and $\bar{L}_{i,j}$, respectively. Finally, closure of CMA under unions and intersections imply that $L$ is recognized by a CMA, for which emptiness can be decided. □

**Corollary 6** *Given a non-deterministic DWA $\mathcal{A}$, one can decide whether $\mathscr{L}(\mathcal{A})$ is invariant under tree encodings.*

*Proof.* We briefly explain how to reduce the problem of deciding invariance under tree encodings to a containment problem between DWA. We reuse some of the notation that we introduced in Section 4. Let $L = \mathscr{L}(\mathcal{A})$ for some non-deterministic DWA $\mathcal{A}$. We have that $L$ is invariant under tree encodings iff (i) $L \subseteq U^{\mathsf{enc}}$, where $U$ is the regular language of all trees, and (ii) for all data words $w, w'$ such that $\mathsf{tree}(w) = \mathsf{tree}(w')$, $w \in L$ iff $w' \in L$. The first condition is a simple containment between DWA. Checking the second condition reduces to transforming $\mathcal{A}$ into a TWA $\mathcal{A}^{\mathsf{tree}}$ such that $\mathscr{L}(\mathcal{A}^{\mathsf{tree}}) = \{t \mid \mathsf{enc}(t) \in L\}$, then turning $\mathcal{A}^{\mathsf{tree}}$ back to a DWA $\mathcal{A}'$ such that $\mathscr{L}(\mathcal{A}') = \mathscr{L}(\mathcal{A}^{\mathsf{tree}})^{\mathsf{enc}} = \{w \mid \mathsf{enc}(\mathsf{tree}(w)) \in L\}$ ($\supseteq \mathscr{L}(\mathcal{A})$), and finally deciding whether $\mathscr{L}(\mathcal{A}') \subseteq \mathscr{L}(\mathcal{A})$. □

## 7 Undecidable extensions of DWA

In this section we consider some natural extensions of DWA, specifically alternating DWA and DWA with pebbles, and we show that they quickly lead to undecidable emptiness problems. *Alternating DWA* are defined, as expected, by partitioning the set of states into existential and universal ones and by formulating acceptance as a winning condition in a two-player game (infinite plays are seen as rejecting runs). *Pebble DWA* are the analogue of tree walking pebble automata [6] for data words: like DWA, they can move along global/class predecessors/successors and, in addition, they can drop a pebble from a fixed finite set at a currently visited position, they can lift a pebble from the current position, and they can test whether the current position is marked with a pebble.

**Proposition 8** *Emptiness of alternating DWA is undecidable.*

*Proof.* We reduce Post's correspondence problem (PCP) to emptiness of alternating DWA. For this we consider a PCP instance that consists of a series of pairs $(u_i, v_i)$ for $i = 1, \ldots, n$, with $n > 0$ and $u_i, v_i$ words over an alphabet $\Sigma$. We introduce a new alphabet $\Gamma = \Sigma \uplus \{1, \ldots, n\} \uplus \{\#\}$ and we encode a solution $u_{i_1} \cdot \ldots \cdot u_{i_m} = v_{i_1} \cdot \ldots \cdot v_{i_m}$ ($m > 0$) of the PCP instance by means of a data word $w \in (\Gamma \times \mathbb{D})^*$, such that:

1. the projection of $w$ onto $\Gamma$ is the string $i_1 \cdot u_{i_1} \cdot \ldots i_m \cdot u_{i_m} \cdot \# \cdot i_1 \cdot v_{i_1} \cdot \ldots i_m \cdot v_{i_m}$,
2. the data value associated with # occurs exactly once, while the other data values, which are associated with symbols in $\Sigma \uplus \{1, \ldots, n\}$, occur exactly twice, once to the left and once to the right of the separator #,
3. any two positions with equal data value carry the same symbol from $\Gamma$,
4. the sequence of data values associated with symbols in $\Sigma$ (resp., in $\{1, \ldots, n\}$) occurring to the left of # coincides with the sequence of data values associated with symbols in $\Sigma$ (resp. in $\{1, \ldots, n\}$) occurring to the right of #.

Let $L$ be the language of all data word encodings of solutions of the PCP instance. Below, we show that $L$ can be recognized by an alternating DWA, which implies that the considered PCP problem reduces to non-emptiness of $L$.

The string projection of $L$ onto $\Gamma$ is a regular language of the form $\{i \cdot u_i \mid 1 \leq i \leq n\}^+ \# \{i \cdot v_i \mid 1 \leq i \leq n\}^+$. This means that the first condition that defines a data word encoding can be checked by a deterministic DWA. The second and third conditions are also easily checked by deterministic DWA with access to local types.

It remains to describe a DWA that checks the last condition by exploiting alternation. For this is sufficient to consider only the subsequence of data values associated with symbols in $\Sigma$ to the left and to the right of #. More precisely, starting from the leftmost $\Sigma$-labeled position of the input data word $w$, the automaton repeatedly performs the following sequence of moves, until the rightmost $\Sigma$-labeled position is reached: from a position $i$, it first moves universally to some $\Sigma$-labeled position $j > i$ before the occurrence of #, then it moves to the class successor $j \oplus 1$, reaches universally some $\Sigma$-labeled position $k > j \oplus 1$, and moves to the class predecessor $k \ominus 1$. If the input word $w$ is a valid encoding of a solution of the PCP instance, and in particular if $w$ satisfies condition 4. above, then the automaton eventually halts in the rightmost position of $w$. Otherwise, if $w$ does not satisfies condition 4., then there exist a position $j$ to the left of # and a position $k$ to the right of # such that $j \oplus 1 < k$ and $k \ominus 1 < j$. This means that the automaton admits an infinite run that cycles between positions $j$ and $k$, and thus rejects the input word $w$. □

**Proposition 9** *Emptiness of pebble DWA is undecidable.*

*Proof.* The proof is a variant of that of Proposition 8. Given an instance of the PCP problem, we define the language $L$ of all encodings of solutions of this instance. The data language $L$ can be equally recognized by a deterministic DWA with a single pebble. As before, the first three conditions that define membership of a data word $w$ in $L$ can be checked by deterministic DWA without pebbles, while the last condition requires the use a pebble, since it concerns the order of the data values associated with symbols in $\Sigma \uplus \{\#\}$. Specifically, if we denote by $w'$ the subsequence of $w$ obtained by selecting the positions labeled over $\Sigma \uplus \{\#\}$, then checking the last condition amounts to verifying that, in $w'$, every position $i$ to the left of # satisfies $\Big(\big((i \oplus 1) + 1\big) \ominus 1\Big) - 1 \;=\; i$. This test can be directly performed on the input data word $w$ by a deterministic automaton that executes the following steps: it places a pebble at each position $i$ to the left of #, it moves first along axis $\oplus 1$ and then to the right reaching the next $\Sigma$-labeled position (if there is no such position, it backtracks to position $i$ and accepts iff the next symbol is #); then it moves along the axis $\ominus 1$ and again to the left to the previous $\Sigma$-labeled position, where it checks the presence of the pebble (if not, the automaton halts and rejects); finally, it lifts the pebble and continues the computation with the next $\Sigma$-labeled position $i + 1$, until the separator # is reached. □

## 8 Discussion

We showed that the model of walking automaton can be adapted to data words in order to define robust families of data languages. We studied the complexity of the fundamental problems of word acceptance, emptiness, universality, and containment (quite remarkably, all these problems are shown to be decidable). We also analyzed the relative expressive power of the deterministic and non-deterministic models of Data Walking Automata, comparing them with other classes of automata appeared in the literature (most notably, Data Automata and Class Memory Automata). In this respect, we proved that deterministic DWA, non-deterministic DWA, and CMA form a strictly increasing hierarchy of data languages, where the top ones are functional projections of the bottom ones.

It follows from our results that DWA satisfy properties analogous to those satisfied by Tree Walking Automata – for instance non-deterministic DWA, like non-deterministic TWA, are effectively closed under all Boolean operations, are strictly less expressive than Tiling Automata, and are not closed under functional projections.

We also know that DWA are incomparable with one-way non-deterministic Register Automata [8]: on the one hand, DWA can check that all data values are distinct, whereas Register Automata cannot; on the other hand, Register Automata can recognize languages of data strings that do not encode valid runs of Turing machines, while Data Walking Automata cannot, as otherwise universality would become undecidable. Variants of DWA can also be considered, for instance, by adding registers, pebbles, alternation, or nesting. Unfortunately, none of these extensions yields a decidable emptiness problem. As an example, we have shown that the use of alternation or pebbles in DWA allows one to easily encode positive instances of Post's correspondence problem, thus implying undecidability of emptiness.

Finally, we leave open the following questions:

- *Are non-deterministic DWA closed under complementation?*
- *Do DWA capture all languages definable in* $\mathsf{FO}^2[\Sigma, <, \oplus 1]$*, i.e. the two-variable fragment of first-order logic with access to the letters in* $\Sigma$*, the linear order* $<$ *on positions, and the class successor predicate* $\oplus 1$*? Similarly, do DWA capture all languages definable in Basic Data LTL?*

We recall that a question similar to the first one and concerning Tree Walking Automata was left open in [2,3]. Any counterexample to closure under complementation of Tree Walking Automata would immediately give a negative answer to our first question. More generally, negative answers to our questions may come from considering the *complement* of the following language, which is definable in $\mathsf{FO}^2[\Sigma, <, \oplus 1]$ and conjectured to be not recognizable by DWA:

$$L_{\mathsf{bridges}} \;=\; \left\{ w_1\, d\, w_2\, d\, w_3\, e\, w_4\, e\, w_5\, f\, w_6\, f\, w_7 \;\middle|\; d, e, f \in \mathbb{D},\ w_1, ..., w_7 \in \mathbb{D}^* \right\}.$$

**Acknowledgments.** The first author thanks Thomas Colcombet for detailed discussions and acknowledges that some of the ideas were inspired during these. The second author acknowledges Mikołaj Bojańczyk and Thomas Schwentick for detailed discussions about the relationship between DWA and Data Automata. The authors are also grateful to the anonymous referees for the many helpful remarks on the paper.

## References

1. Björklund, H., Schwentick, T.: On notions of regularity for data languages. Theoretical Computer Science **411**(4-5), 702–715 (2010)
2. Bojańczyk, M., Colcombet, T.: Tree-walking automata cannot be determinized. Theor. Comput. Sci. **350**(2-3), 164–173 (2006)
3. Bojańczyk, M., Colcombet, T.: Tree-walking automata do not recognize all regular languages. SIAM Journal **38**(2), 658–701 (2008)
4. Bojańczyk, M., David, C., Muscholl, A., Schwentick, T., Segoufin, L.: Two-variable logic on data words. ACM Transactions on Computational Logic **12**(4), 27 (2011)
5. Bojańczyk, M., Muscholl, A., Schwentick, T., Segoufin, L.: Two-variable logic on data trees and XML reasoning. Journal of the Association for Computing Machinery **56**(3) (2009)

6. Engelfriet, J., Hoogeboom, H.: Tree-walking pebble automata. In: Jewels are forever, contributions to Theoretical Computer Science in honor of Arto Salomaa, pp. 72–83. Springer (1999)
7. Holzer, M., Kutrib, M.: Descriptional and computational complexity of finite automata: a survey. Information and Computation **209**(3), 456–470 (2011)
8. Kaminski, M., Francez, N.: Finite-memory automata. Theoretical Computer Science **134**(2), 329–363 (1994)
9. Kara, A., Schwentick, T., Zeume, T.: Temporal logics on words with multiple data values. In: Proceedings of the IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS), pp. 481–492. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2010)
10. Libkin, L., Vrgoc, D.: Regular expressions for data words. In: Proceedings of the 18th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR), *LNCS*, vol. 7180, pp. 274–288. Springer (2012)
11. Manuel, A., Zeume, T.: Two-variable logic on 2-dimensional structures. In: Proceedings of the 22th EACSL Annual Conference on Computer Science Logic (CSL), *LIPIcs*, vol. 23, pp. 484–499. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2013)
12. McNaughton, R., Papert, S.: Counter-free Automata. MIT (1971)
13. Neven, F., Schwentick, T., Vianu, V.: Finite state machines for strings over infinite alphabets. ACM Transactions on Computational Logic **5**(3), 403–435 (2004)
14. Rabin, M., Scott, D.: Finite automata and their decision problems. IBM Journal of Research and Development **3**(2), 114–125 (1959)
15. Schwentick, T., Zeume, T.: Two-variable logic with two order relations. In: Proceedings of the 19th EACSL Annual Conference on Computer Science Logic (CSL), *LNCS*, vol. 6247, pp. 499–513. Springer (2010)
16. Sipser, M.: Halting space-bounded computations. Theor. Comput. Sci. **10**, 335–338 (1980)
17. Thomas, W.: Elements of an automata theory over partial orders. In: Partial Order Methods in Verification, pp. 25–40. American Mathematical Society (1997)
18. Vollmer, H.: Introduction to Circuit Complexity: a uniform approach. Texts in Theoretical Computer Science. An EATCS Series. Springer (1999)