Synchronizing Automata over Nested Words

Dmitry Chistikov, Pavel Martyugin, Mahsa Shirmohammadi

Submission to the Journal of Automata, Languages and Combinatorics
Submitted on May 11, 2019

SYNCHRONIZING AUTOMATA OVER NESTED WORDS

Dmitry Chistikov[(A,B)] Pavel Martyugin[(C)] Mahsa Shirmohammadi[(D,E)]

[(A)] *Centre for Discrete Mathematics and its Applications (DIMAP) & Department of Computer Science, University of Warwick, United Kingdom*
d.chistikov@warwick.ac.uk

[(C)] *Institute of Mathematics and Computer Science, Ural Federal University, Ekaterinburg, Russia*
martuginp@gmail.com

[(D)] *CNRS, IRIF, Université de Paris, France*
mahsa.shirmohammadi@irif.fr

ABSTRACT

We extend the concept of a synchronizing word from deterministic finite-state automata (DFA) to nested word automata (NWA): A well-matched nested word is called synchronizing if it resets the control state of any configuration, i.e., takes the NWA from all control states to a single control state.

We show that although the shortest synchronizing word for an NWA, if it exists, can be (at most) exponential in the size of the NWA, the existence of such a word can still be decided in polynomial time. As our main contribution, we show that deciding the existence of a short synchronizing word (of at most given length) becomes PSPACE-complete (as opposed to NP-complete for DFA). The upper bound makes a connection to pebble games and Strahler numbers, and the lower bound goes via small-cost synchronizing words for DFA, an intermediate problem that we also show PSPACE-complete. We also characterize the complexity of a number of related problems, using the observation that the intersection nonemptiness problem for NWA is EXP-complete.

Keywords: synchronizing words, nested words, visibly pushdown automata, Strahler number, decision problems, formal language theory

1. Introduction

The concept of a synchronizing word for finite-state machines has been studied in automata theory for more than half a century [37, 33]. Given a deterministic finite

The conference version of this paper appeared in the proceedings of FoSSaCS 2016 [7]. The present paper contains full proofs.

[(B)] Most of this work was done while DC was affiliated with the Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern and Saarbrücken, Germany.

[(E)] Most of this work was done while MS was affiliated with the Department of Computer Science, University of Oxford, United Kingdom.

automaton (DFA) $\mathcal{D}$ over an input alphabet Σ, a word w is called *synchronizing* for $\mathcal{D}$ if, no matter which state $q \in Q$ the automaton $\mathcal{D}$ starts from, the word w brings it to some specific state $\bar{q}$ that only depends on w but not on q. Put differently, a synchronizing word *resets* the state of an automaton. If the state of $\mathcal{D}$ is initially unknown to an observer, then feeding $\mathcal{D}$ with the input w effectively restarts $\mathcal{D}$, making it possible for the observer to rely on the knowledge of the current state henceforth.

In this paper we extend the concept of a synchronizing word to so-called *nested words*. This is a model that extends usual words by imparting a parenthetical structure to them: some letters in a word are declared *calls* and *returns*, which are then matched to each other in a uniquely determined "nesting" (non-crossing) way. On the language acceptor level, this hybrid structure (linear sequence of letters with matched pairs) corresponds to a pushdown automaton where every letter in the input word is coupled with the information on whether the automaton should push, pop, or not touch the pushdown (the stack). Such machines were first studied by Mehlhorn [28] under the name of *input-driven pushdown automata* in 1980 and have recently received a lot of attention under the name of *visibly pushdown automata*. The latter term, as well as the model of nested words and *nested word automata* (in NWA the matching relation remains a separate entity, while in input-driven pushdown automata it is encoded in the input alphabet), is due to Alur and Madhusudan [3].

The tree-like structure created by matched pairs of letters occurs naturally in many domains; for instance, nested words mimic traces of programs with procedures (which have pairs of calls and returns), as well as documents in eXtensible Markup Language (XML documents, ubiquitous today, have pairs of opening and closing tags). This makes the nested words model very appealing; at the same time, nested words and NWA enjoy many nice properties of usual words and finite-state machines: for example, constructions of automata for operations over languages, and many decidability properties naturally carry over to nested words—a fact widely used in software verification (see, e.g., [12] and references therein). This suggests that the classic concept of a synchronizing word may have an interesting and meaningful extension in the realm of nested words.

Our motivation for the study of synchronizing words for NWA is threefold. First, in the view of the interest in the Černý conjecture (see [37] and discussion below) in automata theory and combinatorics, a natural research direction is a study of the length of synchronizing words for models of computation close to DFA. Our work opens this direction for the NWA model, providing a definition of synchronizing words and showing an exponential worst-case upper bound. Second, researchers in formal methods have been extending the existing algorithmic and combinatorial tools from finite-state automata to nested word automata (see, e.g., [2, 5, 12, 19, 1]), and our study of synchronizing nested words is a contribution to this effort. Finally, we fill the gap in the literature on the complexity of decision problems for infinite-state systems such as pushdown automata, register automata, Markov decision processes, Petri nets, etc. Decidability and complexity questions for many variants of these models have been studied extensively, including the work on NWA [3, 5, 2, 26] and problems on synchronizing words [22, 24, 39, 4, 11]. So it has been somewhat surprising to us that

the concept of synchronizing words for nested word automata has not been explored previously.

Our contribution and discussion

Nested word automata are essentially an expressive subclass of pushdown automata and, as such, define infinite-state transition systems (although the number of *control states* is only finite, the number of *configurations*—incorporating the state of the pushdown store—is infinite). Finding the right definition for a *synchronizing nested word* becomes for this reason a question of relevance: in the presence of infinitely many configurations not all of them may even have equal-length paths to a single designated one (this phenomenon also arises, for instance, in weighted automata [10]). In fact, any nested word w, given as input to an NWA, changes the stack height in a way that does not depend on the initial control state (and can only depend on the initial configuration if w has unmatched returns). We thus choose to define synchronizing words as those that reset the control state of the automaton and leave the pushdown store (the stack) unchanged (Definition 2; cf. location-synchronization in [10]). Consider, for instance, an XML processor [23] that does not keep a heap storage and invokes and terminates its internal procedures in lockstep with opening and closing tags in the input; our definition of a synchronizing word corresponds to an XML document that resets the local variables.

Building on this definition, we show that shortest synchronizing words for NWA can be exponential in the size of the automaton (Example 4), in contrast to the case of DFA: every DFA with n states, if it has a synchronizing word, also has one of length polynomial in n. The best known worst-case upper bound on the length of the shortest synchronizing word is $(1/6 - 125/511104) \cdot n^3 + O(n^2)$, due to Szykuła [35], and the previous record was $(n^3 - n)/6$ due to Pin [31]; Černý proved in the 1960s [36] a worst-case lower bound of $(n-1)^2$ and conjectured that this is also a valid upper bound, but as of now there is a gap between his quadratic lower bound and the cubic upper bounds of Szykuła and Pin (see [37] for a survey). In the case of nested words, the fact that shortest synchronizing words can be exponentially long comes from the repeated doubling phenomenon, typical for pushdown automata.

Although the length of a synchronizing word can be exponential, it turns out that the existence of such a word—the shortest of which, in fact, cannot be longer than exponential—can be decided in polynomial time (Theorem 6), akin to the DFA case. However, generalizing the definition in standard ways (synchronizing from a subset instead of all states, or to a subset of states instead of singletons) raises the complexity to exponential time (Theorem 15); for DFA, the complexity is polynomial space [32, 33]. The lower bounds are by reduction from the intersection nonemptiness problem, which is known to be complete for polynomial space in the case of DFA [21] and which we observe to be complete for exponential time over nested words (Lemma 14).

Our main technical contribution is characterizing the complexity of deciding existence of *short* synchronizing words, where the bound on the length is given as part of the input (written in binary). In the DFA case, this problem is **NP**-complete as shown by Eppstein [13], and for NWA it becomes **PSPACE**-complete (Theorem 7).

For the upper bound (Section 4.1) we first encode unranked trees (which represent nested words) with ranked trees. This reduces the search for a short synchronizing nested word to the search for a tree that satisfies a number of local properties. These properties, in turn, can be captured as acceptance by a certain tree automaton of exponential size. We show that guessing an accepting computation for such a machine—which amounts to guessing an exponentially large tree—can be done in polynomial space. To do this, we rely on the concept of *(black) pebbling games*, developed in the theory of computational complexity for the study of deterministic space-bounded computation (see, e.g., [34, Chapter 10]). We simulate optimal strategies for trees in such games [25], whose efficiency is determined by *Strahler numbers* [17]. Previous use of this technique in formal language theory and verification is primarily associated with derivations of context-free grammars, see, e.g., [15, 16] and [17] for a survey. In this body of work, closest to ours are apparently arguments due to Chytil and Monien [8]. Our key procedure—which can decide *bounded nonemptiness* of *succinct tree automata*—may be of use in other domains as well.

Finally, for the matching polynomial-space lower bound (Section 4.2) we construct a two-step reduction from the problem of existence of *carefully* synchronizing words for partial DFA, whose hardness is known [27]. We define an intermediate problem of *small-cost synchronization* for DFA, where every letter in the alphabet comes with a cost and the task is to decide existence of a synchronizing word whose total cost does not exceed the budget. We show that this natural problem is complete for polynomial space (this strengthens previous results from [18, 10], where costs could be state-dependent). After this, we basically simulate cost-equipped DFA with NWA, relying on the above-mentioned repeated doubling phenomenon. We find it noteworthy that this "counting" feature of nested words (more precisely, the existence of $O(\log n)$-state NWA that only accept words of length n or larger) alone is a ground for hardness.

Some of our techniques naturally extend to (going via) tree automata over ranked trees.

2. Finite-state automata and nested word automata

Words and finite-state automata. A *word* w over a finite alphabet Σ is a sequence $a_1 \cdots a_n$ of letters, whose length is $|w| = n$. We denote by Σ^* the set of all finite words over the finite alphabet Σ.

A deterministic finite-state automaton (DFA) is a tuple $\mathcal{D} = (\Sigma, Q, \Delta)$ where Σ is a finite alphabet, Q is a finite set of states, and $\Delta : Q \times \Sigma \to Q$ is the transition function. The function Δ is totally defined. It extends to finite words in a natural way: $\Delta(q, wa) = \Delta(\Delta(q, w), a)$ for all words $w \in \Sigma^*$ and letters $a \in \Sigma$; and it extends to sets of states by $\Delta(S, w) = \bigcup_{q \in S} \Delta(q, w)$ where $S \subseteq Q$. A finite-state automaton with a partially defined transition function (PFA) is a tuple $\mathcal{P} = (\Sigma, Q, \delta)$ where the transition function $\delta : Q \times \Sigma \rightharpoonup Q$ might not be defined for some states q and letters a.

Nested words and nested word automata. A *nested word* of length k over a finite alphabet Σ is a pair $u = (x, \nu)$, where $x \in \Sigma^k$ and ν is a *matching relation* of length k: a subset $\nu \subseteq \{-\infty, 1, \ldots, k\} \times \{1, \ldots, k, +\infty\}$ such that, first, if $\nu(i, j)$

holds, then $i < j$; second, for $1 \leq i \leq k$ the set $\mu(i) \stackrel{\text{def}}{=} \{j \mid \nu(i,j) \text{ or } \nu(j,i)\}$ contains at most one element; third, whenever $\nu(i,j)$ and $\nu(i',j')$, it cannot be the case that $i < i' \leq j < j'$. We assume that $\nu(-\infty, +\infty)$ never holds.

If $\nu(i,j)$, the position i in the word u is said to be a *call*, and the position j a *return*. All positions from $\{1,\ldots,k\}$ that are neither calls nor returns are *internal*. A call (a return) i is *matched* if ν matches it to an element of $\{1,\ldots,k\}$, i.e., if we have $\mu(i) \cap \{1,\ldots,k\} \neq \emptyset$, and *unmatched* otherwise. We shall call a nested word *well-matched* if it has no unmatched calls and no unmatched returns.

Define a *nested word automaton* (an NWA) over the input alphabet Σ as a structure $\mathcal{A} = (\Sigma, Q, \Gamma, \delta, q_0, \gamma_0)$, where:

- Σ is a finite alphabet,
- Q is a finite non-empty set of control states,
- Γ is a finite non-empty set of stack symbols,
- $\delta = (\delta^{\mathsf{call}}, \delta^{\mathsf{int}}, \delta^{\mathsf{ret}})$, where
  - $\delta^{\mathsf{int}} \colon Q \times \Sigma \to Q$ is an internal transition function,
  - $\delta^{\mathsf{call}} \colon Q \times \Sigma \to Q \times \Gamma$ is a call transition function,
  - $\delta^{\mathsf{ret}} \colon \Gamma \times Q \times \Sigma \to Q$ is a return transition function,
- $q_0 \in Q$ is the initial control state, and
- $\gamma_0 \in \Gamma$ is the initial stack symbol.

A *configuration* of $\mathcal{A}$ is a pair $(q,s) \in Q \times \Gamma^*$. We write $(q,s) \xrightarrow{u} (q',s')$ for a nested word u if the following conditions hold. First suppose $u = (x,\nu)$ has length 1, then:

- if 1 is an internal position in the word u, then $\delta^{\mathsf{int}}(q,x) = q'$ and $s' = s$;
- if 1 is a call, then $\delta^{\mathsf{call}}(q,x) = (q',\gamma)$ and $s' = s\gamma$ for some $\gamma \in \Gamma$;
- if 1 is a return, then:
  - either $\delta^{\mathsf{ret}}(\gamma,q,x) = q'$ and $s = s'\gamma$,
  - or $\delta^{\mathsf{ret}}(\gamma_0,q,x) = q'$ and $s = s' = \varepsilon$.

(The last item means that, if the stack is empty, $s = \varepsilon$, the NWA can take a return transition using the "default" initial stack symbol, γ_0.) Now take as $\longrightarrow$ the reflexive transitive closure of the union of $\xrightarrow{u}$ over all nested words u of length 1; these input words on top of the arrow are concatenated accordingly.

Alternatively, nested words can be seen as words over an extended alphabet. Let $\langle\Sigma$ and $\Sigma\rangle$ be disjoint copies of Σ that contain letters of the form $\langle a$ and $a\rangle$, respectively, for every $a \in \Sigma$. Then any nested word over Σ is associated with a word over the *nested alphabet* $\langle\Sigma \cup \Sigma \cup \Sigma\rangle$. Conversely, every word w over this nested alphabet is unambiguously associated with a matching relation ν_w of length $|w|$ where positions with elements of $\langle\Sigma$, Σ, and $\Sigma\rangle$ are calls, internal positions, and returns, respectively; the word w can thus be identified with a nested word $(\pi(w), \nu_w)$ where π projects letters back to Σ. The automaton $\mathcal{A}$ can then be viewed as an ε-free pushdown automaton over the nested alphabet $\langle\Sigma \cup \Sigma \cup \Sigma\rangle$ in which the direction of stack operations (i.e., whether the automaton pushes, pops, or does not touch the stack) is determined by whether the current position belongs to $\langle\Sigma$, Σ, or $\Sigma\rangle$. Such automata are known under the names *input-driven pushdown automata* and *visibly pushdown*

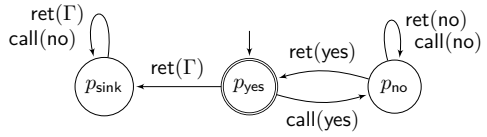

Figure 1: NWA $\mathcal{W}$ that *accepts* the set of all well-matched words. (The set of final states is $\{p_{\mathsf{yes}}\}$.) Since, on all letters, call and return transitions behave the same way, call transitions $\delta^{\mathsf{call}}(q,a) = (q',\gamma)$ are shown as $q \xrightarrow{\mathsf{call}(\gamma)} q'$, and return transitions $\delta^{\mathsf{ret}}(\gamma,q,a) = q'$ as $q \xrightarrow{\mathsf{ret}(\gamma)} q'$. All transitions not shown are self-loops.

automata. A *path* (*run*, *computation*) of $\mathcal{A}$ over an input word $u = a_1 \dots a_k$, where each $a_i \in \langle \Sigma \cup \Sigma \cup \Sigma \rangle$, is a sequence $(p_0, s_0) \dots (p_k, s_k)$ of configurations such that for all $0 < i \leq k$ we have $(p_{i-1}, s_{i-1}) \xrightarrow{a_i} (p_i, s_i)$.

Example 1. Let Σ be a finite alphabet. We define an NWA $\mathcal{W}$ over this alphabet as follows. It has three states p_{yes}, p_{no} and p_{sink}. The state p_{yes} is the only initial state, and the state p_{sink} is absorbing. There are two stack symbols $\Gamma = \{\mathsf{yes}, \mathsf{no}\}$; see Figure 1. The transitions in $\mathcal{W}$ are such that, for all $a \in \Sigma$:

- all internal transitions are self-loops: $\delta^{\mathsf{int}}(p,a) = p$ where $p \in \{p_{\mathsf{yes}}, p_{\mathsf{no}}, p_{\mathsf{sink}}\}$,
- the call transition in p_{yes} and p_{no} always lead to p_{no}:

$$\delta^{\mathsf{call}}(p_{\mathsf{yes}}, a) = (p_{\mathsf{no}}, \mathsf{yes}) \text{ and } \delta^{\mathsf{call}}(p_{\mathsf{no}}, a) = (p_{\mathsf{no}}, \mathsf{no}),$$

- the return transitions in p_{yes} go to p_{sink}: $\delta^{\mathsf{ret}}(\Gamma, p_{\mathsf{yes}}, a) = p_{\mathsf{sink}}$. However, the return transitions in p_{no} depends on the stack symbol:

$$\delta^{\mathsf{ret}}(\mathsf{no}, p_{\mathsf{no}}, a) = p_{\mathsf{no}} \text{ and } \delta^{\mathsf{ret}}(\mathsf{yes}, p_{\mathsf{no}}, a) = p_{\mathsf{yes}};$$

- all transitions in p_{sink} are self-loops.

Observe that the NWA $\mathcal{W}$ satisfies $(p_{\mathsf{yes}}, \varepsilon) \xrightarrow{u} (p_{\mathsf{yes}}, \varepsilon)$ for every well-matched word u; moreover, all runs that start from the state p_{yes} and end in the same state are over well-matched nested words too.

In decision problems that we study in this paper, the *size* of an automaton is taken to be proportional to $|\Sigma| \cdot |Q| \cdot |\Gamma|$.

3. Synchronizing words

A word w is *synchronizing* for a DFA $\mathcal{D}$ if there exists some state $\bar{q} \in Q$ such that $\Delta(Q, w) = \{\bar{q}\}$. The *synchronizing word problem* for DFA asks, given a DFA $\mathcal{D}$, whether there exists some synchronizing word for $\mathcal{D}$. The synchronizing problem for DFA is in **NL** by a simple technique which we call *pairwise synchronization*. Given $\mathcal{D} = (\Sigma, Q, \Delta)$ with n states, there is a synchronizing word for $\mathcal{D}$ iff for every pair of states $q, q' \in Q$ there is a word v such that $\Delta(q, v) = \Delta(q', v)$ (see [37, 33]

for more details). This condition is easily checked in **NL** (reachability in the product automaton); the proof of correctness is an exercise for the reader.

Synchronizing words for PFA are called *carefully synchronizing words* [27]. Such a word for a PFA $\mathcal{P} = (\Sigma, Q, \delta)$ uses only defined transitions in δ; this means that the word $w = a_1 a_2 \cdots a_n$ is synchronizing for $\mathcal{P}$ if

- $\delta(q, a_1)$ is defined for all states $q \in Q$,
- $\delta(q, a_{i+1})$ is defined for all states $q \in \delta(Q, a_1 a_2 \cdots a_i)$ and all $1 \leq i < n$, and
- $\delta(Q, w)$ is a singleton.

The *carefully synchronizing word* problem asks, given a PFA $\mathcal{P}$, whether there exists some carefully synchronizing word for $\mathcal{P}$. This problem is known to be **PSPACE**-complete [27]. The membership in **PSPACE** is by reachability in the subset construction, so every PFA with a synchronizing word has one with length at most exponential in the size of the PFA.

3.1. Synchronizing nested words

Informally, we call a well-matched nested word u synchronizing for an NWA $\mathcal{A}$ if it takes $\mathcal{A}$ from all control states to some single control state. Note that the result of feeding any well-matched word to an NWA does not depend on the stack contents; furthermore, if $(q_1, s_1) \xrightarrow{u} (q_2, s_2)$ and u is well-matched, then $s_1 = s_2$. This lets us extend the definition of $\longrightarrow$ to sets of states: we write $(Q_1, s) \xrightarrow{u} (Q_2, s)$ if, first, the word u is well-matched, second, for all $q_1 \in Q_1$ there exists a $q_2 \in Q_2$ such that $(q_1, s) \xrightarrow{u} (q_2, s)$, and, third, for every state $q_2 \in Q_2$ there exists a $q_1 \in Q_1$ such that $(q_1, s) \xrightarrow{u} (q_2, s)$. If $Q_i = \{q_i\}$, we write (q_i, s) instead of $(\{q_i\}, s)$.

Definition 2. Let $\mathcal{A} = (\Sigma, Q, \Gamma, \delta, q_0, \gamma_0)$ be an NWA. A well-matched nested word u is *synchronizing* for $\mathcal{A}$ if there exists a control state $\bar{q} \in Q$ such that the relation $(Q, \varepsilon) \xrightarrow{u} (\bar{q}, \varepsilon)$ holds.

By the observation above, u is synchronizing if and only if there exists a $\bar{q} \in Q$ such that for all $q \in Q$ and for all $s \in \Gamma^*$ the relation $(q, s) \xrightarrow{u} (\bar{q}, s)$ holds.

Remark 3. Definition 2 crucially relies on the nested structure of the input word, in that this structure determines the stack behaviour of the NWA. Extending this definition to the general case of pushdown automata (PDA) would face the difficulties outlined in the introduction; to the best of our knowledge, no such extension has been proposed to date. The term "synchronization" in the context of PDA is known to be used when referring to the agreement between the transitions taken by the automaton and an external structure [6]: in NWA, for example, input symbols and stack actions are synchronized (in this sense).

Example 4. Given $n \geq 1$, we construct an NWA $\mathcal{A}_n$ with $O(\log n)$ control states and $O(1)$ stack symbols such that the shortest synchronizing word for $\mathcal{A}_n$ has length exactly n.

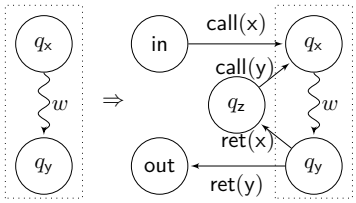

Figure 2: Doubling transformation

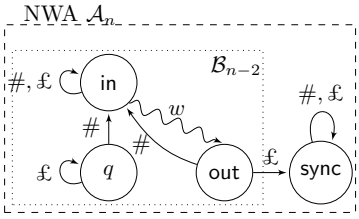

Figure 3: NWA $\mathcal{A}_n$ based on $\mathcal{B}_{n-2}$

Our construction is inductive. We first construct a family of *incomplete* NWA $\mathcal{B}_n$ with stack symbols $\{\mathsf{x}, \mathsf{y}\}$ and two designated states q_{x} and q_{y}. In $\mathcal{B}_n$, the shortest run from q_{x} to q_{y} is driven by some well-matched nested word w of length n, and along this run the state q_{y} is not visited. These NWA will be incomplete in the sense that their transition functions will only be partial; redirecting all missing transitions to the initial state in would make these NWA complete.

For each n, given $\mathcal{B}_n$, we construct NWA $\mathcal{B}_{2n+4}$ and $\mathcal{B}_{2n+5}$ where the length of the shortest run between two new states in and out is exactly $2n + 4$ and $2n + 5$, respectively. The construction of $\mathcal{B}_{2n+4}$ is depicted in Figure 2. Here the shortest run from in to out is over $\mathsf{call}(\mathsf{x}) \cdot w \cdot \mathsf{ret}(\mathsf{x}) \cdot \mathsf{call}(\mathsf{y}) \cdot w \cdot \mathsf{ret}(\mathsf{y})$—see Remark 5 below for details on our call and return notation—and has length $2n + 4$; splitting the state q_{z} into two states, with an internal transition pointing from one to the other, gives us $\mathcal{B}_{2n+5}$. We call this transformation *doubling*. For all $n \geq 4$ the NWA $\mathcal{B}_n$ can be constructed by several doubling transformations starting from one of the automata $\mathcal{B}_0, \mathcal{B}_1, \mathcal{B}_2, \mathcal{B}_3$ (which are simply DFA with 1, 2, 3, and 4 states, respectively, each comprising a chain of states with transitions linking them). The size of $\mathcal{B}_n$ is $O(\log n)$.

For all $n \geq 2$, from the NWA $\mathcal{B}_{n-2}$ we construct an NWA $\mathcal{A}_n$ where the shortest synchronizing word has length exactly n. Figure 3 shows the sketch of the construction: there are two new letters # and £ and a new absorbing state sync. From all states q of $\mathcal{B}_{n-2}$, the letter # resets the NWA to in whereas £-transitions are all self-loops except in the state out where $\mathsf{out} \xrightarrow{£} \mathsf{sync}$. All missing transitions are directed to the state in (note that even in the case of DFA, existence of synchronizing words in the presence of *partial* transition functions is **PSPACE**-complete [27]; it is thus essential that our NWA be complete). Observe that the shortest synchronizing word has length exactly n; it is $\# \cdot w \cdot £$ where w is the shortest word that takes $\mathcal{B}_{n-2}$ from in to out.

Remark 5. Our Example 4 seems to use a "non-uniform" set of call, return, and internal symbols, but this is easily remedied by making some of the symbols indistinguishable. That is, for all call positions in the input words, we write the corresponding letter simply as call, and, similarly, for all return positions it is ret. In figures as well as in the text, the NWA will typically have input alphabet Σ, and for all call and return positions these NWA will make no distinction between different symbols from Σ being read. Finally, the letter in parentheses will usually refer to the stack symbol being pushed or popped.

Our following Theorem 6 extends a characterization of synchronizing words from DFA: an NWA $\mathcal{A}$ has a synchronizing word if and only if for every pair of states p, q there exists a well-matched word u that synchronizes this pair, i.e., there exists some $\bar{q}$ such that $(\{p, q\}, \varepsilon) \xrightarrow{u} (\bar{q}, \varepsilon)$.

Theorem 6. *If an NWA $\mathcal{A}$ has a synchronizing word, then it has one of length at most exponential in the size of $\mathcal{A}$. Moreover, the existence of a synchronizing word can be decided in time polynomial in the size of $\mathcal{A}$.*

Proof. Suppose an NWA $\mathcal{A}$ has a synchronizing word. Then for every pair $p, q \in Q$, where Q is the set of states of $\mathcal{A}$, there exists a well-matched word that is *accepted* by the product automaton $\mathcal{A}(p) \times \mathcal{A}(q) \times \mathcal{W}$, where by $\mathcal{A}(p)$ and $\mathcal{A}(q)$ we denote disjoint copies of $\mathcal{A}$ with initial states p and q respectively, and $\mathcal{W}$ is the NWA defined in Example 1 that only *accepts* well-matched words. As usual, a word is said to be accepted by an automaton if it brings it to some accepting state (where the set of such states is defined in advance). In this product automaton, accepting are all states of the form $(r, r, \bar{q})$ where $r \in Q$ is arbitrary and $\bar{q}$ is accepting in $\mathcal{W}$. Every synchronizing word is obviously accepted by the product automaton; moreover, since this automaton has polynomial size, translating it to a context-free grammar shows that the automaton accepts at least one word, say $w_{p,q}$, of at most exponential size (in the size of $\mathcal{A}$).

Now observe that the task of synchronizing runs of $\mathcal{A}$ that start in different states $q_1, \ldots, q_n \in Q$ can be performed pairwise: place n tokens on states of $\mathcal{A}$, pick any pair of them, say on p and q, and to feed the machine with $w_{p,q}$. Now this pair of tokens is glued together; all tokens move some new locations, but their number is now $n-1$. Repeating the procedure another $n-2$ times then gives a synchronizing word for $\mathcal{A}$.

It only remains to note that the argument above also gives a sufficient (not just necessary) condition for the existence of synchronizing words: indeed, if all product automata $\mathcal{A}(p) \times \mathcal{A}(q) \times \mathcal{W}$ have nonempty languages, then a synchronizing word can be constructed as described above, otherwise no such word can exist. Since emptiness for NWA (and for pushdown automata in general) is decidable in polynomial time, the theorem follows. □

4. Short synchronizing nested words

In this section we prove our main result, characterizing the complexity of deciding the existence of synchronizing nested words with length less than a given bound. The corresponding decision problem for DFA is **NP**-complete [13]: Given a DFA $\mathcal{D}$ and an integer $\ell \geq 1$ written in unary, decide if $\mathcal{A}$ has a synchronizing word u of length at most ℓ. Note that deciding if the shortest synchronizing word has length exactly ℓ, a related but different problem, is **DP**-complete [29].

Since any DFA with a synchronizing word has one of length cubic in its size, it does not matter for DFA if the given bound ℓ is written in binary or in unary. In contrast, as our Example 4 shows, NWA may need an exponentially long word for

synchronization. For this reason, we focus on the existence of synchronizing nested words with length less than ℓ, where ℓ is written in binary. In the alternative version, i.e., if ℓ is written in unary, the problem is **NP**-complete: the upper bound is a guess-and-check argument, and hardness already holds for DFA.

Theorem 7. *The following problem, the Short Synchronizing Nested Word problem, is* **PSPACE***-complete: Given an NWA $\mathcal{A}$ and an integer $\ell \geq 1$ written in binary, decide if $\mathcal{A}$ has a synchronizing word u of length at most ℓ.*

4.1. Membership in **PSPACE**

In this subsection, we show that the Short Synchronizing Nested Word problem is in **PSPACE**. In fact, we can also adjust our arguments (see subsubsection 4.1.2) so that they give a **PSPACE** upper bound for another problem: Given a NWA $\mathcal{A}$, two subsets of its control states $I, F \subseteq Q$, and an integer $\ell \geq 1$ written in binary, decide if there exists a well-matched word of length at most ℓ that takes all states in I into F.

The plan of the proof is as follows. We encode well-matched nested words using binary trees (subsubsection 4.1.1), so that runs of NWA correspond to computations of tree automata and synchronizing words to tuples of such computations (subsubsection 4.1.2). Thus the task of guessing a short synchronizing word is reduced to the task of guessing an accepting computation of a tree automaton on an unknown binary tree of potentially exponential size (Lemma 9); this is the same as guessing an exponentially large binary tree subject to local conditions. We prove that it is possible to solve this *bounded nonemptiness* problem in polynomial space, even if the tree automaton in question has exponentially many states and is only given in symbolic form (subsubsection 4.1.4); our solution relies on the concepts of pebble games and Strahler numbers (subsubsection 4.1.3).

4.1.1. Binary tree representation of nested words

In this subsubsection we describe a representation of nested words with binary trees used in the sequel.

Nested words as binary trees

We denote the binary *tree representation* of a nested word u by $\mathsf{bin}(u)$. The binary tree $\mathsf{bin}(u)$ has nodes of several different *types*. We do not attempt to minimize the number of these types; different representations are, of course, also possible.

Type	Degree	Notes
call-return binary	2	Associated with matched pair $\langle x_i, x_j \rangle$
auxiliary binary	2	*Corresponds* to positions $i < j$
call-return unary	1	Associated with matched pair $\langle x_i, x_j \rangle$
call-return leaf	0	Associated with matched pair $\langle x_i, x_j \rangle$, $j = i + 1$
internal leaf	0	Associated with internal letter x_i

We denote the set of types by Types; each type comes with a fixed degree, which is simply the number of children of a node. Note that auxiliary binary nodes are not associated with any letters in the nested word, although they do correspond to pairs of positions in it.

Consider what happens if we execute the left-to-right depth-first traversal on the tree $\mathsf{bin}(u)$ and *spell* the letters associated with the nodes in the natural way. Specifically, at any call-return node v *associated* with a matched pair $\langle x_i, x_j \rangle$, $i < j$, spell "$\langle x_i$" when entering and "$x_j \rangle$" when leaving the subtree rooted at v; at any internal leaf associated with i, spell "x_i". The traversal of the entire tree $\mathsf{bin}(u)$ spells the word u, and every subtree spells some well-matched factor.

Claim 1. *For any nested word u of length ℓ its binary tree representation $\mathsf{bin}(u)$ has at most $2\ell - 1$ nodes. Moreover, if $\mathsf{bin}(u) = \mathsf{bin}(u')$, then $u = u'$.*

More details on the binary tree representation of nested words are given in Subsubsection 4.1.5.

Trees as terms over a ranked alphabet

We now switch the perspective a little and look at binary tree representations as terms. Indeed, pick the ranked alphabet

$$\mathcal{F} \subseteq \mathsf{Types} \times (\langle \Sigma \times \Sigma \rangle \cup \Sigma \cup \{\varepsilon\}) \tag{1}$$

as follows. All elements of $\mathcal{F}$ have *rank* 0, 1, or 2, according to their first (that is, Types-) component; the rank is simply the admissible number of children (i.e., the degree). The second component stores the associated letter or pair of letters, if any; the value ε corresponds to the undefined association mapping. Since the Types-component already determines whether the second component should carry a pair of call and return letters, a single letter, or ε, we only take valid combinations into $\mathcal{F}$.

As this term representation is essentially the same as the binary representation defined above, we shall denote it by the same symbol $\mathsf{bin}(u)$; that is, $\mathsf{bin}(u)$ is a term over $\mathcal{F}$ for any non-empty well-matched word u. In what follows, we will mostly refer to $\mathsf{bin}(u)$ as a tree but treat it as a term.

4.1.2. From nested word automata to tree automata

From runs of NWA to runs of tree automata

Recall the definition of a *nondeterministic tree automaton* over a ranked alphabet $\mathcal{F}$ (see, e.g., [9]): such an automaton is a tuple $\mathcal{T} = (\mathcal{F}, \mathcal{Q}, \mathcal{Q}^{\mathsf{f}}, \Delta)$ where $\mathcal{Q}$ is a finite set of states, $\mathcal{Q}^{\mathsf{f}} \subseteq \mathcal{Q}$ is a set of final states, and Δ is a set of transition rules. These rules have the form $f(q_1, \ldots, q_r) \mapsto q$ where $q, q_1, \ldots, q_r \in \mathcal{Q}$ and $r \geq 0$ is the rank of the symbol $f \in \mathcal{F}$; nondeterminism of $\mathcal{T}$ means that Δ can contain several rules with identical left-hand sides.

The semantics of tree automata is defined in the following manner. For any tree t over the ranked alphabet $\mathcal{F}$, we assign to any node v of t a state $q \in \mathcal{Q}$ inductively,

phrasing it as "the subtree t_v rooted at v *evaluates* to the state q" (as the automaton is nondeterministic, the same subtree may evaluate to several different states). The inductive assertion is that if f is the label of v, the subtree t_v evaluates to q, and its principal subtrees evaluate to $q_1, \ldots, q_r$, then the transition $f(q_1, \ldots, q_r) \mapsto q$ appears in Δ. The entire tree t is *accepted* if the root of t evaluates to some final state $\bar{q} \in \mathcal{Q}^{\mathsf{f}}$.

Lemma 8. *For any NWA $\mathcal{A}$ with states Q and for all pairs $\bar{p}, \bar{q} \in Q$, there exists a tree automaton $\mathcal{T}(\bar{p}, \bar{q})$ over the ranked alphabet $\mathcal{F}$ as in Equation* (1) *that has the following property: $\mathcal{T}(\bar{p}, \bar{q})$ accepts a tree $\mathsf{bin}(u)$ if and only if the NWA $\mathcal{A}$ has a run on u that starts in state $\bar{p}$ and ends in state $\bar{q}$. Moreover, $\mathcal{T}(\bar{p}, \bar{q})$ can be constructed from $\mathcal{A}$ in time polynomial in the size of $\mathcal{A}$.*

Proof. To construct the tree automaton $\mathcal{T}(\bar{p}, \bar{q}) = (\mathcal{F}, \mathcal{Q}, \mathcal{Q}^{\mathsf{f}}, \Delta)$ we use the following idea. States of $\mathcal{T}(\bar{p}, \bar{q})$ will be *summaries*, $\mathcal{Q} = Q^2$, and subtrees will be evaluated to summaries (p, q) so that the following condition holds. Take any subtree t_v of the input tree t; as discussed in subsubsection 4.1.1, the left-to-right DFS traversal of this subtree spells a word which is a well-matched factor u' of u. The automaton $\mathcal{T}(\bar{p}, \bar{q})$ will pick the states p, q in such a way that the NWA $\mathcal{A}$ will start and finish traversing u' in the states p and q respectively. (Naturally, nondeterministic guessing will be required for $\mathcal{T}$ to pick these pairs of states correctly.)

We now make the details of the construction more precise. Define a *labeling* of t with respect to $\mathcal{A}$ as a function of the form $\lambda\colon V(t) \to Q^2$ where $V(t)$ denotes the set of nodes of t (or, equivalently, the set of ranked symbols in the term representation of t). The labeling λ is *consistent* with the NWA $\mathcal{A}$ if the following conditions are satisfied for all nodes v of t (we assume $\lambda(v) = (p, q)$ and $\lambda(v_s) = (p_s, q_s)$ for $s = 1, 2$).

(I) If v is a call-return binary node associated with a matched pair $\langle x_i, x_j \rangle$ and v_1 and v_2 are its left and right children, then there exists some $\gamma \in \Gamma$ such that $\delta^{\mathsf{call}}(p, x_i) = (p_1, \gamma)$, $q_1 = p_2$, and $\delta^{\mathsf{ret}}(\gamma, q_2, x_j) = q$.

(II) If v is an auxiliary binary node and v_1 and v_2 are its left and right children, then $p = p_1$, $q_1 = p_2$, and $q_2 = q$.

(III) If v is a call-return unary node associated with a matched pair $\langle x_i, x_j \rangle$ and v_1 is its only child, then there exists a $\gamma \in \Gamma$ such that $\delta^{\mathsf{call}}(p, x_i) = (p_1, \gamma)$ and $\delta^{\mathsf{ret}}(\gamma, q_1, x_j) = q$.

(IV) If v is a call-return leaf associated with a matched pair $\langle x_i, x_j \rangle$, $j = i + 1$, then there exists a $\gamma \in \Gamma$ and an $r \in Q$ where $\delta^{\mathsf{call}}(p, x_i) = (r, \gamma)$ and $\delta^{\mathsf{ret}}(\gamma, r, x_j) = q$.

(V) If v is an internal leaf associated with internal letter x_i, then $\delta^{\mathsf{int}}(p, x_i) = q$.

Suppose $\lambda(\mathsf{root}(t)) = (\bar{p}, \bar{q})$ where $\mathsf{root}(t)$ is the root of t. Start the NWA $\mathcal{A}$ in the state p_0 and run the left-to-right DFS traversal of t; whenever the traversal spells a letter, give it to $\mathcal{A}$ as input. Now λ is consistent with $\mathcal{A}$ if and only if for every non-root node v with $\lambda(v) = (p, q)$ the NWA begins the computation on the corresponding well-matched factor in state p and leaves it in state q. Therefore $\mathcal{A}$ has a computation on u that starts in state $\bar{p}$ and terminates in state $\bar{q}$ if and only if there exists a consistent labeling λ of t such that $\lambda(\mathsf{root}(t)) = (\bar{p}, \bar{q})$.

It remains to define the transitions of $\mathcal{T}(\bar{p}, \bar{q})$ in such a way that $\mathcal{T}(\bar{p}, \bar{q})$ guesses a

consistent labeling of $\mathsf{bin}(u)$. At a high level, the existence of an appropriate set Δ follows from our definition of a consistent labeling. Indeed, a formal definition of these transitions follows the list in the definition of consistency above. For example, in line with item I, for every $a, b \in \Sigma$, if f denotes the call-return binary pair $\langle a, b\rangle$, then Δ contains all transitions of the form $f((p_1, q_1), (p_2, q_2)) \mapsto (p, q)$ for which there exists a $\gamma \in \Gamma$ such that $\delta^{\mathsf{call}}(p, a) = (p_1, \gamma)$, $q_1 = p_2$, and $\delta^{\mathsf{ret}}(\gamma, q_2, b) = q$. Similarly, in line with item IV, for every $a, b \in \Sigma$, if f denotes the call-return leaf $\langle a, b\rangle$, then Δ contains all transitions of the form $f() \mapsto (p, q)$ for which there exists a $\gamma \in \Gamma$ and an $r \in Q$ such that $\delta^{\mathsf{call}}(p, a) = (r, \gamma)$ and $\delta^{\mathsf{ret}}(\gamma, r, b) = q$. Other cases are considered analogously. This completes the proof. □

Synchronizing words and implicitly presented tree automata

We can now return to the synchronizing word problem. Suppose $\mathcal{A}$ is an NWA with states Q; now a well-matched nested word u is a synchronizing word for $\mathcal{A}$ if and only if there is a state $\bar{q} \in Q$ such that for all i the tree $\mathsf{bin}(u)$ is accepted by the automaton $\mathcal{T}(q_i, \bar{q})$; here we assume $Q = \{q_1, \ldots, q_n\}$. The following statement rephrases this condition in terms of *products* of tree automata (the definition is standard; see, e.g., [9, Section 1.3]).

Lemma 9. *Let $\mathcal{A}$ be an NWA with states $Q = \{q_1, \ldots, q_n\}$. Take an arbitrary $\ell \in \mathbb{N}$. For $\bar{q} \in Q$, consider the product automaton $\mathfrak{A}_{\bar{q}} = \mathcal{T}(q_1, \bar{q}) \times \ldots \times \mathcal{T}(q_n, \bar{q}) \times \mathcal{N}_\ell$ where $\mathcal{N}_\ell$ is a tree automaton that only depends on ℓ and Σ and accepts the set of trees of the form $\mathsf{bin}(u)$ where the nested word u has length at most ℓ. The NWA $\mathcal{A}$ has a synchronizing word of length at most ℓ iff there exists $\bar{q} \in Q$ such that $\mathfrak{A}_{\bar{q}}$ accepts at least one tree over $\mathcal{F}$.*

Note that the set of states of $\mathfrak{A}_{\bar{q}}$, which we denote by $\mathfrak{Q}$, is, in general, exponential in the size of $\mathcal{A}$. Note, however, that (*i*) each state has a representation—as a tuple of n states of $\mathcal{T}(q_i, \bar{q})$ and a state of $\mathcal{N}_\ell$—polynomial in the size of $\mathcal{A}$ and ℓ and, moreover, that (*ii*) the following problems can be decided in **PSPACE** (and, in fact, in **P**, although we do not need to rely on this):

(a) given a state $\mathfrak{q} \in \mathfrak{Q}$, decide if $\mathfrak{q}$ is a final state of $\mathfrak{A}_{\bar{q}}$;

(b) given a symbol $f \in \mathcal{F}$ of rank r and states $\mathfrak{q}, \mathfrak{q}_1, \ldots, \mathfrak{q}_r \in \mathfrak{Q}$, decide if $f(\mathfrak{q}_1, \ldots, \mathfrak{q}_r) \mapsto \mathfrak{q}$ is a transition in $\mathfrak{A}_{\bar{q}}$.

We emphasize that the complexity bounds in these properties are given with respect to the size of $\mathcal{A}$ and ℓ, i.e., assuming that $\mathcal{A}$ and ℓ (and not $\mathfrak{A}_{\bar{q}}$!) are given as input. We will use these properties (*i*) and (*ii*) in subsubsection 4.1.4; for brevity, we shall simply say that $\mathfrak{A}_{\bar{q}}$ is *implicitly presented in polynomial space*.

Claim 2. *The automaton $\mathfrak{A}_{\bar{q}}$ from Lemma 9 is implicitly presented in polynomial space and does not accept any tree with more than $2\ell - 1$ nodes.*

The second part of the claim follows from Claim 1 in subsubsection 4.1.1.

4.1.3. Pebble games and Strahler numbers

In this subsubsection we recall a classic idea that we use in the proof of Lemma 10.

An instance of the (*black*) *pebble game* (see, e.g., [34, Chapter 10]) is defined on a directed acyclic graph, G. The game is one-player; the player sees the graph G and has access to a supply of *pebbles*. The game starts with no pebbles on (vertices of) the graph. A *strategy* in the game is a sequence of moves of the following kinds:

(a) if all immediate predecessors of a vertex v have pebbles on them, put a pebble on (or move one of these pebbles to) v;

(b) remove a pebble from a vertex v.

Note that for any source v of G, the pre-condition for the move of the first kind is always satisfied. The strategy is *successful* if during its execution every sink of G carries a pebble at least once; the strategy is said to *use* k pebbles if the largest number of pebbles on G during its execution is k. The (*black*) *pebbling number* of G, denoted $\mathsf{peb}(G)$, is the smallest k for which there exists a successful strategy for G using k pebbles.

The black pebbling number captures space complexity of deterministic computations [30, 20]. Intuitively, think of G as a circuit, where sources are circuit inputs and sinks are circuit outputs; nodes with nonzero fan-in are gates that compute functions of their immediate predecessors. A strategy corresponds to computing the value of the circuit using auxiliary memory: *pebbling* a vertex (i.e., putting a pebble on it) corresponds to computing the value of the gate and storing it in memory; removing a pebble from the vertex corresponds to removing it from the memory. The pebbling number is thus (an abstraction of) the minimal amount of memory required to compute the value of the circuit.

Consider the case where the graph is a tree, $G = t$, with all edges directed towards the root; this corresponds to formulas, say arithmetic expressions [14]. For trees, the pebbling number can be computed inductively [25]: if t is a single-vertex tree, then $\mathsf{peb}(G) = 1$; suppose t has principal subtrees $t_1, \ldots, t_d$ (recall that principal subtrees are subtrees rooted at children of the root of a given tree) and

$$\mathsf{peb}(t_1) \geq \mathsf{peb}(t_2) \geq \ldots \geq \mathsf{peb}(t_d),$$

then $\mathsf{peb}(t) = \max(\mathsf{peb}(t_i) + i - 1)$ over $1 \leq i \leq d$. For binary trees (where all vertices have fan-in at most two, $d \leq 2$) the pebbling number (under different names) has been studied independently and rediscovered multiple times (although, to the best of our knowledge, no connection with the literature on pebbling games has ever been pointed out), see [14, 17]. The value $\mathsf{peb}(t) - 1$ is usually called the *Strahler number* of the tree t and is also known, e.g., as the Horton–Strahler number and as tree dimension; this is the largest h such that t has a complete binary tree of height h as a minor.

In the sequel, we choose to talk about Strahler numbers but use the connection to pebble games. The key observation, following from the above-mentioned characterization or from the recurrence in the previous paragraph, is that the Strahler number of an m-node tree does not exceed $\lceil \log_2(m+1) \rceil - 1$ (this bound is tight). This value corresponds to the pebbling strategy that, before pebbling any vertex v of indegree 2,

first (*i*) recurses into the subtree with the larger Strahler number; (*ii*) places (inductively) a pebble on its root and removes all other pebbles from this subtree; and then (*iii*) recurses into the other subtree. We will use this strategy in the following subsubsection.

4.1.4. Bounded nonemptiness for implicitly presented tree automata

Here we combine the ideas from subsubsections 4.1.2 and 4.1.3 to prove the upper bound in Theorem 7.

To do so, we first prove Lemma 10 showing **PSPACE** membership for *bounded nonemptiness* problem in implicitly presented tree automata. This problem asks, given a tree automaton that is implicitly presented in polynomial space and a number m written in binary, whether the automaton accepts some tree with at most m nodes. It is crucial that m constitute part of the input, because for *explicitly* presented tree automata the (non-)emptiness problem is **P**-complete, and an implicitly presented automaton can be exponentially big (this would give us an **EXP** upper bound, which is, in fact, tight by Lemma 14 below if no m is given). The **PSPACE** membership is thanks to the upper bound m on the size of the tree, which significantly shrinks the search space.

To prove Lemma 10, we design a decision procedure using the pebbling strategy for trees that we discussed in subsubsection 4.1.3.

Lemma 10. *For a tree automaton implicitly presented in polynomial space and a number m written in binary, one can decide in* **PSPACE** *if the automaton accepts at least one tree with m or fewer nodes.*

Proof. Denote the tree automaton implicitly presented in polynomial space by $\mathfrak{A}_{\bar{q}}$, as above. We describe a procedure that guesses (with checks done on the fly) an accepting computation of $\mathfrak{A}_{\bar{q}}$. Since the number m is given in binary, we cannot afford to write down the entire accepted tree, as it could take up exponential space.

However, suppose that such a tree t exists and has $m' \leq m$ nodes; we assume without loss of generality that $m = m'$. Consider some pebbling strategy for t, as defined in subsubsection 4.1.3. Our procedure will guess moves of this strategy on the fly and simulate them; it will also guess the tree t in lockstep (simultaneously with the strategy). More precisely, we maintain the following invariant. Take any time step and any vertex v and denote by t_v the subtree of t rooted at v. If the pebbling strategy prescribes that v should have a pebble, then our procedure keeps in memory a pair $(\mathfrak{q}, k)$ where $\mathfrak{q} \in \mathfrak{Q}$ is a state of $\mathfrak{A}_{\bar{q}}$ that t_v evaluates to, and k is the total number of nodes in t_v. Note that any such pair $(\mathfrak{q}, k)$ takes up space polynomial in the size of the input: states of $\mathfrak{A}_{\bar{q}}$ have such representations by the assumptions of the lemma, and k never needs to grow higher than m.

We now describe how the moves of the strategy are simulated by our procedure. Suppose the strategy prescribes placing a pebble on a vertex v; by the rules of the pebble game, this means that all immediate predecessors $v_1, \ldots, v_d$ (if any) currently have pebbles on them. By our invariant, we already keep in memory corresponding pairs $(\mathfrak{q}_1, k_1), \ldots, (\mathfrak{q}_d, k_d)$. Our procedure now guesses the node v, i.e., its label $f \in \mathcal{F}$

in t. Then the procedure guesses a new state, $\mathfrak{q} \in \mathfrak{Q}$, verifies in polynomial space that $f(\mathfrak{q}_1, \ldots, \mathfrak{q}_d) \mapsto \mathfrak{q}$ is a transition in $\mathfrak{A}_{\bar{q}}$, and that $k = k_1 + \ldots + k_d + 1$ does not exceed m. If any check is failed, the procedure declares the current nondeterministic branch rejecting; if all the checks are passed, the procedure stores the pair $(\mathfrak{q}, k)$. Naturally, whenever a strategy prescribes removing a pebble from a vertex, the procedure simply erases the corresponding pebble from the memory (in fact, since t is a tree, we can assume that every pair $(\mathfrak{q}, k)$ is removed immediately after its use). At some point, the procedure guesses that the strategy can terminate; this means that the root of the tree t carries a pebble. The procedure picks some pair $(\mathfrak{q}, k)$ from the memory and verifies in polynomial space that the state $\mathfrak{q}$ is indeed final in $\mathfrak{A}_{\bar{q}}$. This signifies acceptance of t_v.

It remains to argue that the procedure only uses polynomial space. The tree t has m nodes, so, by the upper bound on Strahler numbers, the optimal strategy needs $\mathsf{peb}(t) \leq \lfloor \log_2(m+1) \rfloor$ pebbles, which is polynomial in the size of the input. If some guessed step requires more, the strategy cannot be optimal, and the procedure declares the branch rejecting. □

The idea of the proof of Lemma 10 can be distilled in a different form: We can show that the *bounded emptiness* problem (are all trees up to a certain size rejected?) is in **PSPACE** for *succinct tree automata*. These are tree automata where the set of states, $\mathfrak{Q}$, can be exponentially large, but does not need to be written out explicitly, and the set of transitions and the set of final states are represented with Boolean circuits (or, alternatively, with logical formulas over an appropriate theory). The proof follows that of Lemma 10.

We can now prove the upper bound of Theorem 7 by following lemma.

Lemma 11. *The Short Synchronizing Nested Word problem can be decided in* **PSPACE**.

Proof. Combine Lemma 9 and 10 with the fact that the automaton $\mathfrak{A}_{\bar{q}}$ from the former is implicitly presented in polynomial space. Indeed, suppose an NWA $\mathcal{A}$ with states Q and an integer ℓ are given. By Lemma 9, a synchronizing word for $\mathcal{A}$ of length at most ℓ exists if and only if there exists a state $\bar{q} \in Q$ such that the tree automaton $\mathfrak{A}_{\bar{q}}$ accepts some tree over the ranked alphabet $\mathcal{F}$; recall that this is the alphabet defined by Equation (1) in subsubsection 4.1.1. First note that the state $\bar{q}$ can be guessed in polynomial space. Then recall from Claim 2 in subsubsection 4.1.2 that $\mathfrak{A}_{\bar{q}}$ only accepts trees with at most $2\ell-1$ nodes; thus deciding its emptiness reduces to deciding its *bounded emptiness*. Again by Claim 2, $\mathfrak{A}_{\bar{q}}$ is implicitly presented in polynomial space, and thus we can apply Lemma 10 with $m = 2\ell - 1$. □

4.1.5. Binary tree representation of nested words

Here we give more details on the binary tree representation of nested words, which was briefly discussed in Subsubection 4.1.1.

Nested words as trees of unbounded degree.

Given a non-empty well-matched nested word $u = (x, \nu)$ of length ℓ over an alphabet Σ, we define the (essentially standard) tree representation of u as follows. Recall that the matching relation satisfies $\nu \subseteq \{1, \ldots, \ell\}^2$, as u is well-matched, and that $\nu(i, j)$ implies $i < j$. Moreover, whenever $\nu(i, j)$ and $\nu(i', j')$, it cannot be the case that $i < i' \leq j < j'$; this means that the segments $[i, j]$, $[i', j'] \subseteq [1, \ell]$ are either disjoint or contained in one another. Therefore, this property also holds for the binary relation

$$\nu \cup \{(i, i) \mid \text{there is no } j \text{ such that } \nu(i, j) \text{ or } \nu(j, i)\} \cup \{(0, \ell + 1)\}. \tag{2}$$

In other words, the set defined by (2) forms the node set of an ordered rooted tree:

- a node (i', j') is a (non-strict) descendant of (i, j) if and only if $[i', j'] \subseteq [i, j]$;
- if nodes $v_1 = (i_1, j_1)$ and $v_2 = (i_2, j_2)$ are siblings, then either $i_1 \leq j_1 < i_2 \leq j_2$, in which case v_1 is to the left of (comes before) v_2, or $i_2 \leq j_2 < i_1 \leq j_1$, and then v_2 is to the left of (comes before) v_1.

The root of the tree is the pair $(0, \ell + 1)$.

Now take any non-root node $v = (i, j)$ of this tree. If $i < j$, then the ith position in u is a call and the jth position a return, and we associate v with the matched pair of letters $\langle x_i, x_j \rangle$ where $x = x_1 \ldots x_\ell$; we write $\mu(v) = \langle x_i, x_j \rangle$. Otherwise $i = j$ and the ith position in u is internal; in this case we associate v with the letter x_i and write $\mu(v) = x_i$. We perform this for all non-root nodes v; the obtained ordered rooted tree is denoted by $\mathsf{tree}(u)$, the *simple tree representation* of the nested word $u = (x, \nu)$. Let V be the set of all nodes of $\mathsf{tree}(u)$; by convention, in the sequel we treat the values of the partial *association mapping* $\mu \colon V \rightharpoonup \langle \Sigma \times \Sigma \rangle \cup \Sigma$ as part of the tree itself.

We would like to remark that our simple tree representation is very similar to the mapping that transforms so-called "hedge words" into trees [3, subsection 7.1]. In our case, however, positions of x matched by ν do not have to carry identical letters from Σ; moreover, we add a special node as the root of the tree. Note that, in general, nodes of $\mathsf{tree}(u)$ can have unbounded degree (number of children).

Nested words as binary trees.

The final step in our construction is "binarization" of the trees. Based on $\mathsf{tree}(u)$, we construct a new binary tree as follows. For every node v in $\mathsf{tree}(u)$ that has more than two children, say $v_1, \ldots, v_k$ with $k \geq 3$, replace the star formed by v and $v_1, \ldots, v_k$ by any ordered binary tree with root v and leaves $v_1, \ldots, v_k$ (preserving the left-to-right DFS traversal order) where all non-leaf nodes have exactly 2 children (the number of new "auxiliary" nodes will be $k - 2$, not including v). We do not insist on picking any particular shape of the k-leaf tree, because we do not need to rely on uniqueness of representation. Similarly, if the root of $\mathsf{tree}(u)$ has more than one child, we perform an analogous transformation to make the root a unary node. After this we remove the root (recall that it was added artificially in the first place).

The newly obtained tree is binary; we denote it by $\mathsf{bin}(u)$ and call it the *tree representation* of u. (We will not really need the simple tree representation $\mathsf{tree}(u)$

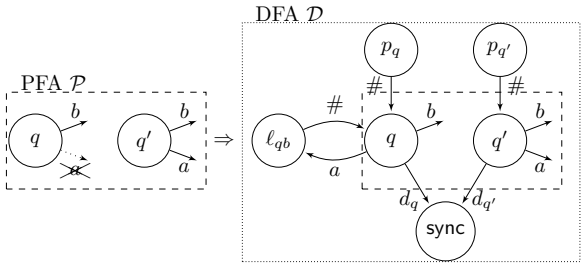

Figure 4: The sketch of the reduction from careful synchronization to small-cost synchronization. All transitions not shown in $\mathcal{D}$ are self-loops.

defined above.) While the construction of $\mathsf{bin}(u)$ is not sophisticated, nodes of $\mathsf{bin}(u)$ come in many different *types*; a summary is given in the table in subsubsection 4.1.1.

4.2. **PSPACE**-*hardness*

The matching lower bound for the Short Synchronizing Nested Word problem is established by a reduction from the *small-cost synchronizing word* problem, which we introduce and prove **PSPACE**-complete below.

For a deterministic finite automaton (DFA) $\mathcal{D} = (\Sigma, Q, \Delta)$ over Σ, consider a function $\mathsf{cost} : \Sigma \to \mathbb{Z}_{>0}$ that assigns (strictly) positive costs to letters $a \in \Sigma$. This function extends to finite words in a natural way: $\mathsf{cost}(w \cdot a) = \mathsf{cost}(w) + \mathsf{cost}(a)$ where $w \in \Sigma^*$, and $\mathsf{cost}(\varepsilon) = 0$. The *small-cost synchronizing word* problem asks, given a DFA equipped with a cost function and a $\mathsf{budget} \in \mathbb{Z}_{>0}$ both written in binary, whether the DFA has a synchronizing word w with $\mathsf{cost}(w) \le \mathsf{budget}$.

Theorem 12. *The small-cost synchronizing word problem is* **PSPACE**-*complete.*

Proof. Since all letters in the given DFA have strictly positive costs, the length of all synchronizing words w with $\mathsf{cost}(w) \le \mathsf{budget}$ is less than budget. This property is essential for a direct guess-and-check algorithm that checks the existence of such words.This algorithm keeps track of the set S_i of current states (starting from $S_0 = Q$) and the accumulated cost C_i (starting with $C_0 = 0$). After guessing a new letter a_i, it updates the current set of states to $S_{i+1} = \Delta(S_i, a_i)$ as well as updates the accumulated cost to $C_{i+1} = C_i + \mathsf{cost}(a_i)$. Since the accumulated costs is strictly increasing with each guess, the algorithm needs at most exponentially many steps. This shows that the membership of the small-cost synchronizing word problem is in **PSPACE**.

The **PSPACE**-hardness is by a reduction from the carefully synchronizing word problem. Given a PFA $\mathcal{P} = (\Sigma, Q, \delta)$, we construct a DFA $\mathcal{D}$ equipped with a function cost and a budget such that $\mathcal{P}$ has a synchronizing word if and only if $\mathcal{D}$ has a synchronizing word w with $\mathsf{cost}(w) \le \mathsf{budget}$.

The sketch of the reduction is depicted in Figure 4. The cost associated to all $a \in \Sigma$ is $\mathsf{cost}(a) = 1$. A new letter $\#$ is introduced with $\mathsf{cost}(\#) = 2^{2|Q|}$; and for all $q \in Q$, a new letter d_q is added with $\mathsf{cost}(d_q) = 2^{|Q|} + 1$. Setting $\mathsf{budget} = 2^{2 \cdot |Q|} + 2^{|Q|+1} + 1$

restricts $\mathcal{D}$ to input # at most once while synchronizing; moreover, if # is read then only one letter among $\{d_q \mid q \in Q\}$ can be chosen.

Let Δ be the transition function of $\mathcal{D}$ where $\Delta(q,a) = \delta(q,a)$ if $\delta(q,a)$ is defined for $q \in Q$ and $a \in \Sigma$. For all $q \in Q$, a new state p_q is added where all transitions are self-loops except #-transition: $\Delta(p_q, \#) = q$. Similarly, for all $q \in Q$ and $a \in \Sigma$, if $\delta(q,a)$ is not defined, we then add a new state ℓ_{qa} where all transitions are self-loops except #-transition: $\Delta(\ell_{qa}, \#) = q$; we also define $\Delta(q,a) = \ell_{qa}$. Hence, to synchronize the states p_q and ℓ_{qa} the letter # must be read at least once. It remains to define for all $q \in Q$: $\Delta(q, \#) = q$, $\Delta(q, d_q) = \mathsf{sync}$, where sync is a new state with all transitions being self-loops, and, for all $q \neq q'$, $\Delta(q, d_{q'}) = q$. The automaton $\mathcal{D}$ can only be synchronized in sync, and since sync is reached only by letters d_q ($q \in Q$), the automaton $\mathcal{D}$ must input d_q at least for one $q \in Q$.

To prove the correctness of the reduction, first observe that if $\mathcal{P}$ has a synchronizing word, then there exist another synchronizing word v with $|v| \leq 2^{|Q|}$ and some state q such that $\delta(Q, v) = \{q\}$. As a result, the word $\# \cdot v \cdot d_q$ is synchronizing for $\mathcal{D}$ and we have that $\mathsf{cost}(\# \cdot v \cdot d_q) = 2^{2 \cdot |Q|} + \mathsf{cost}(v) + 2^{|Q|} + 1$. Since the cost of each letter a is 1, we have that $\mathsf{cost}(v) = |v| \leq 2^{|Q|}$; and thus $\mathsf{cost}(\# \cdot v \cdot d_q) \leq \mathsf{budget}$.

Now, assume that $\mathcal{D}$ has some synchronizing word with cost at most budget; let w be one of the shortest such words. By construction, w must have exactly one occurrence of # and one occurrence of d_q for some state $q \in Q$. Since w is one of the shortest synchronizing words, it follows that $w = w_1 \cdot \# \cdot w_2 \cdot d_q$ where $w_1, w_2 \in \Sigma^*$. We prove that $w_2 = a_1 a_2 \cdots a_n$ is a valid synchronizing word for $\mathcal{P}$ by three observations:

- The set of reached states after inputting # is exactly $Q \cup \{\mathsf{sync}\}$.
- For all $1 \leq i \leq n$ and all states $q \in \Delta(Q, a_1 a_2 \cdots a_i)$, the successor state is never ℓ_{qa_j}; otherwise since w_2 has no occurrence of #, we would get an immediate contradiction with the fact that w is a synchronizing word for $\mathcal{D}$. Thus, the automaton $\mathcal{D}$ only fires "defined" transitions of $\mathcal{P}$ while reading w_2.
- Since $\Delta(q_1, d_q) \neq \Delta(q_2, d_q)$ for all letters d_q and pairs of states $q_1 \neq q_2$, we conclude that $\Delta(Q, w_2) = \delta(Q, w_2)$ is a singleton.

The **PSPACE**-hardness result follows. □

Theorem 12 strengthens **PSPACE**-hardness results for similar models [18, 10]: the key difference is that in our setting the cost function can only depend on input letters and not on individual transitions. Particularly, in [10], the cost function ranges over all integers (in contrast to our setting that it ranges only over strictly positive integers); and the **PSPACE**-hardness result in their model heavily benefits from the freedom of choosing negative weights.

Reduction to Short Synchronizing Nested Word problem

We prove the following by reduction from the small-cost synchronizing word problem.

Lemma 13. *The Short Synchronizing Nested Word problem is* **PSPACE***-hard.*

Proof. The proof is a reduction from the small-cost synchronizing word problem: given a DFA $\mathcal{D} = (\Sigma, Q, \Delta)$ over Σ, $\mathsf{cost}\colon \Sigma \to \mathbb{Z}_{>0}$, and $\mathsf{budget} \in \mathbb{Z}_{>0}$, we find an

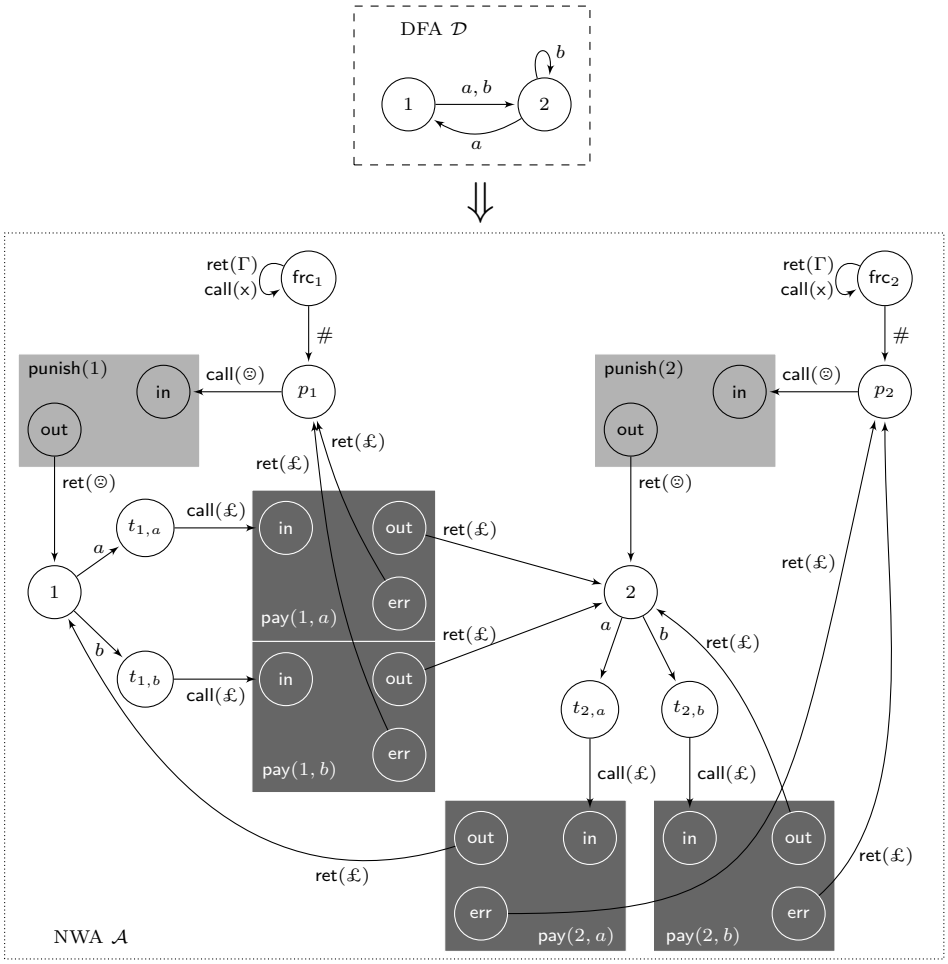

Figure 5: An example of the reduction to the Short Synchronizing Nested Word problem. For $q \in \{1, 2\}$, all #-transitions from q and from all states of gadget $\mathsf{pay}(q, a)$, gadget $\mathsf{pay}(q, b)$, and gadget $\mathsf{punish}(q)$ lead to p_q. All a, b-transitions in all states are self-loops, except in states $1, 2$. The NWA $\mathcal{A}$ has a synchronizing nested word of length at most $4 \cdot \mathsf{budget} + |w_{\mathsf{punish}}| + 1$ if and only if $\mathcal{D}$ has a synchronizing word with cost at most budget.

NWA $\mathcal{A}$ and a length ℓ such that $\mathcal{D}$ has a synchronizing word w with $\mathsf{cost}(w) \leq \mathsf{budget}$ if and only if $\mathcal{A}$ has a synchronizing nested word of length at most ℓ.

The intuition behind the reduction is as follows. We encode the cost of each letter a in $\mathcal{D}$ with the length of a particular well-matched nested word $a \cdot w_a$ in $\mathcal{A}$; as a result, runs in $\mathcal{D}$ are in a sense *simulated* by runs in $\mathcal{A}$. The nested word $a \cdot w_a$ is associated with a special gadget that we insert as a part of $\mathcal{A}$; we denote this gadget $\mathsf{pay}(q, a)$ (there is a separate copy for each $q \in Q$). The intention is that the length of a

Table 1: Summary of the transition function δ of the NWA $\mathcal{A}$ with $\Gamma = \{\mathsf{x}, \mathsf{y}, £, ☹\}$ constructed from the DFA $\mathcal{D} = (\Sigma, Q, \Delta)$ over Σ. The table specifies the destination of all transitions: e.g., when $\mathcal{A}$ is at $q \in Q$ and reads call, it pushes x and stays at q.

State \ Input	Σ	#	call (pushing γ)	ret
$q \in Q$	$t_{q,a}$	p_q	$\gamma = \mathsf{x}$ self-loop	self-loop

For all $q \in Q$ and $a \in \Sigma$:

$t_{q,a}$	self-loop	p_q	$\gamma = £$ in of $\mathsf{pay}(q,a)$	self-loop
p_q	self-loop	p_q	$\gamma = ☹$ in of $\mathsf{punish}(q)$	self-loop
frc_q	self-loop	p_q	$\gamma = \mathsf{x}$ self-loop	self-loop
$s \in \mathsf{pay}(q,a)$	self-loop	p_q	See gadget pay in Figure 6 (left) where: • missing transitions go to state err of the same $\mathsf{pay}(q,a)$ • from out, the transition on $\mathsf{ret}(£)$ goes to $\Delta(q,a)$ • from err, the transition on $\mathsf{ret}(£)$ goes to p_q	
$s \in \mathsf{punish}(q)$	self-loop	p_q	See gadget punish in Figure 6 (right) where: • missing transitions go to state in of the same $\mathsf{punish}(q)$ • from out, the transition on $\mathsf{ret}(☹)$ goes to q	

nested word read by $\mathcal{A}$ corresponds to the cost of some word read by $\mathcal{D}$. Obviously, there will be runs of $\mathcal{A}$ of a form deviating from the form $a_1 \cdot w_{a_1} \cdots a_k \cdot w_{a_k}$; we call such deviations *cheating*. We will ensure that, along runs of interest, cheating is impossible: deviating transitions will lead to another set of gadgets, denoted $\mathsf{punish}(q)$, for all $q \in Q$. When a run of $\mathcal{A}$ is *punished*, it is forced to read a very long nested word w_{punish}, which results in exceeding the length ℓ. On the technical level, this "forcing" means that all shorter continuations make no progress to the synchronization objective.

We now show how to construct the NWA $\mathcal{A}$ following this intuition; an example is shown in Figure 5. The set $Q_{\mathcal{A}}$ of states in $\mathcal{A}$ is

$$Q \cup \bigcup_{q \in Q} \{\mathsf{frc}_q\} \cup \bigcup_{q \in Q, a \in \Sigma} (\mathsf{pay}(q,a) \cup \{t_{q,a}\}) \cup \bigcup_{q \in Q} (\mathsf{punish}(q) \cup \{p_q\})$$

where Q denotes, as above, the set of states of the DFA $\mathcal{D}$, and we abuse the notation by letting $\mathsf{pay}(q,a)$ and $\mathsf{punish}(q)$ refer to the sets of states of the corresponding gadgets. The set of stack symbols of $\mathcal{A}$ is $\Gamma = \{\mathsf{x}, \mathsf{y}, £, ☹\}$; the input letters are $\Sigma \cup \{\#\}$ where $\# \notin \Sigma$ (as in Remark on page 8, all letters at call and return positions are written simply as call and ret). Table 1 describes the transition function of $\mathcal{A}$.

It remains to define the gadgets $\mathsf{pay}(q,a)$ and $\mathsf{punish}(q)$. We base the construction of $\mathsf{pay}(q,a)$ and $\mathsf{punish}(q)$ on the family of NWA $\mathcal{B}_n$ from Example 4; see Figure 6. Each gadget has two designated local states in and out, and the shortest run from in to out is over the nested word that we denote by v_a (where $w_a = \mathsf{call} \cdot v_a \cdot \mathsf{ret}$) in

gadget $\mathsf{pay}(q,a)$ and by v_{punish} (where $w_{\mathsf{punish}} = \mathsf{call} \cdot v_{\mathsf{punish}} \cdot \mathsf{ret}$) in $\mathsf{punish}(q)$. We pick the parameter $k = |v_a|$ in $\mathcal{B}_k$ in such a way that $|a \cdot w_a| = |a \cdot \mathsf{call} \cdot v_a \cdot \mathsf{ret}| = 4 \cdot \mathsf{cost}(a)$; note that $k = 4 \cdot \mathsf{cost}(a) - 3 \geq 1$, since $\mathsf{cost}(a) \geq 1$. (The choice for m in $\mathcal{B}_m$ is discussed below.) Since the NWA $\mathcal{B}_n$ in Example 4 have only partially defined transition functions, we complete them by directing all missing transitions (shown as "errors" in Figure 6) to in in punish and to new local states err in pay. Note that this includes missing transitions on call (they all push x to the stack) and missing transitions on ret (at every control state, there is a popping transition for each $\gamma \in \Gamma$). In contrast, on input # all transitions from $\mathsf{pay}(q,a)$ and $\mathsf{punish}(q)$ go to the state p_q.

By construction, every synchronizing word is forced to have at least one occurrence of #, otherwise the runs starting from states frc_q ($q \in Q$) cannot be synchronized with other runs. Therefore, every synchronizing word must contain the subword w_{punish} at least once. Since any "unfaithful" simulation involves reading another occurrence of w_{punish}, we choose to have $\ell < 2 \cdot |w_{\mathsf{punish}}|$ to avoid the possibility of this second occurrence. Since the gadgets $\mathsf{pay}(q,a)$ are constructed so that $|a \cdot w_a| = 4 \cdot \mathsf{cost}(a)$, every run of $\mathcal{D}$ over a word with cost less than budget can be faithfully simulated with a run in $\mathcal{A}$ with length at most $4 \cdot \mathsf{budget}$. Taking into account one occurrence of # and the subword w_{punish} (which will actually occur at the beginning of the shortest synchronizing word), it is natural to pick $\ell = 1 + |w_{\mathsf{punish}}| + 4 \cdot \mathsf{budget}$. We will require one more constraint: if we have $4 \cdot \mathsf{budget} + 1 < |w_{\mathsf{punish}}|$, then this will imply that $\ell < 2 \cdot |w_{\mathsf{punish}}|$ and will ensure that the property discussed above holds. The choice of $m + 2 = |w_{\mathsf{punish}}| = 4 \cdot \mathsf{budget} + 2$ and $\ell = 8 \cdot \mathsf{budget} + 3$ satisfies these constraints and completes the description of the reduction.

To prove the correctness of the reduction, observe that if $\mathcal{D}$ has a synchronizing word $v = a_1 a_2 \cdots a_n$ with $\mathsf{cost}(v) \leq \mathsf{budget}$, then

$$\# \cdot w_{\mathsf{punish}} \cdot a_1 \cdot w_{a_1} \cdot a_2 \cdot w_{a_2} \cdots a_n \cdot w_{a_n}$$

is a synchronizing word for $\mathcal{A}$ of length

$$|\# \cdot w_{\mathsf{punish}}| + 4 \cdot (\mathsf{cost}(a_1) + \mathsf{cost}(a_2) + \cdots + \mathsf{cost}(a_n)) \leq 1 + |w_{\mathsf{punish}}| + 4 \cdot \mathsf{budget}.$$

For the converse direction, assume that $\mathcal{A}$ has some synchronizing word with length at most ℓ. We prove that $\mathcal{D}$ has a synchronizing word with cost at most budget.

We begin with the following observation. In our construction of $\mathcal{A}$, all states have index $q \in Q$, and the destination of transitions departing from these states may depend on q in either of the following two ways. First, many (and in fact most of the) transitions lead to further states that have index q, and this is in fact the only dependence on q in these transitions: if two states only differ by the index q', q'' with $q' \neq q''$, and the same letter is given as input, then the transitions lead to two states that also differ by the index $\bar{q}' \neq \bar{q}''$ only. (For example, all transitions from states frc_q, $q \in Q$, and from states $q \in Q$ themselves are of this kind.) Second, there are return transitions that depart from the out states in $\mathsf{pay}(q,a)$ gadgets, and for these transitions (exclusively) the destination really depends on q (through Δ). We will use this observation in the argument below.

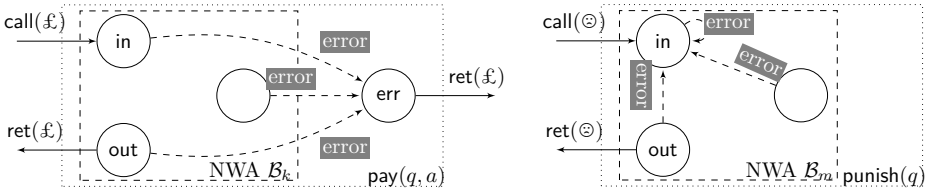

Figure 6: Gadgets $\mathsf{pay}(q, a)$ (on the left) and punish (on the right) where $\mathcal{B}_k, \mathcal{B}_m$ are described in Example 4 with $k = 4 \cdot \mathsf{cost}(a) - 3$ and $m = |w_{\mathsf{punish}}| - 2$

Now let w be one of the shortest synchronizing words of $\mathcal{A}$. Consider the $|Q|$ runs of $\mathcal{A}$ starting in configurations $\{(\mathsf{frc}_q, \varepsilon) \mid q \in Q\}$; we will follow these runs, ignoring all other computations. Using the observation above, one can see by induction on the length of the input word that the configurations reached by these runs on the same input can only differ by the index q of the control state. Notice that since the initial configurations have identical stack content (ε), and the stack alphabet has no symbol that depends on $q \in Q$, we will see that the stack content stays the same across the $|Q|$ runs at all times (even though it may evolve); so we will always find these runs in configurations that only differ by the index $q \in Q$.

Since the runs are synchronized by the word w, this word necessarily contains the letter # and, at some future point, the word w_{punish}. By the arguments above, when these runs reach states from Q for the first time, the stack content will be the same for all of them (it may or may not be ε).

From this point onward, the following properties will hold:

(a) the remainder of the input cannot contain the word w_{punish} (because ℓ is less than $2\,|w_{\mathsf{punish}}|$, and the prefix of w already contains an occurrence of w_{punish}); and

(b) the $|Q|$ runs we are following will never enter states $\{p_q \mid q \in S\}$, for any $|S| \geq 2$, because, in order to synchronize any set of configurations with these control states, a word of length at least w_{punish} will be required.

Using the observation above, we can conclude from property (b) that no run will (re-)enter p_q for any $q \in Q$. (Indeed, if such a run existed, then all the $|Q|$ runs would be in states p_q, $q \in S$, for $|S| \leq 1$, that is, in the same control state. This means that $\mathcal{A}$ has a synchronizing word shorter than w—a contradiction.)

Let $v = b_1 b_2 \cdots b_m$ be the suffix of w read from the point in time that our $|Q|$ runs reach Q for the first time. At this point the runs are in $|Q|$ configurations (q, s) where $q \in Q$ and where $s \in \Gamma^*$ does not depend on q. Denote $S_0 = Q \times \{s\}$ and $S_i = \delta(S_{i-1}, b_i)$ for all $1 \leq i \leq m$. It now follows from the observation above that, for all $1 \leq i \leq m$, $S_i \cap (Q \times \Gamma^*) \neq \emptyset$ if and only if $S_i \subseteq Q \times \{s_i\}$ for some $s_i \in \Gamma^*$ (by induction on i).

Consider the nested word $c_1 c_2 \cdots c_j$ obtained from $v = b_1 \cdots b_m$ by removing all letters b_i except those for which $S_{i-1} \subseteq Q \times \{s_{i-1}\}$. Since the runs we are following never re-enter states p_q for any $q \in Q$, such runs must enter pay-gadgets when departing from configurations in S_{i-1}, reading a letter $a \in \Sigma$ and then call from the input; we will have $c_{i'} = a$. Since a is now fixed, all the gadgets $\mathsf{pay}(q, a)$ have the

same construction: the only way out that involves no occurrence of # in the input is going through the states out of the gadgets (popping £ from the stack); after this, the runs move to $Q \times \Gamma^*$ again. Therefore, for all $1 \le i \le j$ we have

$$Q_i = \Delta(Q, c_1 \cdots c_i),$$

where $Q_i \times \{s\}$ is the ith element in the sequence $S_1, S_2, \ldots$ that is included in $Q \times \Gamma^*$. So the word $c_1 c_2 \cdots c_j$ is a valid input word for the DFA $\mathcal{D}$.

It remains to prove that this word is synchronizing for $\mathcal{D}$ with $\mathsf{cost}(c_1 \cdots c_j) \le \mathsf{budget}$. This word must be a synchronizing word: otherwise there are two distinct states $q, q' \in Q$ that are reached by the prefix of w up to and including c_j. All runs starting from these two states q and q' on the suffix of w after c_j—since they do not visit $Q \times \Gamma^*$ again—must stay in separate pay gadgets and cannot be synchronized, a contradiction.

To complete the proof, we show that $\mathsf{cost}(c_1 c_2 \cdots c_j) \le \mathsf{budget}$. Recall that three factors #, w_{punish}, and v of w are non-overlapping, so $|v| \le 4 \cdot \mathsf{budget}$. By above arguments, word $v = b_1 \cdots b_m$ is a correct simulation of $c_1 \cdots c_j$ in $\mathcal{D}$, thus each c_i is followed by w_{c_i} where $|c_i \cdot w_{c_i}| = 4 \cdot \mathsf{cost}(c_i)$. We thus have $\mathsf{cost}(c_1 \cdots c_j) \le \mathsf{budget}$. □

5. Synchronizing nested words for subsets of states

In this section, we prove that the existence of a well-matched nested word that synchronizes a given set of states to another set of states in an NWA is **EXP**-complete. The corresponding decision problems for DFA are **PSPACE**-complete [32, 33], where hardness is by a reduction from the DFA intersection nonemptiness problem (see [38] for a refined complexity analysis).

In the NWA case, the proofs are an easy adaptation of these arguments and are based on the following observation, which can be proved by a translation from tree automata [9] or by a direct extension of Kozen's proof [21]. (For completeness, we provide the proof.) When we talk about words *accepted* by $\mathcal{A}$, we implicitly assume that $\mathcal{A}$ comes equipped with a subset $Q^{\mathsf{f}} \subseteq Q$ of *final* (*accepting*) states; accepted are words u for which there exists a path $(q_0, \varepsilon) \xrightarrow{u} (\bar{q}, s)$ with $\bar{q} \in Q^{\mathsf{f}}$.

Lemma 14 (cf. [9, Theorem 1.7.5]). *The following Intersection Nonemptiness problem for NWA is* **EXP***-complete: Given NWA $\mathcal{A}_1, \ldots, \mathcal{A}_m$, decide if there exists a well-matched word accepted by all $\mathcal{A}_i$.*

Proof. Membership in **EXP** is immediate: using the standard product construction for NWA, we obtain an exponential-size NWA that accepts only those (well-matched) words that are accepted by all of $\mathcal{A}_1, \ldots, \mathcal{A}_m$. We then run a polynomial-time emptiness check for this NWA.

The proof of **EXP**-hardness relies on the equality **APSPACE** = **EXP**, where **APSPACE** denotes the class of problems solvable by alternating Turing machines that use polynomial space. We first show how to argue **PSPACE**-hardness, recalling the proof by Kozen [21], and then strengthen this to **APSPACE**-hardness.

First consider some *deterministic* polynomial-space Turing machine $\mathcal{M}$ and an input tape x for $\mathcal{M}$. We assume that the input tape of $\mathcal{M}$ is also its working tape. Denote $|x| = n$; this length will not change during any computation of $\mathcal{M}$. We show how to construct polynomially many DFA whose language intersection is non-empty if and only if $\mathcal{M}$ accepts x. These DFA have input alphabet $\delta \times \{1, \dots, n\}$, where δ is the set of transitions of $\mathcal{M}$. Words accepted by *all* these DFA will correspond to accepting computations of $\mathcal{M}$. The component $i \in \{1, \dots, n\}$ of the input letters corresponds to the position of the head of $\mathcal{M}$ on the tape.

For each $i \in \{1, \dots, n\}$, define a DFA $\mathcal{A}_i$ that remembers the letter currently written in the ith cell of the input tape. Initially, it is simply the ith letter of x, but it can be rewritten later. When $\mathcal{A}_i$ reads a pair (t, i) where $t \in \delta$, it checks whether the transition t is compatible with the letter currently written in the ith cell of the tape, and then it may update this letter if t requires this. When $\mathcal{A}_i$ reads a pair $(t, j) \in \times\{1, \dots, n\}$ with $j \neq i$, it does not change its state.

In addition to $\mathcal{A}_1, \dots, \mathcal{A}_n$, we also define a DFA $\mathcal{A}_0$ that does the following book-keeping: it remembers the current control state of $\mathcal{M}$, checks its compatibility with transitions, and updates it accordingly. The same automaton can check whether the positions $i \in \{1, \dots, n\}$ of the head are compatible with the movements prescribed by the transitions, and also whether $\mathcal{M}$ reaches an accepting state at the end. As a result, these $n + 1$ DFA have non-empty language intersection if and only if $\mathcal{M}$ accepts x.

To extend this to **APSPACE**-hardness, we consider an *alternating* Turing machine $\mathcal{M}$. An accepting computation of $\mathcal{M}$ is no longer just a sequence of transitions: it is a (binary) tree, where the edges are transitions, and the branching can be non-deterministic and universal. This tree can be naturally represented as a nested word, and instead of DFA, we will construct NWA. We are only interested in nested words with the property that, in every pair of matched positions in the word, *both* contain the same transition $t \in \delta$ and $i \in \{1, \dots, n\}$. (There is a small NWA that checks this condition.) Moreover, all positions in such a nested word must be calls or returns (i.e., no internal positions).

Each NWA $\mathcal{A}_i$ will, as previously, keep track of the letter written in the ith cell of the input tape. However, the input to $\mathcal{A}_i$ is now a nested word (a purported computation tree of $\mathcal{M}$), so the behaviour of $\mathcal{A}_i$ is a bit more involved. The changes compared to the DFA case are as follows.

- When reading a transition $(t, i) \in \delta \times \{1, \dots, n\}$ in a call position, the NWA $\mathcal{A}_i$ will (in addition to the book-keeping described above) push the current content of the ith cell (*before* a possible update) to the stack.
- When reading a transition $(t, i) \in \delta \times \{1, \dots, n\}$ in a return position, the NWA $\mathcal{A}_i$ will pop the stack and undo the update, "rewriting" the content of the ith cell with the letter it previously pushed.

This way, when NWA read call letters, they move from root to leaves in the computation tree of $\mathcal{M}$ (similarly to the DFA case), and when they read return positions, they move from leaves to root ('back in time'), undoing the changes.

The NWA $\mathcal{A}_0$ will now also keep track of whether subtrees of the computation tree are accepting or not. Suppose $\mathcal{A}_0$ remembers that the current control state

of $\mathcal{M}$ is q. If this state has nondeterministic or universal branching, then $\mathcal{A}_0$ also needs to remember which of the (at most two) branches it has explored, and whether the corresponding subtrees are accepting. After it has explored all the subtrees of the current node, $\mathcal{A}_0$ will know whether the subtree rooted at the current node is accepting or rejecting (according to the logical AND/OR of the values from the smaller subtrees). It will then expect to read a return letter and propagate this information further up the computation tree (in the direction from leaves to root). Naturally, the entire tree needs to be accepting in order for $\mathcal{A}_0$ to accept the given nested word.

At the end, we have $n+1$ NWA, which accept a common well-matched nested word if and only if $\mathcal{M}$ accepts x. This completes the proof.

□

We prove the following theorem.

Theorem 15. *The following decision problems, with an NWA $\mathcal{A}$ part of the input, are* **EXP***-complete:*

- Synchronizing a Subset: *Given a subset $I \subseteq Q$, decide if there exists a well-matched nested word u such that $(I,\varepsilon) \xrightarrow{u} (\bar{q},\varepsilon)$ for some state $\bar{q} \in Q$.*
- Synchronizing to a Subset: *Given a subset $F \subseteq Q$, decide if there exists a well-matched nested word u such that $(Q,\varepsilon) \xrightarrow{u} (F',\varepsilon)$ for some subset $F' \subseteq F$.*
- Synchronizing a Subset to Another Subset: *Given subsets $I \subseteq Q$ and $F \subseteq Q$, decide if there exists a well-matched nested word u such that $(I,\varepsilon) \xrightarrow{u} (F',\varepsilon)$ for some subset $F' \subseteq F$.*

Proof. Let $\mathcal{A} = (\Sigma, Q, \Gamma, \delta, q_0, Q^{\mathsf{f}}, \gamma_0)$ be an NWA. Notice that the first two problems are special cases of the last one. It therefore suffices to prove lower bounds for the former and a matching upper bound of the latter.

EXP*-hardness*

We present reductions from the Intersection Nonemptiness problem for NWA: given NWA $\mathcal{A}_1, \ldots, \mathcal{A}_m$, decide if there exists a well-matched nested word accepted by all $\mathcal{A}_i$.

Suppose NWA $\mathcal{A}_1, \ldots, \mathcal{A}_m$ are given, and let $\mathcal{W}$ be the NWA (over the same alphabet as $\mathcal{A}_1, \ldots, \mathcal{A}_m$) defined in Example 1, and with p_{yes} as the only final state. This automaton accepts all well-matched nested words, and all runs that start and end in p_{yes} are over well-matched nested words too. Everywhere below, when establishing reductions from the Intersection Nonemptiness problem for NWA, we assume without loss of generality that $\mathcal{A}_m = \mathcal{W}$.

Synchronizing a Subset. Let $\mathcal{A}_1, \ldots, \mathcal{A}_m$ be m NWA over the same alphabet Σ. We construct an NWA $\bar{\mathcal{A}}$ and a set I such that there exists a well-matched nested word accepted by all $\mathcal{A}_i$ if and only if exists some well-matched nested word u and some state $\bar{q}$ in $\bar{\mathcal{A}}$ where $(I,\varepsilon) \xrightarrow{u} (\bar{q},\varepsilon)$.

The construction is as follows (see Figure 7). Let $q_1, \cdots, q_m$ be the initial states and $Q_1^{\mathsf{f}}, \cdots, Q_m^{\mathsf{f}}$ the accepting sets for m NWA. Let $\bar{\mathcal{A}}$ be the NWA that has one copy of

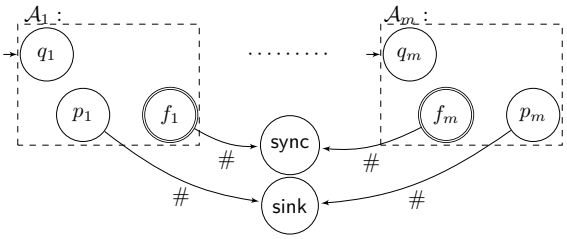

Figure 7: The sketch of the reduction from the Intersection Nonemptiness problem to the Synchronizing a Subset problem in NWA.

each NWA $\mathcal{A}_i$ $(1 \leq i \leq m)$ and new absorbing states sync and sink. For a new internal letter #, let #-transitions in all accepting states $q \in Q_1^{\mathsf{f}} \cup \cdots \cup Q_m^{\mathsf{f}}$ of all $\mathcal{A}_i$ go to sync, whereas all #-transitions in all non-accepting states $q \notin Q_1^{\mathsf{f}} \cup \cdots \cup Q_m^{\mathsf{f}}$ go to sink. From the subset $I = \{q_1, \cdots, q_m, \mathsf{sync}\}$, a synchronizing word cannot escape sync, because sync is an absorbing state. Let u be one of the shortest synchronizing words from I. We now show that u always ends with #.

First, observe that the only way for the runs starting in the initial states $q_1, \cdots, q_m$ to be synchronized in sync is to have the symbol # appear at the right time in the input, when these runs are simultaneously in some accepting states in all copies. Second, due to the fact that $\mathcal{A}_m = \mathcal{W}$, we know that all runs of $\mathcal{A}_m$ from q_m to Q_m^{f} are over well-matched nested words. Thus, there exists a well-matched nested word v such that $u = v \cdot \#$. One can verify that v is an accepting word for all $\mathcal{A}_i$. The **EXP**-hardness follows.

Synchronizing to a Subset. Let $\mathcal{A}_1, \ldots, \mathcal{A}_m$ be m NWA over the same alphabet Σ. We construct $\bar{\mathcal{A}}$ and set F such that there exists a well-matched nested word accepted by all $\mathcal{A}_i$ if and only if there exists some well-matched nested word u and some state $\bar{q}$ in $\bar{\mathcal{A}}$ where $(Q, \varepsilon) \xrightarrow{u} (F', \varepsilon)$ for some subset $F' \subseteq F$.

The construction is as follows (see Figure 8). Let $q_1, \cdots, q_m$ be the initial states and $Q_1^{\mathsf{f}}, \cdots, Q_m^{\mathsf{f}}$ be the accepting sets for m NWA. Let $\bar{\mathcal{A}}$ be the NWA that has one copy of each NWA $\mathcal{A}_i$ $(1 \leq i \leq m)$ and two new states frc and sync. For a new internal letter #, let #-transitions in all states of $\mathcal{A}_i$ go to q_i. The state sync is absorbing, whereas the state frc has self-loops for all letters except #. The #-transition in frc leads to sync. Let $F = Q_1^{\mathsf{f}} \cup \cdots \cup Q_m^{\mathsf{f}} \cup \{\mathsf{sync}\}$. No word can let the NWA $\bar{\mathcal{A}}$ escape sync, because sync is an absorbing state. Let u be the shortest synchronizing word to the set F, thus u must have at least one occurrence of #; otherwise state frc cannot be synchronized to F. Let $u = w \cdot \# \cdot v$ be such that v has no #. As soon as the last # of u is read, each NWA $\mathcal{A}_i$ is reset to its initial state q_i. To be synchronized in F, each NWA $\mathcal{A}_i$ from q_i must reach the final state Q_i^{f}, by reading the word v from the input. Recall that $\mathcal{A}_m = \mathcal{W}$. If w is not well-matched, then v must complete some pending calls from w—but that moves $\mathcal{W}$ to p_{sink}, a contradiction with the fact that u synchronizes $\bar{\mathcal{A}}$ to F. Thus, the word v are both well-matched (notice that it has no pending calls, because otherwise the run of $\mathcal{A}_m$ on v would terminate in p_{no}). It

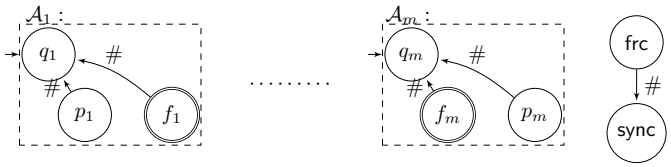

Figure 8: The sketch of the reduction from the Intersection Nonemptiness problem to the Synchronizing to a Subset problem in NWA. All transitions not shown in sync and frc are self-loops.

remains to observe that each $\mathcal{A}_i$ has an accepting run over the well-matched nested word v to its final states. The **EXP**-hardness follows.

Membership in **EXP**

Synchronizing a Subset. We reduce this problem to the synchronizing a subset to another subset problem, which we discuss below. Let $\mathcal{A} = (\Sigma, Q, \Gamma, \delta, q_0, Q^{\mathsf{f}}, \gamma_0)$ be an NWA with n states, and let $I \subseteq Q$. To decide whether there exists a well-matched nested word u and some state $\bar{q} \in Q$ such that $(I, \varepsilon) \xrightarrow{u} (\bar{q}, \varepsilon)$, we make $|Q|$ queries, for each $q \in Q$, to the synchronizing the subset I to the singleton $\{q\}$.

Synchronizing a Subset to Another Subset. Given $\mathcal{A}$ with set Q of states, $I \subseteq Q$ and $F \subseteq Q$, we reduce this problem to the emptiness problem of a product NWA, possibly exponential in the size of $\mathcal{A}$. The reduction is simple: for each state $q \in I$, we introduce an NWA $\mathcal{A}_q$ which is a copy of $\mathcal{A}$ where the initial state is q and the final states are F. Consider the product automaton $\bar{\mathcal{A}}$ of all NWA $\mathcal{A}_q$ with $q \in I$; an accepting word in $\bar{\mathcal{A}}$ synchronizes the set I to F in the original NWA $\mathcal{A}$. The size of this product is $|Q|^{|I|}$ which is exponential, and the emptiness problem for the product automata can be solved in time polynomial in $|Q|^{|I|}$. The **EXP** upper bound follows. □

Acknowledgements

The authors are grateful to Michael Wehar for comments.

References

[1] R. Alur, Nested words and visibly pushdown languages. Web page. Accessed on 28 February 2019.
`https://www.cis.upenn.edu/~alur/nw.html`

[2] R. Alur, V. Kumar, P. Madhusudan, M. Viswanathan, Congruences for visibly pushdown languages. In: L. Caires, G. F. Italiano, L. Monteiro, C. Palamidessi, M. Yung (eds.), *Automata, Languages and Programming, 32nd International Colloquium, ICALP 2005, Lisbon, Portugal, July 11-15, 2005, Proceedings*. Lecture Notes in Computer Science 3580, Springer, 2005, 1102–1114.

[3] R. Alur, P. Madhusudan, Adding nesting structure to words. *Journal of the ACM* **56** (2009) 3.

[4] P. Babari, K. Quaas, M. Shirmohammadi, Synchronizing data words for register automata. In: P. Faliszewski, A. Muscholl, R. Niedermeier (eds.), *MFCS*. LIPIcs 58, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016, 15:1–15:15.

[5] V. Bárány, C. Löding, O. Serre, Regularity problems for visibly pushdown languages. In: B. Durand, W. Thomas (eds.), *STACS*. Lecture Notes in Computer Science 3884, Springer, 2006, 420–431.

[6] D. Caucal, Synchronization of pushdown automata. In: O. H. Ibarra, Z. Dang (eds.), *Developments in Language Theory, 10th International Conference, DLT 2006, Santa Barbara, CA, USA, June 26-29, 2006, Proceedings*. Lecture Notes in Computer Science 4036, Springer, 2006, 120–132.

[7] D. Chistikov, P. Martyugin, M. Shirmohammadi, Synchronizing automata over nested words. In: B. Jacobs, C. Löding (eds.), *Foundations of Software Science and Computation Structures - 19th International Conference, FOSSACS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. Lecture Notes in Computer Science 9634, Springer, 2016, 252–268.

[8] M. Chytil, B. Monien, Caterpillars and context-free languages. In: C. Choffrut, T. Lengauer (eds.), *STACS 90, 7th Annual Symposium on Theoretical Aspects of Computer Science, Rouen, France, February 22-24, 1990, Proceedings*. Lecture Notes in Computer Science 415, Springer, 1990, 70–81.

[9] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, M. Tommasi, Tree Automata Techniques and Applications. Available at: `http://www.grappa.univ-lille3.fr/tata`, 2007. Release October, 12th 2007.

[10] L. Doyen, L. Juhl, K. G. Larsen, N. Markey, M. Shirmohammadi, Synchronizing words for weighted and timed automata. In: V. Raman, S. P. Suresh (eds.), *34th International Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2014, December 15-17, 2014, New Delhi, India*. LIPIcs 29, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2014, 121–132.

[11] L. Doyen, T. Massart, M. Shirmohammadi, The complexity of synchronizing Markov decision processes. *J. Comput. Syst. Sci.* **100** (2019), 96–129.

[12] E. Driscoll, A. V. Thakur, T. W. Reps, OpenNWA: A nested-word automaton library. In: P. Madhusudan, S. A. Seshia (eds.), *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*. Lecture Notes in Computer Science 7358, Springer, 2012, 665–671.

[13] D. Eppstein, Reset sequences for monotonic automata. *SIAM J. Comput.* **19** (1990) 3, 500–510.

[14] A. P. Ershov, On programming of arithmetic operations. *Commun. ACM* **1** (1958) 8, 3–9.

[15] J. Esparza, P. Ganty, S. Kiefer, M. Luttenberger, Parikh's theorem: A simple and direct automaton construction. *Inf. Process. Lett.* **111** (2011) 12, 614–619.

[16] J. Esparza, P. Ganty, R. Majumdar, Parameterized verification of asynchronous shared-memory systems. *J. ACM* **63** (2016) 1, 10:1–10:48.

[17] J. Esparza, M. Luttenberger, M. Schlund, A brief history of Strahler numbers. In: A. Dediu, C. Martín-Vide, J. L. Sierra-Rodríguez, B. Truthe (eds.), *Language and Automata Theory and Applications - 8th International Conference, LATA 2014, Madrid, Spain, March 10-14, 2014. Proceedings*. Lecture Notes in Computer Science 8370, Springer, 2014, 1–13.

[18] F. M. Fominykh, P. V. Martyugin, M. V. Volkov, P(l)aying for synchronization. *Int. J. Found. Comput. Sci.* **24** (2013) 6, 765–780.

[19] M. Heizmann, C. Schilling, D. Tischner, Minimization of visibly pushdown automata using partial Max-SAT. In: A. Legay, T. Margaria (eds.), *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*. Lecture Notes in Computer Science 10205, 2017, 461–478.

[20] J. E. Hopcroft, W. J. Paul, L. G. Valiant, On time versus space. *J. ACM* **24** (1977) 2, 332–337.

[21] D. Kozen, Lower bounds for natural proof systems. In: *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*. IEEE Computer Society, 1977, 254–266.

[22] J. Kretínský, K. G. Larsen, S. Laursen, J. Srba, Polynomial time decidability of weighted synchronization under partial observability. In: L. Aceto, D. de Frutos-Escrig (eds.), *26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1.4, 2015*. LIPIcs 42, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015, 142–154.

[23] V. Kumar, P. Madhusudan, M. Viswanathan, Visibly pushdown automata for streaming XML. In: C. L. Williamson, M. E. Zurko, P. F. Patel-Schneider, P. J. Shenoy (eds.), *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007*. ACM, 2007, 1053–1062.

[24] K. G. Larsen, S. Laursen, J. Srba, Synchronizing strategies under partial observability. In: P. Baldan, D. Gorla (eds.), *CONCUR 2014 - Concurrency Theory - 25th International Conference, CONCUR 2014, Rome, Italy, September 2-5, 2014. Proceedings*. Lecture Notes in Computer Science 8704, Springer, 2014, 188–202.

[25] T. Lengauer, R. E. Tarjan, The space complexity of pebble games on trees. *Inf. Process. Lett.* **10** (1980) 4/5, 184–188.

[26] C. Löding, P. Madhusudan, O. Serre, Visibly pushdown games. In: K. Lodaya, M. Mahajan (eds.), *FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science, 24th International Conference, Chennai, India, December 16-18, 2004, Proceedings*. Lecture Notes in Computer Science 3328, Springer, 2004, 408–420.

[27] P. Martyugin, Computational complexity of certain problems related to carefully synchronizing words for partial automata and directing words for nondeterministic automata. *Theory Comput. Syst.* **54** (2014) 2, 293–304.

[28] K. Mehlhorn, Pebbling moutain ranges and its application of dcfl-recognition. In: J. W. de Bakker, J. van Leeuwen (eds.), *Automata, Languages and Programming, 7th Colloquium, Noordweijkerhout, The Netherlands, July 14-18, 1980, Proceedings*. Lecture Notes in Computer Science 85, Springer, 1980, 422–435.

[29] J. Olschewski, M. Ummels, The complexity of finding reset words in finite automata. In: P. Hlinený, A. Kucera (eds.), *Mathematical Foundations of Computer Science 2010, 35th International Symposium, MFCS 2010, Brno, Czech Republic, August 23-27, 2010. Proceedings*. Lecture Notes in Computer Science 6281, Springer, 2010, 568–579.

[30] M. S. Paterson, C. E. Hewitt, Comparative schematology. In: *Record of the Project MAC conference on concurrent systems and parallel computation*. ACM, 1970, 119–127. MIT AI Memo AIM-201, http://hdl.handle.net/1721.1/5851.

[31] J.-E. Pin, On two combinatorial problems arising from automata theory. *North-Holland Mathematics Studies* **75** (1983), 535–548.

[32] I. K. Rystsov, Polynomial complete problems in automata theory. *Inf. Process. Lett.* **16** (1983) 3, 147–151.

[33] S. Sandberg, Homing and synchronizing sequences. In: M. Broy, B. Jonsson, J. Katoen, M. Leucker, A. Pretschner (eds.), *Model-Based Testing of Reactive Systems, Advanced Lectures [The volume is the outcome of a research seminar that was held in Schloss Dagstuhl in January 2004]*. Lecture Notes in Computer Science 3472, Springer, 2004, 5–33.

[34] J. E. Savage, *Models of computation - exploring the power of computing*. Addison-Wesley, 1998.

[35] M. Szykula, Improving the upper bound on the length of the shortest reset word. In: R. Niedermeier, B. Vallée (eds.), *35th Symposium on Theoretical Aspects of Computer Science, STACS 2018, February 28 to March 3, 2018, Caen, France*. LIPIcs 96, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018, 56:1–56:13.

[36] J. Černý, A. Pirická, B. Rosenauerová, On directable automata. *Kybernetika* **07** (1971) 4, (289)–298.

[37] M. V. Volkov, Synchronizing automata and the cerny conjecture. In: C. Martín-Vide, F. Otto, H. Fernau (eds.), *Language and Automata Theory and Applications, Second International Conference, LATA 2008, Tarragona, Spain, March 13-19, 2008. Revised Papers*. Lecture Notes in Computer Science 5196, Springer, 2008, 11–27.

[38] M. Wehar, Hardness results for intersection non-emptiness. In: J. Esparza, P. Fraigniaud, T. Husfeldt, E. Koutsoupias (eds.), *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part II*. Lecture Notes in Computer Science 8573, Springer, 2014, 354–362.

[39] C. Wu, I. Demongodin, A. Giua, Computation of synchronizing sequences for a class of 1-place-unbounded synchronized petri nets. In: *5th International Conference on Control, Decision and Information Technologies, CoDIT 2018, Thessaloniki, Greece, April 10-13, 2018*. IEEE, 2018, 51–57.