An Infinite Automaton Characterization of Double Exponential Time[⋆]

Salvatore La Torre[1], P. Madhusudan[2], and Gennaro Parlato[1,2]

[1] Università di Salerno, Italy
[2] University of Illinois, Urbana-Champaign, USA

Abstract. Infinite-state automata are a new invention: they are automata that have an infinite number of states represented by words, transitions defined using rewriting, and with sets of initial and final states. Infinite-state automata have gained recent interest due to a remarkable result by Morvan and Stirling, which shows that automata with transitions defined using rational rewriting precisely capture context-sensitive (NLINSPACE) languages. In this paper, we show that infinite automata defined using a form of multi-stack rewriting precisely defines double exponential time (more precisely, 2ETIME, the class of problems solvable in $2^{2^{O(n)}}$ time). The salient aspect of this characterization is that the automata have no ostensible limits on time nor space, and neither direction of containment with respect to 2ETIME is obvious. In this sense, the result captures the complexity class qualitatively, by restricting the power of rewriting.

1 Introduction

The theory of infinite-state automata is a new area of research (see [21] for a recent survey). Infinite-state automata (not to be confused with *finite state automata on infinite words*) are automata with *infinitely* many states that can read *finite* words and accept or reject them, in much the same way as finite-state automata would. In order to represent infinite-state automata using finite means, the states, the transitions, and the initial and final state sets are represented *symbolically*.

The infinite-state automata we study in this paper are defined by using *words* to represent the states of the automaton. Let us fix a finite alphabet Σ as the input alphabet for the infinite-state automata. The set of states of an infinite-state automaton over Σ are words over a finite alphabet Π (which does not need to be related to Σ in any way). The initial and final sets of states of this automaton are defined using word-languages over Π accepted by finitely presented devices (e.g. finite-state automata over Π). Transitions between states are defined using *rewriting* rules that rewrite words to other words: for each $a \in \Sigma$, we have a rewrite rule that rewrites words over Π. A state $u \in \Pi^*$ leads to state $u' \in \Pi^*$ on $a \in \Sigma$ iff the rewrite rule for a can rewrite u to u'. There is a variety of choices for the power of rewriting, but in any case the rewriting rules are presented finitely (e.g. using finite transducers). The language accepted by an infinite-state automaton is defined in the natural way: a word $w \in \Sigma^*$ is accepted if there is a path from *some* initial state to *some* final state tracing w in the automaton.

[⋆] The first and third authors were partially supported by the MIUR grants ex-60% 2006 and 2007 Università degli Studi di Salerno.

Infinite-state automata are naturally motivated in *formal verification*, where, intuitively, a *state* of the model can be represented using a word, and the system's evolution can be described using rewriting rules. Classic examples include boolean abstraction of recursive programs [2] (where a system is described using a state and a stack encoded into words, and the rewriting corresponds to *prefix rewriting*) and *regular model-checking*, where parameterized finite-state systems are represented with finite words and transitions defined using *synchronous rational rewriting* [3].

Infinite-state automata are radically different computation models than Turing machines especially when computational complexity issues are at hand. The notion that rewriting words (or terms) can be a basis for defining computability goes back to the works of Axel Thue [22] (Thue systems) and Emil Post [17] (Post's tag systems). Formal languages defined using grammars (the Chomsky hierarchy) are also in the spirit of rewriting, with semi-Thue systems corresponding to unrestricted grammars and hence Turing machines. While Turing machines can be viewed as rewrite systems (rewriting one configuration to another), the study of computational complexity is often based on time and space constraints on the Turing machine model, and natural counterparts to complexity classes in terms of rewrite systems don't currently exist.

Given a word $w \in \Sigma^*$, note that infinite automata have possibly an *infinite* number of paths on w. Hence, deciding whether w is accepted by the infinite-state automaton is in no way trivial. However, if rewriting rules can be simulated by Turing machines (which will usually be the case), the language accepted by the infinite-state automaton is recursively enumerable.

Recently, Morvan and Stirling showed the following remarkable result: infinite state automata where states are finite words, initial and final sets are defined using regular languages, and transitions are defined using *rational relations*, accept precisely the class of *context-sensitive languages* (nondeterministic linear-space languages) ([15]; see also [5]). Rational relations are relations $R \subseteq \Pi^* \times \Pi^*$ that can be effected by finite-state automata: $(u, u') \in R$ iff the automaton can read u on an input tape and write u' on the output tape, where the two tape heads are only allowed to move right (but can move independent of each other).

Note that the only constraint placed in the above result is the power of rewriting (rational relations) and there is no ostensible limit on space or time. In other words, the constraint on rewriting is a *qualitative constraint* with no apparent restriction on complexity. Indeed, even establishing the upper bound (the easier direction), namely that these automata define languages that are accepted by linear-bounded Turing machines is non-trivial. A naive *simulation* of the infinite-state automaton will not work as the words that represent states on the run can be unboundedly large even for a fixed word w. Notice that we do *not* allow ϵ-transitions in infinite automata, as allowing that would make infinite automata with even regular rewriting accept the class of all recursively enumerable languages.

Our main contribution in this paper is an infinite automaton characterization for the class 2ETIME, the class of languages accepted by Turing machines in time $exp(exp(O(n)))$[3], using a qualitative constraint on rewriting, which is a restricted form of multi-stack pushdown rewriting.

[3] $exp(x)$ denotes 2^x.

A simple generalization of regular rewriting is *pushdown rewriting*, where we allow the rewriting automata the use of a work stack which can be used for intermediate storage when rewriting a word to another. For example the relation $\{(w, ww^r) | w \in \Pi^*\}$ (where w^r denotes the reverse of w) is not regular but can be effected by a pushdown rewrite system. However, defining infinite automata with the power of pushdown rewriting quickly leads to *undecidability* of the membership problem, and these automata can accept non-recursive languages.

We hence place a restriction on pushdown rewriting. We demand that the rewriting device takes its input in a read-only tape and writes it to a write-only tape, and has access to some stacks, but it can switch only a *bounded* number of times the source from which it is reading symbols (i.e., the input tape and the stacks). In other words, the pushdown rewriting can be split into k phases, where in each phase, it either reads from the input tape and does not pop any stack, or pops from just one stack but doesn't read from the input tape. This restriction puts the problem of checking membership just within the boundary of decidability, and results in an automaton model that defines a class of recursive languages.

We show that infinite automata restricted to bounded-phase pushdown rewriting precisely defines the class 2ETIME.

The upper bound, showing the membership problem for any such infinite automaton is decidable in 2ETIME, is established by reducing it to the *emptiness* problem for *finite-phased multi-stack visibly pushdown automata*, which we have shown recently to be decidable [12]. Note that (non-deterministic) Turing machines that directly and naively simulate the infinite automaton could take unbounded space and time. Visibly pushdown automata [1] are pushdown automata where the input symbols determine the operation on the stack, and multi-stack visibly pushdown automata generalize them to multiple stacks [12]. Intuitively, the *accepting runs* that are followed by an n-stack pushdown rewriting system when it transforms a word u to u' can be seen as a multi-stack ($(n + 2)$-stack) *visibly* pushdown automaton. Hence the problem of membership of w for an infinite automaton reduces to the *emptiness problem* of the language of accepting runs over w. Moreover, if each rewriting step in the infinite automaton is bounded phase, then the number of phases in the multi-stack automata is $O(|w|)$. In [12], we show that the k-phase reachability for multi-stack automata is solvable in time $exp(exp(O(poly(k))))$ using (monadic second-order) logic interpretations on finite trees. We sharpen the above result in this paper to obtain $exp(exp(O(k)))$ time decision procedure for emptiness by implementing two crucial subprocedures that correspond to capturing the linear ordering and the successor relation from the tree directly using nondeterministic tree automata and two-way alternating tree automata, respectively.

Turning to the lower bound, we establish that all 2ETIME languages are accepted by infinite automata defined using bounded-phase pushdown rewriting. We show that for every alternating ESPACE Turing machine (i.e. working in space $2^{O(n)}$, which is equivalent to 2ETIME [9]), there is an infinite automaton with bounded-phase rewriting accepting the same language.

Related Work: A recent result by Rispal [18] shows that infinite automata defined using *synchronous rational relations*, which are strictly less powerful than rational relations, also define exactly the class of context-sensitive languages (see also [5]). Meyer [14] has characterized the class ETIME (the class of languages accepted by Turing machines in time $exp(O(n))$) with infinite automata defined via automatic term transducers.

Bounded-phase visibly multi-stack pushdown automata have been introduced and studied by us in [12]. These automata capture a robust class of context-sensitive languages that is closed under all the boolean operations and has decidable decision problems. Also, they turned out to be useful to show decidability results for concurrent systems communicating via unbounded FIFO queues [13].

Capturing complexity classes using logics on graphs in descriptive complexity theory [10], which was spurred by Fagin's seminal result capturing NP using $\exists SO$, also has the feature that the characterizations capture complexity classes without any apparent restriction of time or space.

Finally, there's a multitude of work on characterizing the infinite graphs that correspond to restricted classes of machines (pushdown systems [16], prefix-recognizable graphs [7], higher-order pushdown automata [4], linear-bounded automata [6], and the entire Chomsky hierarchy [8]).

2 Multi-stack pushdown rewriting

A multi-stack pushdown transducer is a transducer from words to words that has access to one or more pushdown stacks.

For any set X, let X_ϵ denote $X \cup \{\epsilon\}$, and let X^* denote the set of finite words over X. Also, for any $i, j \in \mathbb{N}$, let $[i, j]$ denote the set $\{i, i+1, \ldots, j\}$.

Fix finite alphabets Π and Γ. An n-stack pushdown transducer over Π is a tuple $\mathcal{T} = (Q, q_0, \delta, \Gamma, F)$ where Q is a finite set of states, $q_0 \in Q$ is the initial state, Γ is the stack alphabet, and $F \subseteq Q$ is the set of final states. The transition relation is $\delta \subseteq (Q \times Q \times \Pi_\epsilon \times \Pi_\epsilon \times [0, n] \times \Gamma_\epsilon \times \Gamma_\epsilon)$, with the restriction that if $(q, q', a, b, i, \gamma, \gamma') \in \delta$, then $\gamma = \gamma' = \epsilon$ iff $i = 0$.

A transition of the form $(q, q', a, b, i, \gamma, \gamma')$, with $a, b \in \Pi_\epsilon$ and $\gamma, \gamma' \in \Gamma_\epsilon$, intuitively means that the pushdown transducer, when in state q with γ on the top of its i'th stack (provided $i > 0$) can read a from the input tape, write b onto the output tape, replace γ with γ' onto the i'th stack, and transition to state q'. When $i = 0$, $\gamma = \gamma' = \epsilon$ and hence no stack is touched when changing state though input symbols can be read.

Note that $\gamma = \epsilon$ and $\gamma' \neq \epsilon$ corresponds to a push transition, $\gamma \neq \epsilon$ and $\gamma = \epsilon$ corresponds to a pop transition. Without loss of generality, let us assume that in every transition, $\gamma = \epsilon$ or $\gamma' = \epsilon$ holds, and if $a \neq \epsilon$ then $\gamma = \epsilon$ (i.e., when reading a symbol from the input tape, none of the stacks can be popped).

A configuration of the pushdown transducer $\mathcal{T}$ is a tuple $(w_1 q w_2, \{s_i\}_{i=1}^n, w')$ where $w_1, w_2, w' \in \Pi^*$, $q \in Q$ and $s_i \in \Gamma^*$ for each $i \in [1, n]$. Such a configuration means that the input head is positioned just after w_1 on the input tape that has $w_1 w_2$ written on it, q is the current state, s_i is the current content of the i'th stack, and w' is the output written thus far onto the output tape (with the head positioned at the end of w').

Transitions between configurations are defined by moves in δ as follows:

$$(w_1 q a w_2, \{s_i\}_{i=1}^n, w') \xrightarrow{(q,q',a,b,j,\gamma,\gamma')} (w_1 a q' w_2, \{s'_i\}_{i=1}^n, w'b),$$

where $(q, q', a, b, j, \gamma, \gamma') \in \delta$, if $j \neq 0$ then $s_j = \widehat{s}\gamma$ and $s'_j = \widehat{s}\gamma'$, and $s'_i = s_i$ for each $i \neq j$.

Let us define the configuration graph of the transducer $\mathcal{T}$ as the graph whose vertices are the configurations and whose edges are the transitions between configurations as defined above.

A multi-stack pushdown transducer $\mathcal{T}$ rewrites w to w', if there is a path in the configuration graph from configuration $(q_0 w, \{\epsilon\}_{i=1}^n, \epsilon)$ to configuration $(w q_f, \{s_i\}_{i=1}^n, w')$, with $q_f \in F$.

Pushdown rewriting is powerful, and the problem of deciding whether w can be rewritten to w' even in two steps by even a one-stack transducer is undecidable (see Appendix for a proof):

Lemma 1. *The problem of checking if a word w can be rewritten to a word w' in two steps by a 1-stack pushdown transducer is undecidable.*

We want a tractable notion of transducers in order to define infinite automata that accept recursive languages. We hence introduce a bounded version of pushdown transducers.

We say that a pushdown transducer is *k-phase* ($k \in \mathbb{N}$), if, when transforming any w_1 to w_2, it switches at most k times between reading the input and popping either one of the stacks, and between popping different stacks. More formally, a transition of the form $(q, q', a, b, i, \gamma, \gamma')$ is a *not-pop transition* if it's not a transition that pops any stack, i.e. if $\gamma' \neq \epsilon$ or $i = 0$. Let $NotPop$ denote the set of not-pop transitions. Let Pop_i ($i \neq 0$) denote the set of all transitions except those that read from the input tape or pop from a stack j different from i, i.e. Pop_i is the set of transitions of the form $(q, q', a, b, j, \gamma, \gamma')$ where $a = \epsilon$ and if $j \neq i$ then $\gamma = \epsilon$.

A k-phase transducer is one which on any run $c_0 \xrightarrow{m_1} c_1 \xrightarrow{m_2} c_2 \ldots \xrightarrow{m_i} c_i$ the sequence $m_1 m_2 \ldots m_i$ can be split as $w_1 w_2 \ldots w_k$ where for every $h \in [1, k]$, $w_h \in NotPop^* \cup \bigcup_{i=1}^n (Pop_i^*)$.

A bounded-phase pushdown transducer is a pushdown transducer which is k-phase for some $k \in \mathbb{N}$.

Infinite automata defined by multi-stack pushdown transducers

We define now *infinite-state* automata over an alphabet Σ. The states in this automaton will correspond to words over an alphabet Π, the set of states one can transition to from a state on a letter d in Σ will be defined using a multi-stack pushdown transducer corresponding to d, and initial and final state sets will be identified using regular sets of words over Π.

Fix a finite alphabet Σ. An infinite-state pushdown transducer automaton (PTA) over Σ is a tuple $\mathcal{A} = (\Pi, \{\mathcal{T}_d\}_{d \in \Sigma}, Init, Final)$, where Π is a finite alphabet, for each $d \in \Sigma$, $\mathcal{T}_d$ is a pushdown transducer over Π, and $Init$ and $Final$ are finite-state automata (NFAs) over Π.

A PTA $\mathcal{A} = (\Pi, \{\mathcal{T}_d\}_{d\in\Sigma}, \mathit{Init}, \mathit{Final})$ defines an infinite graph $G = (V, E)$ defined as follows:

- The set of vertices V is the set of words over Π
- $v \xrightarrow{d} v'$ iff the pushdown transducer $\mathcal{T}_d$ can rewrite v to v'.

A bounded-phase PTA (BPTA) is a PTA in which every transducer is of bounded-phase.

A run of the PTA $\mathcal{A}$ on a word over $d_1 \ldots d_n \in \Sigma^*$ is a sequence $v_0, v_1, \ldots v_n$, where v_0 is accepted by the automaton Init, and for each $i \in [1, n]$, $v_{i-1} \xrightarrow{d_i} v_i$ is in G. Such a run is accepting if the final vertex is accepted by Final, i.e. $v_n \in L(\mathit{Final})$.

A word w is accepted by a PTA $\mathcal{A}$ if there is some accepting run of $\mathcal{A}$ on w. The language accepted by $\mathcal{A}$, denoted $\mathcal{L}(A)$ is the set of all words it accepts.

In the rest of the paper we often write $exp(x)$ for 2^x. Let 2ETIME(Σ) denote the class of all languages over Σ that can be accepted by Turing machines working in time $exp(exp(O(n)))$.

We can now state our main theorem:

Theorem 1. *A language L over Σ is accepted by a bounded-phase PTA iff $L \in$2ETIME(Σ).*

3 The Upper Bound

In this section, we show that bounded-phase pushdown transducer automata define a class of languages contained in 2ETIME.

Let us fix a BPTA $\mathcal{A} = (\Pi, \{\mathcal{T}_d\}_{d\in\Sigma}, \mathit{Init}, \mathit{Final})$. The proof that $L(\mathcal{A})$ is contained in 2ETIME is structured as follows:

(a) First, we show that the problem of checking if a word w is accepted by a BPTA can be reduced to the *emptiness* problem for k-phase multi-stack visibly pushdown automata (defined below) of state-space $O(|w|)$ and such that $k = O(|w|)$.

(b) In [12], we have shown that the emptiness problem for k-phase multi-stack pushdown automata with state-space Q can be decided in time $exp(|Q| \cdot exp(O(poly(k))))$. Applying this would give a 2EXPTIME procedure and not a 2ETIME procedure for our problem (2EXPTIME is the class of problems that can be solved by a Turing machine using $exp(exp(O(poly(n))))$ time). Consequently, we sharpen the result above, and show that emptiness can be indeed decided in time $exp(|Q| \cdot exp(O(k)))$, which establishes our theorem.

Bounded phase multi-stack pushdown automata

Multi-stack visibly pushdown automata (MVPA) are automata with a finite number of stacks, where the input letter determines which stack the automaton touches and whether it pushes or pops from that stack. We refer to actions that push onto a stack as calls and actions that pop a stack as returns.

An n-stack call-return alphabet is a tuple $\widetilde{\Sigma}_n = \langle \{(\Sigma_c^i, \Sigma_r^i)\}_{i\in[1,n]}, \Sigma_{int} \rangle$ of pairwise disjoint finite alphabets. For any $i \in [1, n]$, Σ_c^i is a finite set of *calls of the stack* i,

Σ_r^i is a finite set of *returns of stack* i, and Σ_{int} is a finite set of *internal actions*. Let $\widetilde{\Sigma}$ denote the union of all the alphabets in $\widetilde{\Sigma}_n$.

An n-stack visibly pushdown automaton $M = (Q, Q_I, \Gamma, \delta, Q_F)$ (where Q is a finite set of states, $Q_I \subseteq Q$ and $Q_F \subseteq Q$ are initial and final sets of states, Γ is the stack alphabet and δ is the transition relation) over such an alphabet can push on the i'th stack exactly one symbol when it reads a call of the i'th call alphabet, and pop exactly one symbol from the i'th stack when it reads a return of the i'th return alphabet. Also, it cannot touch any stack when reading an internal letter. The semantics of MVPAs is defined in the obvious way, and we refer the reader to [12] for details.

A k-phase MVPA (k-MVPA) is intuitively an MVPA which works in (at most) k phases, where in each phase it can push onto *any* stack, but pop at most from one stack. Formally, given a word $w \in \widetilde{\Sigma}^*$, we denote with $Ret(w)$ the set of all returns in w. A word w is a *phase* if $Ret(w) \subseteq \Sigma_r^i$, for some $i \in [1, n]$, and we say that w is a *phase of stack* i. A word $w \in \widetilde{\Sigma}^+$, is a k-phase word if k is the minimal number such that w can be factorized as $w = w_1 w_2 \dots w_k$, where w_h is a phase for each $h \in [1, k]$. Let $Phases(\widetilde{\Sigma}_n, k)$ denote the set of all k-phase words over $\widetilde{\Sigma}_n$.

For any k, a *k-phase multi-stack visibly pushdown automaton* (k-MVPA) A over $\widetilde{\Sigma}_n$ is an MVPA M parameterized with a number k; the language accepted by A is $L(A) = L(M) \cap Phases(\widetilde{\Sigma}_n, k)$.

Reduction to *k*-MVPA emptiness

Consider a BPTA $\mathcal{A} = (\Pi, \{\mathcal{T}_d\}_{d \in \Sigma}, Init, Final)$. Recall that given a word $w = d_1 \dots d_m \in \Sigma^*$, the automaton $\mathcal{A}$ accepts w iff there is a sequence of words $u_0, \dots, u_m$ such that $u_0 \in L(Init)$, $u_m \in L(Final)$, and for each $i \in [1, m]$, u_{i-1} can be rewritten to u_i by the transducer $\mathcal{T}_{d_i}$.

Suppose that the transducers of $\mathcal{A}$ have at most n stacks. We consider the $(n+2)$-stack call-return alphabet $\widetilde{\Sigma}_{n+2} = \langle \{(\Sigma_c^i, \Sigma_r^i)\}_{i \in [1, n+2]}, \{int\} \rangle$ where each $\Sigma_c^i = \{c_i\}$ and $\Sigma_r^i = \{r_i\}$. I.e., we have exactly one call and one return for each stack, and exactly one internal letter.

Assume that an $(n+2)$-stack MVPA starts with u_{i-1}^r on stack 1. Using stacks $2, \dots, n+1$ as the intermediate stacks, it can generate u_i on stack $n+2$ by simulating the transducer $\mathcal{T}_{d_i}$ (the word it reads is dictated by the actions performed on the stack). Then, it can replace stack 1's content with the reverse of stack $(n+2)$'s content to get u_i^r on the stack 1, and empty stacks $2, \dots, n+1$. Since the pushdown rewrite system is bounded phase, it follows that the above rewriting takes only a bounded number of phases. Simulating the rewrites for the entire word w (i.e. $u_0 \to u_1 \to \dots u_m$), and checking the initial words and final words belong to $Init$ and $Final$, respectively, takes at most $O(m)$ phases. Moreover, we can build this MVPA to have $O(m)$ states (for a fixed BPTA $\mathcal{A}$). We hence have:

Lemma 2. *The problem of checking whether w is accepted by a fixed PTA is polynomial-time reducible to the emptiness problem of an $O(|w|)$-phase* MVPA *with $O(|w|)$ states.*

Solving *k*-MVPA emptiness

In [12], the decidability of emptiness of k-MVPA proceeds by first defining a map from words over $\widetilde{\Sigma}$ to trees, called *stack trees*, by showing that the set of stack trees that correspond to words forms a *regular* set of trees, and reducing k-MVPA emptiness to emptiness of tree automata working on the corresponding stack trees.

The map from words to trees rearranges the positions of the word into a binary tree by encoding a matching return of a call as its right child. This mapping hence easily captures the matching relation between calls and returns, but loses sight of the linear order in w. Recovering the linear order is technically hard, and is captured using monadic second-order logic (MSO) on trees.

Fix a k-phase word w of length m. We say that a factorization $w_1, \ldots, w_k$ of w is *tight* if: (1) the first symbol of w_h is a return for every $h \in [2, k]$, (2) if $k > 1$ then $Ret(w_1) \neq \emptyset$, and (3) w_h and w_{h+1} are phases of different stacks for every $h \in [1, k-1]$. It is easy to see that, for every k-phase word w there is a unique tight factorization, and thus we can uniquely assign a phase number to each letter occurrence within w as follows: for $w = w'dw''$, $d \in \widetilde{\Sigma}$, the phase of d is h iff $w_1, \ldots, w_k$ is the tight factorization of w and d is within w_h.

A *stack tree* is defined as follows:

Definition 1. *Let w be a k-phase word over $\widetilde{\Sigma}_n$ with $|w| = m$, and $w_1, \ldots, w_k$ be the tight factorization of w. The word-to-tree map of w, $wt(w)$, which is a $(\widetilde{\Sigma} \times [1, k])$-labeled tree (V, λ), and the bijection $pos : V \rightarrow [1, m]$ are inductively defined (on $|w|$) as follows:*

- *If $m = 1$, then $V = \{root\}$, $\lambda(root) = (w, 1)$, and $pos(root) = 1$.*
- *Otherwise, let $w = w'd$, $d \in \widetilde{\Sigma}$, and $wt(w') = (V', \lambda')$. Then:*
  - *$V = V' \cup \{v\}$ with $v \notin V'$.*
  - *$\lambda(v) = (d, k)$ and $\lambda(v') = \lambda'(v')$, for every $v' \in V'$.*
  - *If there is a $j < m$ such that d is a return and the j'th letter of w is its matching call (of the same stack), then v is the right-child of $pos^{-1}(j)$.*
    Otherwise v is the left-child of $pos^{-1}(m-1)$.
  - *$pos(v) = m$.*

The tree $wt(w)$ is called the stack tree *of w. A k-stack tree is the stack tree of a k-phase word.*

The proof that the set of stack trees that correspond to words accepted by a k-MVPA forms a regular set of trees requires showing that: (a) the set of all stack trees is regular and (b) given a stack tree, checking whether a k-MVPA has an accepting run over the corresponding word can be done by a tree automaton.

Part (a) involves the definition of a linear order $\prec'$ on tree nodes which corresponds the linear order $\prec$ on the word from the stack tree, and [12] shows that given a tree automaton of size r accepting the $\prec'$ relation (formally, accepting trees with two nodes marked x and y such that $x \prec' y$), we can build an automaton of size exponential in r to accept all stack trees. It is further shown in [12] that the $\prec'$ relation can be captured by an automaton of size $r = exp(poly(k))$. In order to get a $exp(exp(O(k)))$

automaton for accepting stack trees, we show now that the $\prec'$ relation can be defined using automata of size $r = exp(O(k))$ (Lemma 4 below).

Part (b) requires traversing the stack tree according to the linear order on w using a *two-way alternating automaton*. We show below that there is a two-way alternating tree automaton of size $2^{O(k)}$ that traverses the tree consecutively from one node to its successor. More precisely, we show that given a tree where the first and last events of each phase are marked, there is a 2-way alternating automaton that, when placed at a node x in the tree, will navigate to the successor of x (reaching a final state) (Lemma 5 below). It follows from [12] that using this automaton, we can check whether the word corresponding to the stack tree is accepted by a k-MVPA using a nondeterministic automaton of size $exp(exp(O(k)))$. This primarily involves an exponential conversion of alternating tree automata to nondeterministic automata [19, 23], followed by other checks that can be effected by nondeterministic automata of similar size.

We present the above two results in two technical lemmas below (see the Appendix for more details of the proof).

Tree automata accepting stack trees

Here we prove that the $\prec'$ relation can be captured by an automaton of size $exp(O(k))$. To do that, we define a relation $\prec_*$ for which it is direct to build a tree automata of size $exp(O(k))$ that captures it, and then we show that $\prec_*$ coincides with $\prec'$.

For a $(\widetilde{\Sigma} \times [1, k])$-labeled tree $T = (V, \lambda)$, we define a map $phase_T : V \to [1, k]$ as $phase_T(x) = h$ iff $\lambda(x) = (a, h)$ for some $a \in \widetilde{\Sigma}$.

Stack trees must first satisfy some simple conditions. A tree is *well-formed* if (i) the phase numbers are monotonically increasing along any path in the tree, (ii) every right child is a return, with a call of the same stack as its parent, and (iii) the phase of the root is 1.

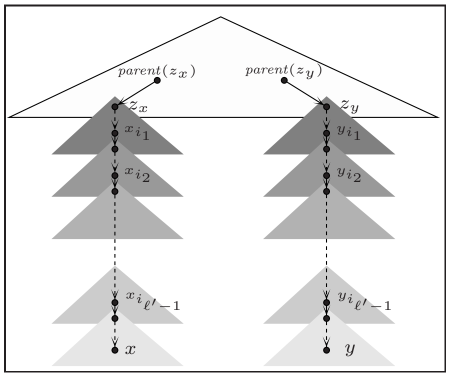
Let T be a well-formed tree, x be a node of T, x' be an ancestor of x, and $x_1 \ldots x_\ell$ be the path in T from x' to x. Let $I = \{i_1, i_2, \ldots, i_{\ell'-1}\}$ be the set of all indices $i \in [1, \ell - 1]$ such that $phase_T(x_i) \neq phase_T(x_{i+1})$. Assume that $i_1 < i_2 < \ldots < i_{\ell'-1}$. We denote by $PhasePath_T(x', x)$ the sequence $p_1, p_2, \ldots, p_{\ell'}$ such that $p_j = phase_T(x_{i_j})$ for every $j \in [1, \ell' - 1]$, and $p_{\ell'} = phase_T(x_\ell)$.

In the following, $<_{prefix}$ is the linear order of nodes according to a preorder visit of the tree, and T_z denotes the largest subtree of T which contains z and whose nodes are labeled with the same phase number as z.

Definition 2. *Let $T = (V, \lambda)$ be a well-formed tree. For every $x, y \in V$, $x \prec_* y$ if one of the following holds:*

1. *$phase_T(x) < phase_T(y)$;*
2. *$T_x = T_y$ and $x <_{prefix} y$;*
3. *There exists an ancestor z_x of x and an ancestor z_y of y such that*
   - *$z_x \neq z_y$,*
   - *$phase_T(parent(z_x)) < phase_T(z_x)$,*
   - *$phase_T(parent(z_y)) < phase_T(z_y)$,*
   - *$PhasePath_T(z_x, x)$ $= PhasePath_T(z_y, y)$ $= p_1, \ldots, p_{\ell'}$ (see figure on the right, where similarly shaded regions belong to the same phase)*

   and one of the following holds
   (a) ℓ' is odd and $phase_T(parent(z_y)) < phase_T(parent(z_x))$, or ℓ' is even and $phase_T(parent(z_x)) < phase_T(parent(z_y))$.
   (b) $T_{parent(z_x)} = T_{parent(z_y)}$, and either ℓ' is odd and $parent(z_y) <_{prefix} parent(z_x)$, or ℓ' is even and $parent(z_x) <_{prefix} parent(z_y)$.



It is not hard to see that there is a non-deterministic automaton that guesses the phase-path $p_1, \ldots, p_{\ell'}$ (since this sequence is always ordered in increasing order, we can represent it as the set $\{p_1, \ldots, p_{\ell'}\}$, and hence the number of guesses is $O(2^k)$) and checks whether $x \prec_* y$.

The following lemma states that $\prec_*$ and $\prec'$ indeed coincide (the definition of $\prec'$ and a proof of the lemma are reported in the Appendix).

Lemma 3 (CHARACTERIZATION OF $\prec'$). *Let $T = (V, \lambda)$ be a $(\widetilde{\Sigma} \times [k])$-labeled tree that is well-formed. Then, $x \prec_* y$ if and only if $x \prec' y$ for every $x, y \in V$.*

From the above argument and lemma, and the result shown in [12] we get:

Lemma 4. *For any k, there is a nondeterministic tree automaton of size $exp(O(k))$ that accepts a well-formed tree with two nodes labeled x and y iff $x \prec' y$.*

Thus, we have the following theorem.

Theorem 2. *For any k, there is a nondeterministic tree automaton of size $exp(exp(O(k)))$ which accepts the set of all k-stack trees.*

Tree automata traversing stack trees

Given a k-stack tree T and two nodes x, y of T, we say that y is the *successor* of x if x corresponds to a position j of w and y to position $j+1$ of w, where $wt(w) = T$.

In this section, we show that there is a two-way alternating tree automaton (see [19, 23] for a definition), with $exp(O(k))$ states, that when started at a node x on a k-stack tree T, navigates to the successor of x. However, we assume that we are given *markers* that mark the first letter (marked with s) and last letter (marked with e) of each phase.

```
Procedure Successor($x$)
if EndPhase($x$) then
        return (NextPhase($phase_T(x)$));
elseif ( $y \leftarrow$ PrefixSucc($x$) exists ) then
        return ($y$) ;
else {$z \leftarrow$ ParentRoot($x$);
    $z' \leftarrow$ Predecessor($z$);
    while ( $phase_T(rightChild(z'))$
            $\neq phase_T(x)$ ) do
            $z' \leftarrow$ Predecessor($z'$);
    return ($rightChild(z')$); }
```

```
Procedure Predecessor($x$)
if BeginPhase($x$) then
        return (PrevPhase($phase_T(x)$));
elseif ( $y \leftarrow$ PrefixPred($x$) exists )
        then return ($y$) ;
else {$z \leftarrow$ ParentRoot($x$);
    $z' \leftarrow$ Successor($z$);
    while ( $phase_T(rightChild(z'))$
            $\neq phase_T(x)$ ) do
            $z' \leftarrow$ Successor($z'$);
    return ($rightChild(z')$); }
```

Fig. 1. Successor and predecessor in stack trees.

We can build conjunctively another automaton that checks using $exp(exp(O(k)))$ states that these markers are correct.

Formally, let $T = (V, \lambda)$ be a $(\widetilde{\Sigma} \times [1, k] \times \{s, e, \perp\})$-labeled tree and $T' = (V, \lambda')$ be the $(\widetilde{\Sigma} \times [1, k])$-labeled tree where $\lambda'(x) = (a, i)$ if $\lambda(x) = (a, i, d)$. We say that T is a *k-stack tree with markers*, if T' is a k-stack tree, and all the vertices corresponding to positions of $wt^{-1}(T')$ where a phase starts (resp., ends) are labeled in T with s (resp. e). For two nodes $x, y \in V$, we say that y is the successor of x if y is the successor of x in T'.

Lemma 5. *There exists a two-way alternating tree automaton, with $exp(O(k))$ states that given a k-stack tree T, when started at a node x of T, will navigate precisely to the successor of x (reaching a final state).*

Proof. The 2-way alternating automaton is best described *algorithmically*.It will be easy to see that this algorithm can be executed by a 2-way alternating automaton of the required size. The algorithm is shown in Fig. 1. With EndPhase(x) we denote a predicate that holds true whenever x is the last letter of a phase. With NextPhase(i), $i < k$, we denote the first letter of phase $i + 1$. With PrefixSucc(x), we denote the next letter in the preorder visit of T_x. With ParentRoot(x), we denote the parent of the root of T_x. BeginPhase(x), PrevPhase(i) and PrefixPred(x) are defined analogously.

Intuitively, if x is the last letter of a phase, we navigate to the first letter of the next phase (effected by the first clause). Otherwise, we check whether we can find the successor locally, in the same subtree T_x; this corresponds to finding the next element in the preorder visit of T_x and is delegated to the second clause. If x is the last letter of T_x, then the successor is hard to find. Let z be the parent of the root of T_x and i be the phase number of x. Intuitively, the successor of x is obtained by taking the *last* node *before* z that has a matching return whose phase is i. We hence execute the function Predecessor iteratively till we reach a node that has a right-child of phase i.

Implementing the above requires a 2-way alternating automaton to keep a list of phase numbers. Again, the list can be maintained as a set (since the phase numbers on the list are ordered), and we can engineer the automaton to have $exp(O(k))$ states.

Alternation is used to prove falsity of conditional clauses that are not pursued in the algorithm. □

From the above lemmas and the result from [12], we get:

Theorem 3. *The emptiness problem for k-MVPAs of state-space Q is decidable in time* $exp(|Q| \cdot exp(O(k)))$.

Combining Lemma 2 and the above theorem we get:

Theorem 4. *The membership problem for BPTAs is decidable in* 2ETIME.

4 The Lower Bound

In this section, we show that any language in 2ETIME is accepted by an infinite-state bounded-phase pushdown transducer automata, thereby completing the proof that such automata exactly characterize 2ETIME (Theorem 1).

We start giving a lemma which describes an interesting feature of the bounded-phase multi-stack pushdown rewriting. It states that if we have an unbounded number of pairs of bounded-length words, say bounded by N, then we can check whether every pair (w, w') is such that $|w| = |w'|$ and for each i the i'th symbol of w and w' belong to some relation over symbols, using at most $\lceil \log N \rceil / c$-steps of 2^c-phase multi-stack pushdown rewriting. Consider a finite relation $R \subseteq \Pi \times \Pi$, and two words $w = a_1 \ldots a_m$ and $w' = a'_1 \ldots a'_{m'}$ over Π. We say that (w, w') satisfies R if and only if $m = m'$ and $(a_i, a'_i) \in R$ for $i = 1, \ldots, m$. (The proof of the following lemma is given in the Appendix.)

Lemma 6. *Let Π be a finite alphabet, # be a symbol which is not in Π, $R \subseteq \Pi \times \Pi$, and w be any word of the form $u_1 \# v_1 \# u_2 \# v_2 \# \ldots \# u_m \# v_m$, with $m > 0$ and $u_i, v_i \in \Pi^{2^{cn}}$ for $i = 1, \ldots, m$ with $c, n > 0$.*
There exists a 2^c-phase 2-stack pushdown transducer $\mathcal{T}$ that rewrites within n steps each such word w to a symbol \$ if and only if (u_i, v_i) satisfies R for every $i = 1, \ldots, m$.

Proof sketch. The transducer $\mathcal{T}$ splits each pair (u_i, v_i) into 2^c pairs of words, and writes them onto the output tape. This transducer can be implemented using two stacks and 2^c-phases. (see Appendix for a detailed proof). In n steps, the transducer hence reduces the problem of checking whether every (u_i, v_i) satisfies R to that of checking whether a large number of pairs of letters belongs to R, which can be effected by a regular automaton. □

A transducer, as stated in the above lemma, can be used to check for a Turing machine whether a configuration is a legal successor of another one. We apply this result as a crucial step in proving the following theorem which states the claimed lower bound.

Theorem 5. *For each language L in* 2ETIME(Σ)*, there is a bounded-phase pushdown transducer automaton $\mathcal{A}$ such that $L = \mathcal{L}(\mathcal{A})$.*

Proof sketch. (See the Appendix for more details.) We reduce the membership problem for alternating Turing machines working in $2^{O(n)}$ space to the membership problem for BPTAs. The result then follows from [9].

We briefly sketch a BPTA $\mathcal{A}$ that accepts a words w if and only if w is accepted by a $2^{O(n)}$ space Turing machine $\mathcal{M}$. First $\mathcal{A}$ guesses a word w and a run t of $\mathcal{M}$ encoding them as a sequence of pairs of words (u_i, v_i) such that all the steps taken in t, and w along with the initial configuration, are all represented by at least one such pair. Then, it checks if the guessed sequence indeed encodes an accepting run of $\mathcal{M}$ on w.

In the first task we make use of a slight variation of a standard encoding of trees by words where each pair of consecutive configurations of $\mathcal{M}$ are written consecutively in the word. The second task is by Lemma 6. We observe that it suffices to have single initial and final states for $\mathcal{A}$. □

5 Discussion

We have shown an infinite-automata characterization of the class 2ETIME. This result was obtained independently of the work by Meyer showing that term-automatic infinite automata capture the class ETIME [14]. These two results, along with the characterization of NLINSPACE [15], are currently the only characterizations of complexity classes using infinite automata.

The power of multi-stack rewriting. While infinite automata capture fairly complex languages, there has been little study done on how simple infinite automata can be designed to solve natural algorithmic problems. In this section, we investigate the power of our rewriting. We give infinite automata that solve SAT and QBF (crucially using Lemma 6), and explore connections to infinite automata based on term rewriting. While this of course follows from the lower bound shown in Section 4, the construction is instructive.

We start observing some interesting features of bounded-phase multi-stack pushdown rewriting. We can generate words corresponding to tree encodings, or, in general, belonging to a context free language. (Checking whether a word belongs to a context free language while rewriting can be a problem though: for example, it is not clear how to rewrite in 1-step a word w to a symbol 1 iff $w \in \{a^n b^n \mid n \geq 0\}$.) Also, in each rewriting we can duplicate a bounded number of times any portion of the read word. This can be useful to start many threads of computation on the same string thus speeding-up the total computation. Finally, words can be (evenly) split into a bounded number of sub-words. By iterating such splitting, we can check simple relations between an unbounded number of words, each of exponential length, as shown in Lemma 6.

SAT and QBF. Let us encode Boolean formulas in the standard way, by representing each quantifier, connective, constant and bracket with different symbols, and variables with unbounded length binary strings.

On the first step, $\mathcal{A}$ prepares the computation by rewriting its initial state with a triple (w_1, w_2, w_3) where w_1 is the encoding of a well-formed formula, w_2 is a copy of

w_1 along with a valuation for each variable *occurrence*, and w_3 is the list of variable occurrences coupled with their valuation as annotated in w_2. The word w_1 is guessed nondeterministically using a stack to ensure it is well-formed, and is used by $\mathcal{A}$ to match the input formula. The word w_2 is obtained by copying w_1 and nondeterministically guessing on each variable occurrence a valuation (note that two occurrences of the same variable may be assigned with different values along some runs). Word w_2 is used to evaluate the formula in the guessed valuation. Word w_3 is extracted from w_2 and is later used to generate all pairs $(xb, x'b')$ where x, x' are variable occurrences and b, b' are respectively their assigned values. Such pairs are then checked to see if they define a consistent valuation.

Observe now that evaluating the formula requires a number of steps of rewriting bounded by its height. Also, the pairs of occurrences can be generated in $n-1$ steps of rewriting where n is the number of variable occurrences in the formula: a sequence $x_1 \dots x_n$ is rewritten according to the recurrence $pairs(x_1 \dots x_n)$ is (x_1, x_2) along with $pairs(x_1 x_3 \dots x_n)$ and $pairs(x_2 x_3 \dots x_n)$. Finally, from Lemma 6 checking for pair consistency can be done in the length of the variable representation. Therefore, all tasks are accomplished by the time $\mathcal{A}$ terminates its input and therefore it can correctly accept or reject the input word.

This construction can be generalized to encode QBF. The main difference is that variables are assigned one at each step: when the corresponding quantifier is eliminated. The elimination of universal quantifiers requires duplication of the formula, which can be effected using a work stack.

Term-automatic rewriting. Another way to define infinite automata is to represent states using *terms* (or trees), and use term rewriting to define relations. In [14], term automatic rewriting infinite automata are considered, and it is shown that they precisely capture ETIME (the class of languages accepted by Turing machines in time $2^{O(n)}$).

A binary relation R over terms is *automatic* if it is definable via a tree automaton which reads overlappings of the pair of terms, i.e., the terms are read synchronously on the parts where the corresponding domains intersect (see [14]).

Intuitively, a stack allows us to faithfully represent terms using a well-bracketed word. We now show how to directly translate a term-automatic infinite automaton $\mathcal{A}$ to a multi-stack rewriting infinite automaton $\mathcal{B}$ accepting the same language. Automaton $\mathcal{B}$ on the first step nondeterministically guesses the entire run of $\mathcal{A}$, i.e., a sequence of terms $t_1, \dots, t_N$ where $N-1$ is the length of the word which will be read. Then, it checks if it is indeed an accepting run by generating all the pairs of consecutive terms in the sequence, and then checking them as in Lemma 6. To ensure that terms match when paired, we need to guess terms which all have the same shape (with dummy labels used to mark unused parts of the tree). Also, in order to have all tasks processed on time (i.e., before the input to the automaton is completely read), the guessed terms must be of size at most exponential in N. It is not hard to show by standard techniques that if a term-automatic infinite automaton has an accepting run over a word w, then it has also an accepting run on it which visits terms of size at most exponential in the length of w. Hence the infinite automaton $\mathcal{B}$ accepts the same language as $\mathcal{A}$.

Conclusions and future directions. We have defined (B)PTA with possible infinite initial and final states. Restricting the definition to single initial and final state does not alter the class of recognized languages. In fact, for each (B)PTA $\mathcal{A} = (\Pi, \{\mathcal{T}_d\}_{d \in \Sigma}, Init, Final)$, we can easily construct a language equivalent (B)PTA $\mathcal{A}'$ which has only an initial and a final state.

We observe that, since the construction in Theorem 5 showing 2ETIME hardness uses transducers with only two stacks, the full power of BPTA can be achieved with just two stacks. If we allow transducers with only one stack we can show 2^{2^n} lower bound (we need to use Lemma 10 in the Appendix) but it is left open whether we can capture all 2ETIME (i.e. time $2^{2^{O(n)}}$) using just one-stack transducers.

There are several choices for rewriting that can be studied. For example, prefix rewriting (where essentially the input word is treated as a stack, and an automaton works on it to produce a new stack) precisely defines context-free languages [21]. Regular and synchronized regular rewriting leads to automata that accept context-sensitive languages [15, 18]. Reducing the power of rewriting to one that is weaker than synchronous regular relations seems hard (for e.g., consider relations $R \subseteq \Sigma^* \times \Sigma^*$ where the language $\{w\#w' \mid (w, w') \in R\}$ is regular; this leads to infinite automata that only capture *regular* languages).

We believe that our results may open a new technique to finding rewriting classes that capture complexity classes. Intuitively, a rewriting mechanisms for which checking whether any word in a regular language L can be rewritten in n steps to a word in a regular language L' can be solved in time (or space) $C(n)$ may be a good way to come up with conjectur rewriting schemes that define infinite automata for the class $C(n)$-time (or space).

Along this vein, consider bounded context-switching rewriting where the input word is rewritten to an output word using a finite number of stacks, but where there is only a *bounded* number of switches between the stacks (including the input tape). This is weaker than the rewriting in this paper as the automaton is not allowed to push onto all stacks in one phase. The membership problem for bounded-context-switching automata can be seen to be NP-complete, and it will be interesting to see if this leads us to an infinite automaton characterization of NP.

The most interesting question would be to investigate if any complexity-theoretic result can be proved in a radically different fashion using infinite automata. As mentioned in [21], given that we have infinite automata for the class NL, showing that NL=CO-NL using infinite automata seems an excellent idea to pursue.

References

1. R. Alur and P. Madhusudan. Visibly pushdown languages. In *STOC*, pp. 202–211, 2004.
2. T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN*, vol. 1885 of *LNCS*, pp. 113–130, 2000.
3. A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract regular model checking. In *CAV*, vol. 3114 of *LNCS*, pp. 372–386, 2004.
4. A. Carayol and Stefan Wöhrle. The Caucal hierarchy of infinite graphs in terms of logic and higher-order pushdown automata. In *FSTTCS*, vol. 2914 of *LNCS*, pp. 112–123, 2003.
5. A. Carayol and A. Meyer. Context-sensitive languages, rational graphs and determinism. *Logical Methods in Computer Science*, 2(2), 2006.

6. A. Carayol and A. Meyer. Linearly bounded infinite graphs. *Acta Inf.*, 43(4):265–292, 2006.
7. D. Caucal. On infinite transition graphs having a decidable monadic theory. In *ICALP*, vol. 1099 of *LNCS*, pp. 194–205, 1996.
8. D. Caucal and T. Knapik. A Chomsky-like hierarchy of infinite graphs. In *MFCS*, vol. 2420 of *LNCS*, pp. 177–187, 2002.
9. A. K. Chandra, D. Kozen, and L. J. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, 1981.
10. H.-D. Ebbinghaus and J. Flum. *Finite Model Theory*. Springer, 1995.
11. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.
12. S. La Torre, P. Madhusudan, and G. Parlato. A robust class of context-sensitive languages. In *LICS*, pp. 161–170. IEEE Computer Society, 2007.
13. S. La Torre, P. Madhusudan, and G. Parlato. Context-bounded analysis of queue systems. In *TACAS*, vol. 4963 of *LNCS*, pp. 299–314, 2008.
14. A. Meyer. Traces of term-automatic graphs. In *MFCS*, vol. 4708 of *LNCS*, pp. 489–500, 2007.
15. C. Morvan and C. Stirling. Rational graphs trace context-sensitive languages. In *MFCS*, vol. 2136 of *LNCS*, pp. 548–559, 2001.
16. D. E. Muller and P. E. Schupp. The theory of ends, pushdown automata, and second-order logic. *Theor. Comput. Sci.*, 37:51–75, 1985.
17. E. L. Post. Formal reductions of the general combinatorial decision problem. *American Journal of Mathematics*, 65(2):197–215, 1943.
18. C. Rispal. The synchronized graphs trace the context-sensitive languages. *Electr. Notes Theor. Comput. Sci.*, 68(6), 2002.
19. G. Slutzki. Alternating Tree Automata. *Theor. Comput. Sci.*, 41:305–318, 1985.
20. W. Thomas. Languages, automata, and logic. *Handbook of formal languages*, vol. 3, pp. 389–455, 1997.
21. W. Thomas. A short introduction to infinite automata. In *DLT*, vol. 2295 of *LNCS*, pp. 130–144, 2001.
22. A. Thue. Probleme ber vernderungen von zeichenreihen nach gegebener regeln. *Kra. Vidensk. Selsk. Skrifter. 1. Mat. Nat. Kl. : 10*, 1914.
23. M. Vardi. Reasoning about The Past with Two-Way Automata. In *ICALP*, vol. 1443 of *LNCS*, pp. 628–641, 1998.

A Proof of Lemma 1

The problem of deciding whether w can be rewritten to w' even in two steps by even a one-stack transducer is undecidable:
Lemma 1 *The problem of checking if a word w can be rewritten to a word w' in two steps by a 1-stack pushdown transducer is undecidable.*

Proof. We use a reduction from the *Post's Correspondence Problem* (PCP) which is known to be undecidable [11]. Given two sequences of h words $u_1, \ldots, u_h$ and $v_1, \ldots, v_h$ over Σ, the PCP is to determine whether there exists a sequence of indices $i_1, \ldots, i_m$ such that $u_{i_1} \ldots u_{i_m} = v_{i_1} \ldots v_{i_m}$.

We define a first pushdown transducer $\mathcal{T}_1$ that rewrites the empty word ε to any string of the form $u_{i_1} \ldots u_{i_m} \# v^r_{i_m} \ldots v^r_{i_1}$ where with w^r we denote the reverse of a word w. $\mathcal{T}_1$ simply guesses a sequence of indices and for each guessed index i, writes u_i onto the output tape and v_i onto the stack. Then, it writes $\#$ onto the tape, and pops the stack content onto the output tape. We define a second pushdown transducer $\mathcal{T}_2$ that rewrites a string of the form $w\#w'$ to \$ if and only if $w' = w^r$.

Clearly, the considered PCP has a solution if and only if ε can be rewritten to \$ by first applying $\mathcal{T}_1$ and then $\mathcal{T}_2$. $\square$

B Proof of Lemma 3

In this section we prove that the relation $\prec'$ defined in [12] (which is reported below) and the relation $\prec_*$ defined in Definition 2 are actually the same.

Definition 3 ([12]). *Let $T = (V, \lambda)$ be a $(\widetilde{\Sigma} \times [1, k])$-labeled tree with $phase_T(x) \geq phase_T(parent(x))$, for every $x \in V \setminus \{root\}$. For every $h \in [1, k]$, we inductively define the relations $<_h \subseteq V \times V$ and $<_{[h]} \subseteq V \times V$, as follows:*

- *$x <_h y$ iff $phase_T(x) = phase_T(y) = h$ and either (1) $T_x = T_y$ and $x <_{prefix} y$, or (2) $T_x \neq T_y$, $h > 1$ and $parent(root(T_y)) <_{[h-1]} parent(root(T_x))$.*
- *$x <_{[h]} y$ iff either (1) $phase_T(x), phase_T(y) < h$ and $x <_{[h-1]} y$, or (2) $phase_T(x) = phase_T(y) = h$ and $x <_h y$, or (3) $phase_T(x) < h$ and $phase_T(y) = h$.*

We define the relation $\prec'$ as $<_{[k]}$.

For the sake of readability, we also report Definition 2 here.

Definition 2 *Let $T = (V, \lambda)$ be a well-formed tree. For every $x, y \in V$, $x \prec_* y$ if one of the following holds*

1. $phase_T(x) < phase_T(y)$;
2. $T_x = T_y$ and $x <_{prefix} y$;
3. *There exists an ancestor z_x of x and an ancestor z_y of y such that*
   - $z_x \neq z_y$,
   - $phase_T(parent(z_x)) < phase_T(z_x)$,
   - $phase_T(parent(z_y)) < phase_T(z_y)$,

– $PhasePath_T(z_x, x) =$
  $PhasePath_T(z_y, y) = p_1, \ldots, p_{\ell'}$,

and one of the following holds

(a) ℓ' is odd and $phase_T(parent(z_y)) < phase_T(parent(z_x))$, or ℓ' is even and $phase_T(parent(z_x)) < phase_T(parent(z_y))$.

(b) $T_{parent(z_x)} = T_{parent(z_y)}$, and either ℓ' is odd and $parent(z_y) <_{prefix} parent(z_x)$, or ℓ' is even and $parent(z_x) <_{prefix} parent(z_y)$.

Before proving $\prec' = \prec_*$, we give a technical lemma. It says that when x, y are two nodes of a well-formed tree T having the same phase but belonging to different subtrees, i.e. $T_x \neq T_y$, then condition 3 of Definition 2 holds.

Lemma 7. *Let $T = (V, \lambda)$ be a well-formed $(\widetilde{\Sigma} \times [1, k])$-labeled tree. If $x, y \in V$, $phase_T(x) = phase_T(y)$, $h = phase_T(x)$, $T_x \neq T_y$, and $x <_{[h]} y$ then there exists an ancestor z_x of x and an ancestor z_y of y such that*

– $z_x \neq z_y$,
– $phase_T(parent(z_x)) < phase_T(z_x)$,
– $phase_T(parent(z_y)) < phase_T(z_y)$,
– $PhasePath_T(z_x, x) =$
  $PhasePath_T(z_y, y) = p_1, \ldots, p_{\ell'}$,

and one of the following holds

1. *ℓ' is odd and $phase_T(parent(z_y)) < phase_T(parent(z_x))$, or ℓ' is even and $phase_T(parent(z_x)) < phase_T(parent(z_y))$.*
2. *$T_{parent(z_x)} = T_{parent(z_y)}$, and either ℓ' is odd and $parent(z_y) <_{prefix} parent(z_x)$, or ℓ' is even and $parent(z_x) <_{prefix} parent(z_y)$.*

Proof. The proof is by contradiction. Let C be the set of all pairs (a, b) such that $a <_{[h]} b$ and the lemma does not hold. We consider a pair (x, y) of C that minimizes $depth_T(x) + depth_T(y)$, where $depth_T$ of a node z of T gives the depth of z in T.

From the hypotheses of the lemma and Definition 3, we have that if $x <_{[h]} y$ then $x <_h y$. Since $T_x \neq T_y$, and hence $h > 1$, then $parent(root(T_y)) <_{[h-1]} parent(root(T_x))$. Let $c = parent(root(T_y))$ and $d = parent(root(T_x))$. We distinguish two cases depending on whether T_c and T_d are the same tree or not.

If $T_c = T_d$ then $c <_{prefix} d$. Therefore, if we pick $z_x = root(T_x)$ and $z_y = root(T_y)$, we have that $z_x \neq z_y$ (since $T_x \neq T_y$), $PhasePath_T(z_x, x) = PhasePath_T(z_y, y) = h$ with $\ell' = 1$, and condition *2.* of the lemma holds.

If $T_c \neq T_d$ then we consider two subcases: if $phase_T(c)$ is equal to $phase_T(d)$ or not. Consider first the case in which $phase_T(c) \neq phase_T(d)$. Since $x <_{[k]} y$ we must have that $phase_T(c) < phase_T(d)$. Thus, if we pick $z_x = root(T_x)$ and $z_y = root(T_y)$, we have that $z_x \neq z_y$ (since $T_x \neq T_y$), $PhasePath_T(z_x, x) = PhasePath_T(z_y, y) = h$ with $\ell' = 1$, and condition *1.* of the lemma holds. The remaining case to consider is when also $phase_T(c)$ and $phase_T(d)$ are equal. Since $T_c \neq T_d$ and $depth_T(c) + depth_T(d) < depth_T(x) + depth_T(y)$ we can apply the lemma: there exists an ancestor z_c of c and an ancestor z_d of d such that

– $z_c \neq z_d$,
– $phase_T(parent(z_c)) < phase_T(z_c)$,
– $phase_T(parent(z_d)) < phase_T(z_d)$,
– $PhasePath_T(z_c, c) =$
  $PhasePath_T(z_d, d) = p_1, \ldots, p_{\ell'}$,

and one of the following statements holds

1. ℓ' is odd and $phase_T(parent(z_d)) < phase_T(parent(z_c))$, or ℓ' is even and $phase_T(parent(z_d)) < phase_T(parent(z_c))$.
2. $T_{parent(z_c)} = T_{parent(z_d)}$, and either ℓ' is odd and $parent(z_d) <_{prefix} parent(z_c)$, or ℓ' is even and $parent(z_c) <_{prefix} parent(z_d)$.

Notice that z_d is an ancestor of x and z_c is an ancestor of x. Thus, we set $z_x = z_d$ and $z_y = z_c$. Moreover, $PhasePath_T(z_x, x) = PhasePath_T(z_y, y) = p_1, \ldots, p_{\ell'} h$ whose length is $\ell' + 1$. Now, it is direct to see that one of the conditions between *1.* or *2.* of the lemma holds when we consider x, z_x, y, and z_y. □

Now we first prove that if $x \prec' y$ holds then also $x \prec_* y$ holds.

Lemma 8. *Let $T = (V, \lambda)$ be a well-formed $(\widetilde{\Sigma} \times [1, k])$-labeled tree. For every $x, y \in V$, if $x \prec' y$ then $x \prec_* y$.*

Proof. If $x \prec' y$ then $x \neq y$ and one of the following cases holds:

1. $phase_T(x) < phase_T(y)$;
2. $T_x = T_y$ (thus $phase_T(x) = phase_T(y)$);
3. $phase_T(x) = phase_T(y)$ and $T_x \neq T_y$.

Let $phase_T(y) = h$. It is easy to see that if $x \prec' y$ then $x <_{[h]} y$. The proof continues by proving that $x \prec_* y$ for each case outlined above.

If $phase_T(x) < phase_T(y)$ then by Definition 2 we get immediately that $x \prec_* y$.

If $T_x = T_y$ then, by definition of $<_{[h]}$ (see Definition 3), $x <_h y$ and consequently $x <_{prefix} y$. Therefore, $x \prec_* y$ holds.

Finally, if $phase_T(x) = phase_T(y)$ and $T_x \neq T_y$, and since $x <_{[h]} y$ we can apply Lemma 7 and then by Definition 2 we have that $x \prec_* y$, which concludes the proof. □

Lemma 9. *Let $T = (V, \lambda)$ be a well-formed $(\widetilde{\Sigma} \times [1, k])$-labeled tree. For every $x, y \in V$, if $x \prec_* y$ then $x \prec' y$.*

Proof. From Definition 2, it is easy to see that, given any pair $x, y \in V$ with $x \neq y$, exactly one between $x \prec_* y$ and $y \prec_* x$ holds.

The proof is by contradiction. Assume that there exist x, y such that $x \prec_* y$ and $x \not\prec' y$. Since $\prec'$ is a linear ordering (see [12]), we must have that $y \prec' x$. Now applying Lemma 8 we have that $y \prec_* x$. Therefore, both $y \prec_* x$ and $y \prec_* x$ hold, but this is a contradiction. □

From Lemma 8 and 9, we get the main result of the section:

Lemma 3 (CHARACTERIZATION OF $\prec'$) *Let $T = (V, \lambda)$ be a well-formed $(\widetilde{\Sigma} \times [1, k])$-labeled tree. Then, $x \prec' y$ if and only if $x \prec_* y$ for every $x, y \in V$.*

C Emptiness of k-MVPAs

In this section, we prove that for any class of words accepted by a k-MVPA, the class of trees corresponding to them forms a regular tree language.

Theorem 6. *If L is a k-MVPL, then $wt(L)$ is regular. Moreover, if A is a k-MVPA accepting L, then there is a tree automaton that accepts $wt(L)$ with number of states at most exponential in the size of A and double exponential in the number of phases k (more precisely, $exp(|A|2^{O(k)})$ states).*

Proof. We start giving an MSO sentence φ which guarantees that $wt^{-1}(T)$ is a word of L.

Let $\delta = \{\delta_1, \ldots, \delta_t\}$ be the set of A transitions. We denote with $\bar{s}$ a list of k variables $s_1, \ldots, s_k$ and with $\bar{e}$ a list of k variables $e_1, \ldots, e_k$. Then, φ is of the form
$\exists Y_1 \ldots \exists Y_t \exists \bar{s} \exists \bar{e} \;\; phaseBounds \wedge simulation.$
We use variable Y_i to guess all tree nodes where transition δ_i is taken (along a run). We use variables s_i and e_i to guess the tree nodes corresponding to the beginning and the end of phase i, respectively.

Formula $phaseBounds$ is used to check that the guess on variables is correct and is defined as $\bigwedge_{i=1}^{k} \neg\exists x.(less(x, s_i) \wedge less(e_i, x) \wedge SamePhase(x, s_i))$, where $SamePhase(y, z)$ holds true iff y and x agree on the phase number, and $less(u, v)$ holds true iff $u \prec v$. By Lemma 4, the $\prec$ relation can be captured by a nondeterministic tree automaton with $2^{O(k)}$ states. Therefore, we can construct a $(exp(exp(O(n))$ size tree automaton for checking $phaseBounds$.

Formula $simulation$ is used to check that the guessed run is correct. To accomplish this we need to traverse the tree from the root according to the successor relation (w.r.t. $\prec$), and check whether the states match. By Lemma 5, we can implement this with a 2-way alternating tree automaton with $|Q|\, 2^{O(k)}$ states, where Q is the set of A states, which can be translated to a nondeterministic tree automaton of size $exp(|Q| \cdot exp(O(k))$. The rest of this formula does the usual checks and can be translated to a tree automaton with $exp(exp(O(k))$ states (see [20] for similar proofs).

Since the quantifiers outside $phaseBounds$ and $simulation$ are all existential, the size of the total automaton for φ is $exp(|Q| \cdot exp(O(k)))$. From Theorem 2, we have that there is a nondeterministic tree automaton of size $exp(exp(O(k)))$ which accepts the set of all k-stack trees. Therefore, we can intersect the two automata and get a tree automaton accepting $wt(L)$ of size $exp(|A|2^{O(k)})$, which concludes the proof. $\square$

We can now show the main result of this section, which follows from the above theorem and the fact that tree automata emptiness is solvable in linear time.

Theorem 7. *(*EMPTINESS FOR k-MVPLS*) The emptiness problem for k-MVPLs is decidable in $exp(exp(O(k))$ time.*

D Proofs of Section 4

We start showing a result which differs from Lemma 6 for allowing transducers with 2^c stacks. Then, we argue how this proof can be adapted to show Lemma 6. We end the section proving Theorem 5.

Lemma 10. *Let Π be a finite alphabet, $\#$ be a symbol which is not in Π, $R \subseteq \Pi \times \Pi$, and w be any word of the form $u_1\#v_1\#u_2\#v_2\#\dots\#u_m\#v_m$, with $m > 0$ and $u_i, v_i \in \Pi^{2^{cn}}$ for $i = 1,\dots,m$ with $c, n > 0$.*
There exists a 2^c-phase $(2^c - 1)$-stack pushdown transducer $\mathcal{T}$ that rewrites within n steps each such word w to a symbol $\$$ if and only if (u_i, v_i) satisfies R for every $i = 1,\dots,m$.

Proof. We first prove the lemma for $c = 1$.

The idea is to construct a 2-phase 1-stack transducer $\mathcal{T}$ that evenly splits the sequences between the stack and the output tape. In particular, $\mathcal{T}$ works in two main modes. On inputs of the form stated in the lemma, if words u_i and v_i have length greater than 2, then $\mathcal{T}$ does the two following macro steps: (i) symbols of Γ are rewritten alternatively onto the output tape and the stack, while $\#$ is rewritten both onto the output tape and the stack; (ii) when the reading of the input sequence is completed, $\mathcal{T}$ writes $\#$ on the output tape and moves all the symbols from the stack to the output tape. Otherwise, i.e. if u_i and v_i have length 1, then $\mathcal{T}$ acts as before except for the writing on the output tape: in the first mode it checks the relation R on the first symbols of the pair of words (u_i, v_i), and pushes the second symbols onto the stack; then in the second mode, while popping the symbols from the stack, it checks the relation R on the remaining symbols and then writes $\$$ onto the output tape if all the checked pairs fulfill R.

The effect of each $\mathcal{T}$ rewriting is to split into two halves the words of the pairs encoded in the input. Also, such splitting does not break the link between symbols at the same position in the starting pairs of words (u_i, v_i). By repeatedly applying such rewriting for n steps, we get $\$$ if and only if the pairs (u_i, v_i) of the starting word satisfy R for each $i = 1,\dots,m$.

For $c > 1$, we adapt the above idea, except that we use $2^c - 1$ stacks, and split each word into 2^c sub-words in each rewrite step using 2^c phases. This reduces a word of length 2^{cn} to length 1 in n steps. $\square$

To prove Lemma 6, we define a transducer which rewrites each pair (u, v) by splitting it into 2^c pairs which preserve the symbol position relation between u and v, as argued for showing Lemma 10. However, the way this splitting is achieved is different. While reading the pair from the input, the symbols which are 2^c apart from each other are written to the output, while the others are pushed onto one stack. Then, moving the symbols from one stack to the other, the transducer iteratively extracts (moving them onto the output) the symbols which are in turn $2^c - 1, 2^c - 2, \dots, 1$ apart form each other. This transducer is fairly more complex than that given for showing Lemma 10. In fact, it uses $O(2^c)$ states while the other uses only $O(c)$ states. Thus, we have the following lemma.

Lemma 6 *Let Π be a finite alphabet, $\#$ be a symbol which is not in Π, $R \subseteq \Pi \times \Pi$ and w be any word of the form $u_1\#v_1\#u_2\#v_2\#\dots\#u_m\#v_m$, with $m > 0$ and $u_i, v_i \in \Pi^{2^{cn}}$ for $i = 1,\dots,m$.*
There exists a 2^c-phase 2-stack pushdown transducer $\mathcal{T}$ that rewrites within n steps each such word w to a symbol $\$$ if and only if (u_i, v_i) satisfies R for every $i = 1,\dots,m$.

A transducer, as stated in the above lemmas, can be used to check for a Turing machine whether a configuration is a legal successor of another one. We apply this result as a crucial step in proving the following theorem which states the claimed lower bound.

Theorem 5 *For each language L in $2\text{ETIME}(\Sigma)$, there is a bounded-phase pushdown transducer automaton $\mathcal{A}$ such that $L = \mathcal{L}(\mathcal{A})$.*

Proof. Recall that 2ETIME coincides with the class of all languages that can be accepted by alternating Turing machines working in space $2^{O(n)}$ [9]. Therefore, to show the theorem is suffices to reduce the membership problem for alternating Turing machines working in $2^{O(n)}$ space to the membership problem for BPTAs.

We fix a word w over Σ, and an alternating Turing machine $\mathcal{M}$ which uses $2^{O(n)}$ space on each input of size n. In the following, we briefly describe a BPTA $\mathcal{A}$ that accepts w if and only if $w \in \mathcal{L}(\mathcal{M})$.

The behavior of $\mathcal{A}$ can be summarized into two main tasks: (i) first, $\mathcal{A}$ guesses a word w and a run t of $\mathcal{M}$ encoding them as a sequence of pairs of words (u_i, v_i) such that all the steps taken in t, and w along with the initial configuration, are all represented by at least one such pair; (ii) then, it checks if the guessed sequence indeed encodes an accepting run of $\mathcal{M}$ on w.

We encode each configuration c such that a symbol in it encodes also the left and the right neighbour symbols on the tape along with the transition taken to get to c from its parent configuration in the run. That is, suppose that we take a transition e to enter the configuration $a_1 \ldots (q, a_i) \ldots a_h$ in the considered run, we encode this as $(-, a_1, a_2, e)(a_1, a_2, a_3, e) \ldots (a_{i-1}, (q, a_i), a_{i+1}, e) \ldots (a_{h-1}, a_h, -, e)$. Note that this way checking if a configuration c is the e-successor of c' reduces to checking if (c, c') satisfies an appropriate relation.

To implement the first task we use a slight variation of a standard encoding of binary trees by words. (Recall that a run of an alternating Turing machine is a tree of configurations instead of a sequence of configurations as in standard Turing machines. Also, without loss of generality we can assume such trees to be binary.) For a word c we denote with c^r its reverse. We inductively define the encoding of a tree t, denoted $\langle t \rangle$, as: (1) if t has a single node labeled with a word c, $\langle t \rangle$ is $\# \, c \, \# \, c^r$; (2) if t has root labeled with c and only a subtree t_0, $\langle t \rangle$ is $\# \, c \, \langle t_0 \rangle \, \# \, c^r$; (3) if t has root labeled with c, left subtree t_0 and right subtree t_1, $\langle t \rangle$ is $\# \, c \, \langle t_0 \rangle \, \langle t_1 \rangle \, \# \, c^r$. The run of $\mathcal{M}$ is guessed generating such an encoding in the first step of an $\mathcal{A}$ run on w. To do this, a 2-phase 1-stack pushdown rewriting suffices.

We chose such an encoding since two consecutive configurations c_1 and c_2 of the guessed run appear in the encoding either as ...$\# c_1 \# c_2$ or as ...$\# c_2^r \# c_1^r$, and thus it is possible to extract in one step all the pairs we need to perform the second task. We can implement this task using 2-phase 1-stack pushdown rewriting.

Since we use a constant number of steps to perform the first task (just 2 steps), and we can do the second task using Lemma 10, we have the theorem. □