Synchronizing Automata over Nested Words

Dmitry Chistikov[1(✉)], Pavel Martyugin[2], and Mahsa Shirmohammadi[3]

[1] Max Planck Institute for Software Systems (MPI-SWS),
Kaiserslautern and Saarbrücken, Germany
dch@mpi-sws.org
[2] Institute of Mathematics and Computer Science,
Ural Federal University, Ekaterinburg, Russia
martuginp@gmail.com
[3] University of Oxford, Oxford, UK
mahsa.shirmohammadi@cs.ox.ac.uk

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.

1 Introduction

The concept of a synchronizing word for finite-state machines has been studied in automata theory for more than half a century [22,25]. Given a deterministic finite 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"

B. Jacobs and C. Löding (Eds.): FOSSACS 2016, LNCS 9634, pp. 252–268, 2016.
DOI: 10.1007/978-3-662-49630-5_15

(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 [17] 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 [1].

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., [6] 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 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 [5]). 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 1; cf. location-synchronization in [5]). Consider, for instance, an XML processor 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 2), 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 $(n^3-n)/6$, due to Pin [20]; Černý proved in the 1960s [24] 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 bound of Pin (see [25] for a survey). In the case of nested words, the exponential 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 3), 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 4); for DFA, the complexity is polynomial space [21,22]. 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 [14] and which we observe to be complete for exponential time over nested words (Lemma 5).

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 [7], and for NWA it becomes **PSPACE**-complete (Theorem 6). We believe that both upper and lower bound techniques that we use to prove this result are of interest.

Specifically, for the upper bound (Sect. 4) 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., [23, Chapter 10]). We simulate optimal strategies for trees in such games [15], whose efficiency is determined by *Strahler numbers* [11]. Previous use of this technique in formal language theory and verification is primarily associated with derivations of context-free grammars, see, e.g., [9,10] and [11] for a survey. In this body of work, closest to ours are apparently arguments due to Chytil and Monien [3]. We believe that 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 (Sect. 5) we construct a two-step reduction from the problem of existence of *carefully* synchronizing words for partial DFA, whose hardness is known [16]. 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 [5,12], 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 alone is a ground for hardness.

We mention without proof that some of our techniques naturally extend to (going via) tree automata over ranked trees.

2 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 $\mu(i) \cap \{1, \ldots, k\} \neq \emptyset$, 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 $\mathcal{A} = (Q, \Gamma, \delta, q_0, \gamma_0)$, where:

- 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 tuple $(q, s) \in Q \times \Gamma^*$. 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, 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 $s = s' = \varepsilon$.

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 each $a \in \Sigma$. 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 automata*. A *path* (*run*, *computation*) of an automaton $\mathcal{A}$ 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 of configurations (p_i, s_i), $i = 0, \dots, k$, with $(p_{i-1}, s_{i-1}) \xrightarrow{a_i} (p_i, s_i)$ for all i. We will sometimes talk about words *accepted* by $\mathcal{A}$, in which case we implicitly assume that $\mathcal{A}$ comes equipped with a subset $Q^{\mathsf{f}} \subseteq Q$; accepted are words u for which there exists a path $(q_0, \varepsilon) \xrightarrow{u} (\bar{q}, s)$ with $\bar{q} \in Q^{\mathsf{f}}$.

3 Synchronizing Words for NWA

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 1. *A well-matched nested word u is* synchronizing *for an NWA $\mathcal{A} = (Q, \Gamma, \delta, q_0, \gamma_0)$ 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. Definition 1 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 [2]: in NWA, for example, input symbols and stack actions are synchronized (in this sense).

Example 2. 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.

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 Fig. 1. Here the shortest run from in to out is over $\mathsf{call}(\mathsf{x}) \cdot w \cdot \mathsf{ret}(\mathsf{x}) \cdot \mathsf{call}(\mathsf{y}) \cdot w \cdot \mathsf{ret}(\mathsf{y})$ and has length $2n+4$; splitting the state q_z into two states, with internal transitions 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 NWAs with $1, 2, 3, 4$ states). 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 2 shows the sketch of the construction: there are two new letters $\#$ and $\pounds$ and a new absorbing state sync. From all states q of $\mathcal{B}_{n-2}$, the letter $\#$ resets the NWA to in whereas $\pounds$-transitions are all self-loops except in the state out where $\mathsf{out} \xrightarrow{\pounds} \mathsf{sync}$. 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 [16]; it is thus of utmost importance that our NWA are complete). Observe that the shortest synchronizing word has length exactly n; it is $\# \cdot w \cdot \pounds$ where w is the shortest word that takes $\mathcal{B}_{n-2}$ from in to out.

Remark. Our Example 2 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. All call positions in the word are simply call, and all return positions are ret; in figures, the letter in parentheses is the pushed or popped stack symbol.

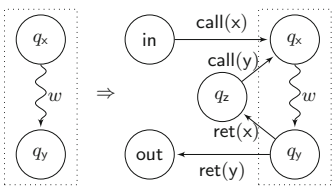

Fig. 1. Doubling transformation

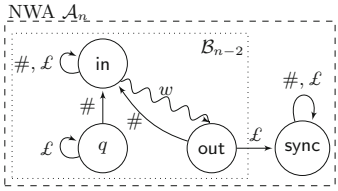

Fig. 2. NWA $\mathcal{A}_n$ based on $\mathcal{B}_{n-2}$

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

Theorem 3. *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 $\mathcal{A}$.*

This theorem extends a characterization of synchronizing automata 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., $(\{p, q\}, \varepsilon) \xrightarrow{u} (\bar{q}, \varepsilon)$ for some $\bar{q}$.

Theorem 4. *The following decision problems, with an NWA $\mathcal{A}$ part of the input, are* **EXP***-complete:*

(1) 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 $\bar{q} \in Q$.
(2) 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 $F' \subseteq F$.
(3) 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 $F' \subseteq F$.

The corresponding decision problems for DFA are **PSPACE**-complete [21,22], where hardness is by a reduction from the DFA intersection nonemptiness problem (see [26] for a more 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 or by a direct extension of Kozen's proof [14]:

Lemma 5. *The following problem is* **EXP***-complete: Given NWA $\mathcal{A}_1, \ldots, \mathcal{A}_m$, decide if there exists a well-matched word accepted by all $\mathcal{A}_i$.*

The following theorem is our main result.

Theorem 6. *The following problem* Short Synchronizing Nested Word *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.*

The corresponding decision problem for DFA is **NP**-complete [7]. (Note that deciding if the shortest synchronizing word has length exactly ℓ, a related but different problem, is **DP**-complete [18].) Since any DFA with a synchronizing word has one of length cubic in its size, it does not matter for DFA if ℓ is written in binary or in unary. In contrast, as our Example 2 shows, NWA may need an exponentially long word for synchronization; this explains the choice of the setting above. (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.)

4 Upper Bound of Theorem 6

In this section, we show that the following problem is in **PSPACE**: Given a nested word automaton $\mathcal{A}$ and an integer $\ell \geq 1$ written in binary, decide if there exists a synchronizing word for $\mathcal{A}$ of length at most ℓ. In fact, we can also adjust our arguments (see Subsect. 4.2) so that they give a **PSPACE** upper bound for another problem: Given a nested word automaton $\mathcal{A}$, 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 to F.

The plan of the proof is as follows. We encode nested words using binary trees (Subsect. 4.1), so that runs of NWA correspond to computations of tree automata and synchronizing words to tuples of such computations (Subsect. 4.2). 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 8); this is the same as guessing an exponentially large binary tree subject to local conditions. We prove that it's 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 (Subsect. 4.4); our solution relies on the concepts of pebble games and Strahler numbers (Subsect. 4.3).

4.1 Binary Tree Representation of Nested Words

In this subsection we describe a representation of nested words with binary trees used in the sequel. Because of space constraints, we only give a short summary.

Nested Words as Binary Trees. We denote the binary *tree representation* of a nested word u by $\mathsf{bin}(u)$. The explicit construction of $\mathsf{bin}(u)$ is not sophisticated, but we only describe the result. Nodes of $\mathsf{bin}(u)$ come in several different *types*. We did 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.

In general, 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 $i < j$, 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 $u = u'$.*

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.

4.2 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., [4]): such an automaton is a tuple $\mathcal{T} = (\mathcal{Q}, \mathcal{Q}^{\mathsf{f}}, \Delta)$ 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 7. *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 (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 $\mathcal{A}$.*

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., [4, Sect. 1.3]).

Lemma 8. *An NWA $\mathcal{A}$ with states $Q = \{q_1, \ldots, q_n\}$ has a synchronizing word of length at most ℓ iff there exists a state $\bar{q} \in Q$ such that the product automaton $\mathfrak{A}_{\bar{q}} = \mathcal{T}(q_1, \bar{q}) \times \ldots \times \mathcal{T}(q_n, \bar{q}) \times \mathcal{N}_\ell$ accepts some tree over $\mathcal{F}$. Here $\mathcal{N}_\ell$ 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.*

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 Subsect. 4.4; 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 8 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 Subsect. 4.1.

4.3 Pebble Games and Strahler Numbers

In this subsection we recall a classic idea that we use in the proof of Lemma 9 in the following Subsect. 4.4. We believe that the involved concepts, albeit classic, deserve more attention from our community than they have hitherto received.

An instance of the (*black*) *pebble game* (see, e.g., [23, 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 a pebble 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 [13,19]. 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 [8]. For trees, the pebbling number can be computed inductively [15]: if t is a single-vertex tree, then $\mathsf{peb}(G) = 1$; suppose t has principal subtrees $t_1, \ldots, t_d$ 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 [8,11]. 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 last characterization or from the recurrence above, is that the Strahler number of an m-node tree does not exceed $\lfloor \log_2(m+1) \rfloor - 1$ (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 subsection.

4.4 Bounded Nonemptiness for Implicitly Presented Tree Automata

Here we combine the ideas from Subsects. 4.2 and 4.3 to prove the upper bound in Theorem 6.

Lemma 9. *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 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 tight by Lemma 5 if no m is given). The upper bound on the size of the tree significantly shrinks the search space, so we refer to this problem as *bounded nonemptiness*. Assuming this lemma, the proof of the upper bound of Theorem 6 goes as follows.

Proof (upper bound of Theorem 6). Combine Lemmas 8 and 9 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 8, 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 (1) in Subsect. 4.1. First note that the state $\bar{q}$ can be guessed in polynomial space. Then recall from Claim 2 in Subsect. 4.2 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 9 with $m = 2\ell - 1$. This concludes the proof. □

To prove Lemma 9, we design a decision procedure using the pebbling strategy for trees that we discussed in Subsect. 4.3.

Proof (of Lemma 9). 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 Subsect. 4.3. Our procedure will guess moves of this strategy on the fly and simulate them; it will also guess the tree t in lockstep. 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. This completes the proof. □

The idea of the proof of Lemma 9 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 9.

5 Lower Bound of Theorem 6

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.

5.1 Small-Cost Synchronizing Words in DFA

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

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

State	Σ	#	call(γ)	ret(Γ)
$q \in Q$	$t_{q,a}$	p_q	$\gamma = \mathsf{x}$ self-loop	self-loop
force	self-loop	p_q for some 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 pay(q, a)	self-loop
p_q	self-loop	p_q	$\gamma = ☹$ in of punish(q)	self-loop
$s \in$ pay(q, a)	self-loop	p_q	See gadget pay in Figure 4 (left) where: • missing transitions go to state err of the same pay(q, a) • from out, the transition ret$(£)$ goes to $\Delta(q, a)$ • from err, the transition ret$(£)$ goes to p_q	
$s \in$ punish(q)	self-loop	p_q	See gadget punish in Figure 4 (right) where: • missing transitions go to state in of the same punish(q) • from out, the transition ret$(☹)$ goes to q	

Theorem 10. *The small-cost synchronizing word problem is* **PSPACE***-complete.*

The upper bound is guess-and-check: any synchronizing word w with $\mathsf{cost}(w) \leq \mathsf{budget}$ has $|w| \leq \mathsf{budget}$, since $\mathsf{cost}(a) \geq 1$ for all $a \in \Sigma$. The lower bound is by a reduction from the *careful synchronization* problem. Carefully synchronizing words [16] are a generalization of synchronizing words to finite-state automata with a partial transition function. Theorem 10 strengthens **PSPACE**-hardness results for similar models [5,12]: the key difference is that in our setting the cost function can only depend on input letters and not on individual transitions.

5.2 Reduction to Short Synchronizing Nested Word

We prove the **PSPACE**-hardness of Short Synchronizing Nested Word by a reduction from the small-cost synchronizing word problem: given a DFA $\mathcal{D} = (Q, \Delta)$ over Σ, $\mathsf{cost}\colon \Sigma \to \mathbb{Z}_{>0}$, and $\mathsf{budget} \in \mathbb{Z}_{>0}$, we find an 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 $\mathcal{D}$ will be, 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 pay(q, a) (there is a separate copy for each $q \in Q$). The intention is that the length of a nested word read by $\mathcal{A}$ corresponds to the cost of some word read by $\mathcal{D}$. Obviously, there will be runs of $\mathcal{A}$ that have structure 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 punish(q), $q \in Q$. When

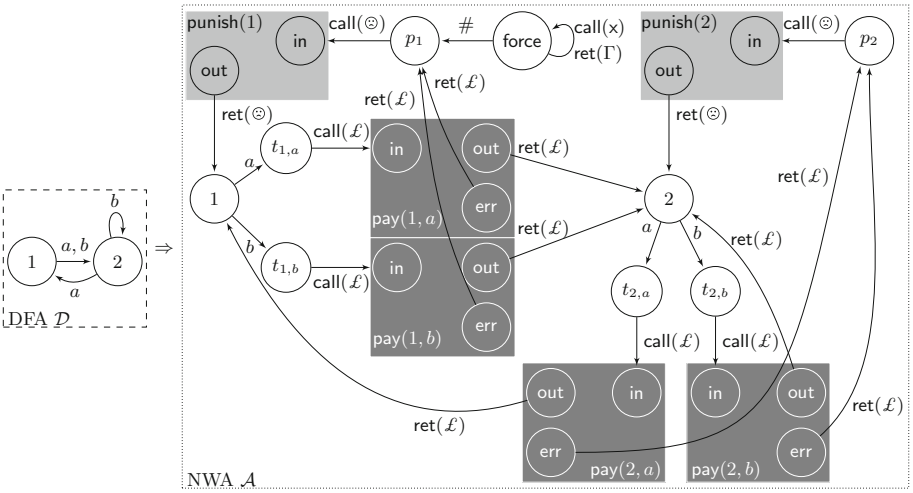

Fig. 3. An example of the reduction to the Short Synchronizing Nested Word. For $q \in \{1, 2\}$, all #-transitions from q and from all states of gadgets $\mathsf{pay}(q, a)$, $\mathsf{pay}(q, b)$, and $\mathsf{punish}(q)$ lead to p_q. All a, b-transitions in all states are self-loops, except in states 1, 2. The NWA $\mathcal{A}$ has a synchronizing nested word of length $4 \cdot \mathsf{budget} + |w_{\mathsf{punish}}| + 1$ if and only if $\mathcal{D}$ has a synchronizing word with cost at most budget.

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; a small example is shown in Fig. 3. The set of states in $\mathcal{A}$ is $Q \cup \{\mathsf{force}\} \cup \bigcup_{q \in Q, a \in \Sigma} (\mathsf{pay}(q, a) \cup \{t_{q,a}\}) \cup \bigcup_{q \in Q} (\mathsf{punish}(q) \cup \{p_q\})$ 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 6, all call and return positions are assumed to have "fake" input letters call and ret). Table 1 describes transitions of $\mathcal{A}$.

It remains to define the gadgets $\mathsf{pay}(q, a)$ and $\mathsf{punish}(q)$. Recall that they need to let through runs on nested words w_a and w_{punish}; deviations are considered

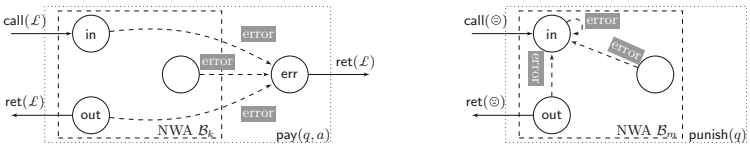

Fig. 4. Gadgets $\mathsf{pay}(q, a)$ (on the left) and punish (on the right) where $\mathcal{B}_k, \mathcal{B}_m$ are described in Example 2 with $k = 4 \cdot \mathsf{cost}(a) - 3$ and $m = |w_{\mathsf{punish}}| - 2$

cheating and are handled appropriately. We base the construction of $\mathsf{pay}(q, a)$ and $\mathsf{punish}(q)$ on the family of NWA $\mathcal{B}_n$ from Example 2; see Fig. 4. 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 $\mathsf{pay}(q, a)$ 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 $\mathsf{cost}(a) \geq 1$. Our choice for m in $\mathcal{B}_m$ will be given below. Now recall that the NWA $\mathcal{B}_n$ in Example 2 had only partially defined transition functions; we make them complete by directing all missing transitions (shown as "errors" in Fig. 4) 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.

In fact, every synchronizing word is forced to have at least one occurrence of #, otherwise the run starting from yet another state force cannot be synchronized with other runs. Therefore, every synchronizing word needs to have at least one occurrence of w_{punish}, and this determines our choice of ℓ and $|w_{\mathsf{punish}}|$. It is natural to pick $\ell = 1 + |w_{\mathsf{punish}}| + 4 \cdot \mathsf{budget}$; since we want to have $\ell < 2 \cdot |w_{\mathsf{punish}}|$, we need to make sure that $|w_{\mathsf{punish}}| > 4 \cdot \mathsf{budget} + 1$. We thus choose $m + 2 = |w_{\mathsf{punish}}| = 4 \cdot \mathsf{budget} + 2$ and $\ell = 8 \cdot \mathsf{budget} + 3$.

This completes the description of our reduction; we omit the proof of correctness because of space constraints. This reduction provides the lower bound in Theorem 6.

Acknowledgements. The authors are grateful to Michael Wehar for comments.

References

1. Alur, R., Madhusudan, P.: Adding nesting structure to words. J. ACM **56**(3), 16 (2009)
2. Caucal, D.: Synchronization of pushdown automata. In: Ibarra, O.H., Dang, Z. (eds.) DLT 2006. LNCS, vol. 4036, pp. 120–132. Springer, Heidelberg (2006)
3. Chytil, M.P., Monien, B.: Caterpillars and context-free languages. In: Choffrut, C., Lengauer, T. (eds.) STACS 90. LNCS, vol. 415, pp. 70–81. Springer, Heidelberg (1990)
4. Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree Automata Techniques and Applications, 12 October 2007. http://www.grappa.univ-lille3.fr/tata
5. Doyen, L., Juhl, L., Larsen, K.G., Markey, N., Shirmohammadi, M.: Synchronizing words for weighted and timed automata. In: 34th International Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS, 15–17 December 2014, New Delhi, India, pp. 121–132 (2014)
6. Driscoll, E., Thakur, A., Reps, T.: OpenNWA: a nested-word automaton library. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 665–671. Springer, Heidelberg (2012)

7. Eppstein, D.: Reset sequences for monotonic automata. SIAM J. Comput. **19**(3), 500–510 (1990)
8. Ershov, A.P.: On programming of arithmetic operations. Commun. ACM **1**(8), 3–9 (1958)
9. Esparza, J., Ganty, P., Kiefer, S., Luttenberger, M.: Parikh's theorem: a simple and direct automaton construction. Inf. Process. Lett. **111**(12), 614–619 (2011)
10. Esparza, J., Ganty, P., Majumdar, R.: Parameterized verification of asynchronous shared-memory systems. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 124–140. Springer, Heidelberg (2013)
11. Esparza, J., Luttenberger, M., Schlund, M.: A brief history of Strahler numbers. In: Dediu, A.-H., Martín-Vide, C., Sierra-Rodríguez, J.-L., Truthe, B. (eds.) LATA 2014. LNCS, vol. 8370, pp. 1–13. Springer, Heidelberg (2014)
12. Fominykh, F.M., Martyugin, P.V., Volkov, M.V.: P(l)aying for synchronization. Int. J. Found. Comput. Sci. **24**(6), 765–780 (2013)
13. Hopcroft, J.E., Paul, W.J., Valiant, L.G.: On time versus space. J. ACM **24**(2), 332–337 (1977)
14. Kozen, D.: Lower bounds for natural proof systems. In: 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October–1 November, pp. 254–266 (1977)
15. Lengauer, T., Tarjan, R.E.: The space complexity of pebble games on trees. Inf. Process. Lett. **10**(4/5), 184–188 (1980)
16. Martyugin, P.: Computational complexity of certain problems related to carefully synchronizing words for partial automata and directing words for nondeterministic automata. Theor. Comput. Syst. **54**(2), 293–304 (2014)
17. Mehlhorn, K.: Pebbling moutain ranges and its application of DCFL-recognition. In: Proceedings Automata, Languages and Programming, 7th Colloquium, Noordweijkerhout, The Netherland, July 14–18, pp. 422–435 (1980)
18. Olschewski, J., Ummels, M.: The complexity of finding reset words in finite automata. In: Hliněný, P., Kučera, A. (eds.) MFCS 2010. LNCS, vol. 6281, pp. 568–579. Springer, Heidelberg (2010)
19. Paterson, M.S., Hewitt, C.E.: Comparative schematology. In: Record of the Project MAC Conference on Concurrent Systems and Parallel Computation, pp. 119–127. ACM, MIT AI Memo AIM-201 (1970). http://hdl.handle.net/1721.1/5851
20. Pin, J.-É.: On two combinatorial problems arising from automata theory. North-Holland Math. Stud. **75**, 535–548 (1983)
21. Rystsov, I.K.: Polynomial complete problems in automata theory. Inf. Process. Lett. **16**(3), 147–151 (1983)
22. Sandberg, S.: Homing and synchronizing sequences. In: Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., Pretschner, A. (eds.) Model-Based Testing of Reactive Systems. LNCS, vol. 3472, pp. 5–33. Springer, Heidelberg (2005)
23. Savage, J.E.: Models of Computation - Exploring the Power of Computing. Addison-Wesley, Boston (1998)
24. Černý, J., Pirická, A., Rosenauerová, B.: On directable automata. Kybernetika **7**(4), 289–298 (1971)
25. Volkov, M.V.: Synchronizing automata and the Černý conjecture. In: Martín-Vide, C., Otto, F., Fernau, H. (eds.) LATA 2008. LNCS, vol. 5196, pp. 11–27. Springer, Heidelberg (2008)
26. Wehar, M.: Hardness results for intersection non-emptiness. In: Esparza, J., Fraigniaud, P., Husfeldt, T., Koutsoupias, E. (eds.) ICALP 2014, Part II. LNCS, vol. 8573, pp. 354–362. Springer, Heidelberg (2014)