VLDL Satisfiability and Model Checking via Tree Automata[*][†]

Alexander Weinert

Reactive Systems Group, Saarland University, 66123 Saarbrücken, Germany
weinert@react.uni-saarland.de

Abstract

We present novel algorithms solving the satisfiability problem and the model checking problem for Visibly Linear Dynamic Logic (VLDL) in asymptotically optimal time via a reduction to the emptiness problem for tree automata with Büchi acceptance. Since VLDL allows for the specification of important properties of recursive systems, this reduction enables the efficient analysis of such systems.

Furthermore, as the problem of tree automata emptiness is well-studied, this reduction enables leveraging the mature algorithms and tools for that problem in order to solve the satisfiability problem and the model checking problem for VLDL.

1998 ACM Subject Classification D.2.4 Software/Program Verification

Keywords and phrases Visibly Linear Dynamic Logic, Visibly Pushdown Languages, Satisfiability, Model Checking, Tree Automata

Digital Object Identifier 10.4230/LIPIcs.FSTTCS.2017.47

1 Introduction

Visibly Linear Dynamic Logic (VLDL) [24] is an expressive formalism for specifying properties of recursive systems that allows for an intuitive and modular specification of an important subclass of context-free properties. Although there exist tight bounds on the asymptotic complexity of the satisfiability and the model checking problem for VLDL properties [24], the upper bounds for both problems are witnessed by algorithms that rely on an intricate reduction of the problems to the emptiness problem for visibly pushdown automata [2], for which tool support is lacking.

We present novel reductions of the problems of VLDL satisfiability and VLDL model checking to the emptiness problem for tree automata [20], yielding asymptotically optimal algorithms for both problems. Moreover, as the emptiness problem for tree automata reduces to the problem of solving two-player games with perfect information [15], which is of great importance in the fields of program verification and program synthesis and enjoys mature tool support, the algorithms yielded by our reductions allow us to leverage this tool support for solving the problems of VLDL satisfiability and VLDL model checking.

VLDL is an extension of Linear Temporal Logic (LTL) [16], the de-facto standard for the specification of properties of non-recursive systems. Although popular, it is lacking in expressivity, as it cannot even express all ω-regular properties. The logic VLDL addresses this shortcoming by guarding the temporal operators of LTL with visibly pushdown automata (VPAs) [2]. A VPA is a pushdown automaton that operates over a fixed partition of the

[*] Supported by the project "TriCS" (ZI 1516/1-1) of the German Research Foundation (DFG).
[†] A full version of the paper is available at [23], https://arxiv.org/abs/1708.00699.

input alphabet into calls, returns, and local actions, and has to push (pop) a symbol onto (off) its stack whenever it reads a call (return). Upon processing local actions, the automaton must not touch the stack. Due to this extension, VLDL characterizes exactly the visibly pushdown languages over infinite words [24].

A VLDL formula can be compiled into an equivalent VPA over infinite words of exponential size [24]: First, the VLDL formula is translated into a one-way alternating jumping automaton (1-AJA) [5], an automaton without stack that is able to jump from a call to its matching return. This automaton is then transformed into a VPA of exponential size [5]. Since each visibly pushdown automaton is a classical pushdown automaton, the emptiness problem for VPAs is decidable in polynomial time [2]. The translation from 1-AJAs to VPAs, however, is quite involved, as it handles a far more complex model than is needed for the translation of VLDL formulas into 1-AJAs, thus hampering efforts towards an implementation of the translation from VLDL to VPAs. This effort is further encumbered by the scant availability of emptiness checkers and of model checkers for pushdown systems.

In this work, we introduce novel algorithms solving both the emptiness problem and the model checking problem for VLDL formulas in asymptotically optimal time using a translation of VLDL formulas to nondeterministic tree automata with Büchi acceptance. The technical core of this translation is formed by an encoding of words over visibly pushdown alphabets into trees that is adapted from the encoding of such words given by Alur and Madhusudan [2], as well as by a translation of the 1-AJAs constructed from VLDL formulas into tree automata using an adaptation of the breakpoint-construction by Miyano and Hayashi [14] in order to remove alternation and obtain a nondeterministic automaton. We then check satisfiability of a VLDL formula by checking the resulting tree automaton for emptiness. For model checking a visibly pushdown system against a VLDL specification, we translate the negation of the specification as well as the visibly pushdown system into tree automata, which we subsequently intersect and check for emptiness.

Thus, we reduce both the satisfiability and the model checking problem for VLDL to the emptiness problem for nondeterministic tree automata with Büchi acceptance. Hence, we reduce the complex formalism of VLDL to the simple model of nondeterministic tree automata. Moreover, since the problem of tree automata emptiness reduces to that of solving Büchi games, which is efficiently solvable [7, 17] and enjoys mature tool support [9, 10], our novel reductions enable efficient implementations of satisfiability checkers and of model checkers for VLDL.

Related Work. There exist a number of logics other than VLDL that characterize the class of visibly pushdown languages over infinite words, most prominently Visibly Linear Temporal Logic (VLTL) [6], a fixed-point logic [5], and monadic second order logic augmented with a binary matching predicate (MSO$_\mu$) [2]. We focus here on the logic VLDL, as it most naturally extends the concepts used by LTL [16], the de-facto standard for the specification of non-recursive properties.

Moreover, there exist tools for model checking recursive systems, e.g., Bebop [3, 4] and Moped [18, 19]. These tools are, however, no longer under active development, and have, to the best of our knowledge, not found widespread adoption. In combination with the intricate translation of 1-AJAs into VPAs, this motivates the development of the novel translation of VLDL formulas into tree automata presented in this work.

A number of problems have been reduced to the emptiness problem for tree automata, as such automata are a natural model for capturing the branching-time behavior of systems [15]. Moreover, the theory of tree automata is well-studied, with the most famous result being equivalence of tree automata and monadic second order logic of two successors [21, 25]. Finally, the emptiness problem for tree automata with Büchi acceptance reduces to the problem of solving two-player Büchi games with perfect information [8]. Such games can be solved efficiently [17] and, since Büchi games are a special case of parity games, there exist mature solvers for them [9, 10].

Our Contributions. Firstly, in Section 3 we adapt the tree-encoding of words over visibly pushdown alphabets introduced by Alur and Madhusudan [2] to our setting and show that the resulting trees are recognizable by a tree automaton with Büchi acceptance condition in Theorem 1.

Secondly, in Section 4, we show how to construct tree automata recognizing the encodings of all words satisfying a given VLDL formula in Theorem 2. Moreover, we show that the resulting automaton is of exponential size[1] measured in the size of the original formula and that this translation yields an asymptotically optimal algorithm for satisfiability checking of VLDL formulas.

Finally, in Section 5 we provide a translation of visibly pushdown systems into tree automata recognizing the encodings of all traces of the system. When combining this with the results from Section 4, we obtain an asymptotically optimal algorithm for model checking visibly pushdown systems against VLDL specifications. This result is given in Theorem 6.

2 Preliminaries

In this section we introduce the basic notions used in the remainder of this work, namely (nondeterministic) visibly pushdown automata and related concepts [2].

2.1 Visibly Pushdown Languages

A pushdown alphabet $\widetilde{\Sigma} = (\Sigma_c, \Sigma_r, \Sigma_l)$ is a partition of an alphabet Σ into calls Σ_c, returns Σ_r, and local actions Σ_l. We write w and α for finite and infinite words, respectively, and inductively define the stack height $sh(w)$ reached by any automaton after reading w as $sh(\varepsilon) = 0$, $sh(wc) = sh(w) + 1$ for $c \in \Sigma_c$, $sh(wr) = \max\{0, sh(w) - 1\}$ for $r \in \Sigma_r$, and $sh(wl) = sh(w)$ for $l \in \Sigma_l$. Let $\alpha = \alpha_0\alpha_1\alpha_2 \cdots$ be a finite or infinite word. We say that a call $\alpha_i \in \Sigma_c$ at some position i of α is matched if there exists a position $j > i$ such that $\alpha_j \in \Sigma_r$ and $sh(\alpha_0 \cdots \alpha_{i-1}) = sh(\alpha_0 \cdots \alpha_j)$ and call the return at the smallest such position j the matching return of c. If no such j exists, we call c an unmatched call. If α_i is a matched call with α_j as its matching return, we call the infix $\alpha_{i+1} \cdots \alpha_{j-1}$ of α the nested infix of position i. A word is well-matched if it does not contain a return that is not a matching return. Well-matched words may, however, contain unmatched calls.

A visibly pushdown system (VPS) $\mathcal{S} = (Q, \widetilde{\Sigma}, \Gamma, \Delta, q_I)$ consists of a finite set Q of states, a pushdown alphabet $\widetilde{\Sigma} = (\Sigma_c, \Sigma_r, \Sigma_l)$, a stack alphabet Γ, which contains a stack-bottom marker $\bot$, a transition relation $\Delta \subseteq (Q \times \Sigma_c \times Q \times (\Gamma \setminus \{\bot\})) \cup (Q \times \Sigma_r \times \Gamma \times Q) \cup (Q \times \Sigma_l \times Q)$, and an initial state $q_I \in Q$. A configuration (q, γ) of $\mathcal{S}$ is a pair of a state $q \in Q$ and a stack content $\gamma \in \Gamma_c = (\Gamma \setminus \{\bot\})^* \cdot \bot$. The VPS $\mathcal{S}$ induces the configuration graph $G_{\mathcal{S}} = (Q \times \Gamma_c, E)$ with $E \subseteq ((Q \times \Gamma_c) \times \Sigma \times (Q \times \Gamma_c))$ and $((q, \gamma), a, (q', \gamma')) \in E$ if and only if either (i) $a \in \Sigma_c$, $(q, a, q', A) \in \Delta$, and $A\gamma = \gamma'$, (ii) $a \in \Sigma_r$, $(q, a, \bot, q') \in \Delta$, and $\gamma = \gamma' = \bot$, (iii)

[1] Throughout this work, we say that $f(x)$ is exponential in x if there exist a constant c and a polynomial p such that $f(x) = c^{p(x)}$.

$a \in \Sigma_r$, $(q, a, A, q') \in \Delta$, $A \neq \bot$, and $\gamma = A\gamma'$, or (iv) $a \in \Sigma_l$, $(q, a, q') \in \Delta$, and $\gamma = \gamma'$. A run $\pi = (q_0, \gamma_0) \cdots (q_n, \gamma_n)$ of $\mathcal{S}$ on $w = w_0 \cdots w_{n-1} \in \Sigma^*$ is a sequence of configurations where $q_0 = q_I$, $\gamma_0 = \bot$, and where $((q_i, \gamma_i), w_i, (q_{i+1}, \gamma_{i+1})) \in E$ in $G_{\mathcal{S}}$ for all $i \in [0; n-1]$. Infinite runs of $\mathcal{S}$ on infinite words are defined analogously. We define *traces*$(\mathcal{S})$ as the set of all infinite words α for which there exists a run of $\mathcal{S}$ on α.

2.2 Visibly Linear Dynamic Logic

Let P be a finite set of atomic propositions and let $\widetilde{\Sigma} = (\Sigma_c, \Sigma_r, \Sigma_l)$ be a partition of $\Sigma = 2^P$. The syntax of Visibly Linear Dynamic Logic (VLDL) [24] is defined by the grammar

$$\varphi = p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \langle \mathfrak{A} \rangle \varphi \mid [\mathfrak{A}]\varphi \ ,$$

where $p \in P$ and $\mathfrak{A}$ ranges over testing visibly pushdown automata (TVPA) over the fixed alphabet $\widetilde{\Sigma}$. A TVPA $\mathfrak{A} = (Q, \widetilde{\Sigma}, \Gamma, \Delta, q_I, Q_F, t)$ consists of a VPS $\mathcal{S} = (Q, \widetilde{\Sigma}, \Gamma, \Delta, q_I)$, a set of final states $Q_F \subseteq Q$, and a function t mapping states to VLDL formulas over $\widetilde{\Sigma}$. We define $|\mathfrak{A}| = |Q| + |\Gamma|$ and $|\varphi|$ as the sum of $|\mathrm{cl}(\varphi)|$ and the sum of the sizes of the automata contained in φ, where $\mathrm{cl}(\varphi)$ is the set of all subformulas of φ, including those contained as tests in automata and their subformulas. We require this subformula-relation to be non-circular. A run of $\mathfrak{A}$ on a finite word w is a run of the underlying VPS $\mathcal{S}$ on w. Such a run is accepting if its final state is in Q_F.

Let φ be a VLDL formula, let $\alpha = \alpha_0\alpha_1\alpha_2 \cdots \in \Sigma^\omega$ and let $i \in \mathbb{N}$ be a position in α. We define the semantics of φ in the straightforward way for atomic propositions and for Boolean connectives. Furthermore, we define

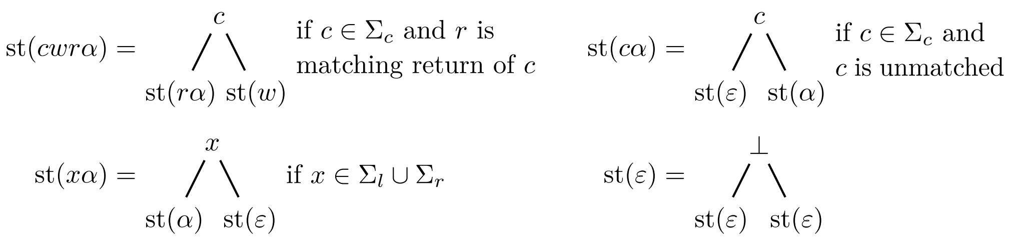
- $(\alpha, i) \models \langle \mathfrak{A} \rangle \varphi$ if there exists $j \geq i$ s.t. $(i, j) \in \mathcal{R}_{\mathfrak{A}}(\alpha)$ and $(\alpha, j) \models \varphi$,
- $(\alpha, i) \models [\mathfrak{A}]\varphi$ if for all $j \geq i$, $(i, j) \in \mathcal{R}_{\mathfrak{A}}(\alpha)$ implies $(\alpha, j) \models \varphi$,

with $\mathcal{R}_{\mathfrak{A}}(\alpha) = \{(i, j) \in \mathbb{N} \times \mathbb{N} \mid \exists$ acc. run $(q_0, \sigma_0) \cdots (q_{j-i}, \sigma_{j-i})$ of $\mathfrak{A}$ on $\alpha_i \cdots \alpha_{j-1}$ and $\forall k \in [0; j-i]. \ (\alpha, i+k) \models t(q_k)\}$. We write $\alpha \models \varphi$ as a shorthand for $(\alpha, 0) \models \varphi$ and say that α is a model of φ in this case. The language of φ is defined as $L(\varphi) = \{\alpha \in \Sigma^\omega \mid \alpha \models \varphi\}$. If $\mathcal{L}(\varphi) \neq \emptyset$, we say that φ is satisfiable.

2.3 Tree Automata

Let $\mathbb{B} = \{0, 1\}$ and let Σ be an alphabet. A Σ-tree t is a mapping $t \colon \mathbb{B}^* \to \Sigma$. We call a finite word $b \in \mathbb{B}^*$ a node and an infinite word $\beta \in \mathbb{B}^\omega$ a branch. Given a node b, we call the nodes $b0$ and $b1$ the left- and right-hand children of b. Analogously, we call the trees rooted at the left- and right-hand children of b the left- and right-hand subtrees of b, respectively. Moreover, b is the parent of both $b0$ and $b1$. The node ε is the root of t. As each node b is associated with the unique path from the root of the tree to b, we say that a node b' is on the path to b if b' is a prefix of b. Similarly, we say that a branch β contains a node b if b is a prefix of β. If $t(b) = a$, we say that b is labeled with a. Moreover, given a tree t and a node b, we define the sub-tree $t\big|_b$ of t rooted at b by $t\big|_b(b') = t(bb')$.

A tree automaton (with Büchi acceptance) $\mathfrak{T} = (Q, \Sigma, \Delta, q_I, Q_F)$ consists of a finite set of states Q, an alphabet Σ, a transition relation $\Delta \subseteq Q \times \Sigma \times Q \times Q$, an initial state $q_I \in Q$, and a set of accepting states $Q_F \subseteq Q$. A run r of $\mathfrak{T}$ on a Σ-tree t is a Q-tree with $r(\varepsilon) = q_I$ and $(r(b), t(b), r(b0), r(b1)) \in \Delta$ for all $b \in \mathbb{B}^*$. A branch of r is accepting if it contains infinitely many nodes b with $r(b) \in Q_F$. A run is accepting if all of its branches are accepting, while an automaton $\mathfrak{T}$ accepts a tree t if there exists an accepting run of $\mathfrak{T}$ on t. The language $L(\mathfrak{T})$ of $\mathfrak{T}$ is defined as the set of all trees accepted by $\mathfrak{T}$. A set of trees is regular if there exists a tree automaton recognizing it. We define $|\mathfrak{T}| = |Q|$. Tree automata are



$\mathrm{st}(cwr\alpha) =$
c
$\mathrm{st}(r\alpha)$ $\mathrm{st}(w)$
if $c \in \Sigma_c$ and r is matching return of c
$\mathrm{st}(c\alpha) =$
c
$\mathrm{st}(\varepsilon)$ $\mathrm{st}(\alpha)$
if $c \in \Sigma_c$ and c is unmatched
$\mathrm{st}(x\alpha) =$
x
$\mathrm{st}(\alpha)$ $\mathrm{st}(\varepsilon)$
if $x \in \Sigma_l \cup \Sigma_r$
$\mathrm{st}(\varepsilon) =$
$\perp$
$\mathrm{st}(\varepsilon)$ $\mathrm{st}(\varepsilon)$

Figure 1 Definition of $\mathrm{st}\colon \Sigma^\omega \cup \Sigma^* \to T_{\Sigma_\perp}$.

closed under intersection via an adaptation of the product-construction for the intersection of automata on words. Hence, for tree automata $\mathfrak{T}_1$,$\mathfrak{T}_2$ there exists a tree automaton $\mathfrak{T}$ with $|\mathfrak{T}| \in \mathcal{O}(|\mathfrak{T}_1||\mathfrak{T}_2|)$ such that $\mathcal{L}(\mathfrak{T}) = \mathcal{L}(\mathfrak{T}_1) \cap \mathcal{L}(\mathfrak{T}_2)$.

3 Stack Trees

Alur and Madhusudan showed how to encode words over some visibly pushdown alphabet as trees by "folding away" the nested infixes of calls into subtrees, thus moving a matched call and its matching return next to each other in the resulting tree [2]. In this section, we slightly adapt their encoding in order to simplify our construction of tree automata later on in Section 4. In that section, we construct for each VLDL formula φ a tree automaton that accepts precisely the encodings of words satisfying φ.

For the remainder of this work, we fix some finite set P of atomic propositions and a pushdown alphabet $\widetilde{\Sigma} = (\Sigma_c, \Sigma_r, \Sigma_l)$ as a partition of 2^P. Let $\alpha \in \Sigma^\omega$ be an infinite word and define $\Sigma_\perp = \Sigma \cup \{\perp\}$, where $\perp$ is some fresh symbol. Intuitively, every node in the resulting $\Sigma_\perp$-tree either denotes one position of α, or it is labeled with the special symbol $\perp$. Formally, we define the function st mapping finite and infinite words over Σ to infinite $\Sigma_\perp$-trees in Figure 1. At every matched call, we encode its matched infix and the suffix starting at and including its matched return in the right- and left-hand subtrees, respectively. At an unmatched call, we encode the infinite suffix of the word starting at the symbol succeeding the call in the right-hand subtree. If the current letter is not a call, we encode the suffix starting at the current letter's successor in the left-hand subtree. All vertices not encoding a symbol of α are labeled with $\perp$.

A $\Sigma_\perp$-tree t is a stack tree if $t = \mathrm{st}(\alpha)$ for some $\alpha \in \Sigma^\omega$. We define the set of all stack trees over Σ as $\mathrm{st}(\Sigma^\omega) = \{\mathrm{st}(\alpha) \mid \alpha \in \Sigma^\omega\}$.

▶ **Theorem 1.** *The set* $\mathrm{st}(\Sigma^\omega)$ *is regular.*

Proof. We first introduce some notation. Let t be a $\Sigma_\perp$-tree. We say that a node b is a matched call if $t(b) \in \Sigma_c$ and $t(b0) \in \Sigma_r$. Similarly, the node $b0$ is a matched return if $t(b) \in \Sigma_c$. If all calls and returns in t are matched, we say that t is well-matched. In contrast to the notion of well-matched words, we demand that a well-matched tree does not contain unmatched calls. Furthermore, we call a branch β finite in t if it eventually only contains $\perp$-labeled vertices. Otherwise, we call β infinite in t. Moreover, we call a tree finite if all of its branches are finite.

We claim that a $\Sigma_\perp$-tree t is a stack tree if and only if $t(\varepsilon) \neq \perp$, if there exists a single branch that is infinite in t, and if the following properties hold true for all $b \in \mathbb{B}^*$:

1. If $t(b) = \perp$, then $t(b0) = t(b1) = \perp$,

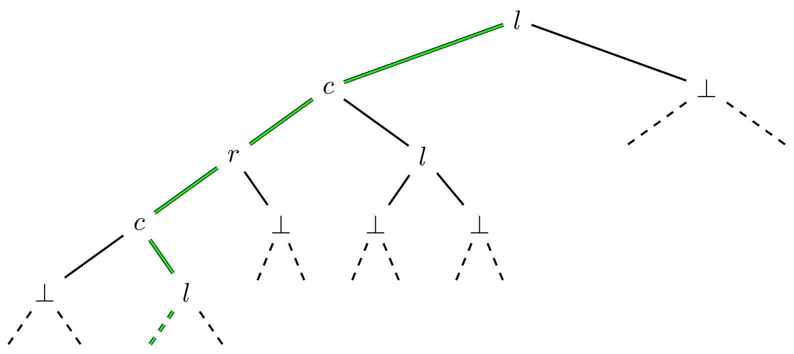


Figure 2 Encoding of $\alpha = lclrcl\cdots$, where the second occurrence of c is unmatched. The cardinal branch of $\mathrm{st}(\alpha)$ is marked in green and doubly lined.

2. if $t(b) \in \Sigma_c$, then
   a. if b is matched, then $t\big|_{b1}$ is finite and well-matched, and
   b. if b is unmatched, then $t(b0) = \perp$ and $t\big|_{b1}$ contains no unmatched returns, and
3. if $t(b) \in \Sigma_l \cup \Sigma_r$, then $t(b1) = \perp$.

Each of these properties can be checked by a tree automaton. As tree automata are closed under intersection, we can construct a single tree automaton that checks all of the above properties. In the full version, we show that the conditions above indeed characterize $\mathrm{st}(\Sigma^\omega)$ [23]. ◀

From the proof of Theorem 1 we obtain that for each $\alpha \in \Sigma^\omega$, there exists a unique branch $\beta = b_0 b_1 b_2 \cdots$ of $\mathrm{st}(\alpha)$ such that $\mathrm{st}(\alpha)(b_0 \cdots b_{i-1}) \neq \perp$ for each $i \in \mathbb{N}$. We call β the cardinal branch of $\mathrm{st}(\alpha)$ and we call the positions of the symbols encoded along β the cardinal positions of α.

We give an example of the tree-encoding of the word $\alpha = lclrcl\cdots$ over the alphabet $\widetilde{\Sigma} = (\Sigma_c, \Sigma_r, \Sigma_l) = (\{c\}, \{r\}, \{l\})$ in Figure 2. The positions $0, 1, 3, 4$ are cardinal positions of α. Moreover, if we assume the second c in α to be unmatched, then the position 5 is a cardinal position as well.

Recall that we defined $sh(w)$ to be the stack height reached by any visibly pushdown automaton after processing $w \in \Sigma^*$. Löding et al. defined the steps of a word $\alpha = \alpha_0\alpha_1\alpha_2\cdots$ as those positions of α that reach a lower bound on the stack height reached during processing the remainder of the word, i.e., $\mathit{steps}(\alpha) = \{i \mid \forall j \geq i.\, sh(\alpha_0\cdots\alpha_i) \leq sh(\alpha_0\cdots\alpha_j)\}$ [13]. This allows for an alternative characterization of cardinal positions: A position i is a cardinal position of α if and only if it is either a step, or if α_i is the matching return of some call occurring at a step.

4 Reducing VLDL Satisfiability to Tree Automata Emptiness

We now reduce the problem of VLDL satisfiability to the emptiness problem for tree automata. The former problem is formulated as follows: "Given some VLDL formula φ, is φ satisfiable?" We formalize the reduction of this problem to the emptiness problem for tree automata in the following theorem:

▶ **Theorem 2.** *For every VLDL formula φ there exists an effectively constructible tree automaton $\mathfrak{T}$ such that $\mathcal{L}(\mathfrak{T}) = \mathrm{st}(\mathcal{L}(\varphi))$ with $|\mathfrak{T}| \in \mathcal{O}(2^{4|\varphi|^3})$.*

Due to this theorem, we obtain an algorithm that checks VLDL formulas for satisfiability by first transforming a given formula φ into the tree automaton $\mathfrak{T}$ recognizing $\mathrm{st}(\mathcal{L}(\varphi))$ and subsequently checking $\mathcal{L}(\mathfrak{T})$ for emptiness. Since tree automata can be checked for emptiness in polynomial time [11, 17], the algorithm runs in exponential time in $|\varphi|$. As the problem of deciding VLDL satisfiability is ExpTime-hard [24], this algorithm is asymptotically optimal.

We split the proof of Theorem 2 into two parts: First, we transform a given VLDL formula into an equivalent one-way alternating jumping automaton (1-AJA) [5] of polynomial size. A 1-AJA is an alternating finite-state automaton on words that is able to "jump" from calls to their matching return, skipping the nested infix. We describe this construction in the proof of Lemma 3. In a second step, we transform the obtained 1-AJA $\mathfrak{A}$ into a tree automaton of exponential size that recognizes the stack trees of words recognized by $\mathfrak{A}$. We describe this construction in the proof of Lemma 5.

Let us first define the above mentioned 1-AJA [5] formally. First, let $\mathit{Dirs} = \{\rightarrow, \curvearrowright\}$, where we use $\rightsquigarrow$ to denote an arbitrary member of Dirs. Moreover, for a finite set Q and $\rightsquigarrow \in \mathit{Dirs}$, let $\mathit{Comms}(Q) = \mathit{Dirs} \times Q \times Q$ and let $\mathcal{B}^+(\mathit{Comms}(Q))$ be the set of positive Boolean formulas over $\mathit{Comms}(Q)$. Note that $\mathcal{B}^+(\mathit{Comms}(Q))$ does not include the shorthands *true* nor *false*. A 1-AJA (with Büchi acceptance) $\mathfrak{A} = (Q, \widetilde{\Sigma}, \delta, q_I, Q_F)$ consists of a finite set of states Q, a visibly pushdown alphabet $\widetilde{\Sigma}$, a transition function $\delta \colon Q \times \Sigma \to \mathcal{B}^+(\mathit{Comms}(Q))$, an initial state $q_I \in Q$, and a set of accepting states $Q_F \subseteq Q$. We define $|\mathfrak{A}| = |Q|$.

Intuitively, when such an automaton $\mathfrak{A}$ is in state q at position i of the word $\alpha = \alpha_0\alpha_1\alpha_2\cdots$, it guesses a set of commands $C \subseteq \mathit{Comms}(Q)$ such that $C \models \delta(q, \alpha_i)$. It then spawns one copy of itself for each command $(\rightsquigarrow, q_\rightarrow, q_\curvearrowright) \in C$ and executes the command with that copy. If $\rightsquigarrow = \curvearrowright$ and if α_i is a matched call, the copy jumps to the position of the matching return of α_i and transitions to state $q_\curvearrowright$. Otherwise, i.e., if $\rightsquigarrow = \rightarrow$, or if α_i is not a matched call, the automaton advances to position $i+1$ and transitions to state $q_\rightarrow$. All copies of $\mathfrak{A}$ proceed in parallel. The automaton $\mathfrak{A}$ accepts a word if all of its copies it visit accepting states infinitely often.

Formally, a run of $\mathfrak{A}$ on an infinite word $\alpha = \alpha_0\alpha_1\alpha_2\cdots$ is an infinite directed acyclic graph $R = (V, E)$ with $V \subseteq \mathbb{N} \times Q$, where $v_I = (0, q_I) \in V$ and all $v \in V$ are reachable from v_I. We call v_I the initial vertex of R and say that a vertex $(i, q) \in V$ is on level i of R. We require that for each $(i, q) \in V$, there exists some $C \subseteq \mathit{Comms}(Q)$ such that $C \models \delta(q, \alpha_i)$ and such that $((i, q), (j, q')) \in E$ if and only if $(j, q') = \mathit{app}(i, c)$ for some $c \in C$. To this end, the command-application function app is defined as $\mathit{app}(i, (\rightsquigarrow, q_\rightarrow, q_\curvearrowright)) = (j, q_\curvearrowright)$ if $\rightsquigarrow = \curvearrowright$ and α_i is a matched call with α_j as its matching return, and $\mathit{app}(i, (\rightsquigarrow, q_\rightarrow, q_\curvearrowright)) = (i+1, q_\rightarrow)$ otherwise. We say that a vertex (i, q) is accepting if q is accepting. Furthermore, a run R is accepting if each vertex in R has at least one successor and if all infinite paths through R starting in v_I contain infinitely many accepting vertices.

In contrast to the classical definition of runs of alternating automata (without jumping capability), an edge in a run of a 1-AJA does not characterize an advance by a single symbol. Instead, there exist "long" edges that characterize the automaton skipping a nested infix. Thus, there may exist positions i such that a run of a 1-AJA on a word does not contain any vertices of the form (i, q), since all copies of the automaton jump over position i. The cardinal positions of a word α, however, serve as synchronization points of a run on α, as no copy of the automaton is able to jump over such positions.

▶ **Lemma 3.** *For every VLDL formula φ there exists an effectively constructible 1-AJA $\mathfrak{A}$ such that $\mathcal{L}(\mathfrak{A}) = \mathcal{L}(\varphi)$ and such that $|\mathfrak{A}| \in \mathcal{O}(|\varphi|^3)$.*

Proof. In earlier work, we constructed a 1-AJA with parity acceptance condition from a given VLDL formula φ by induction over the structure of φ [24]. This more complicated acceptance condition allowed for complementation without a state-space-blowup in the construction of an automaton equivalent to $\varphi = \neg\varphi'$. As we are now aiming for a 1-AJA with a simpler acceptance condition, namely a Büchi condition, we adapt this previous construction.

In order to prevent the costly complementation of 1-AJA (with Büchi acceptance), we require φ to be in negation normal form (NNF), i.e., we assume that negations only occur directly preceding atomic propositions. Should this not be the case, we can easily transform φ into NNF by "pushing down" negations along the syntax tree, using De Morgan's law and the duality $\neg\langle\mathfrak{A}\rangle\varphi \equiv [\mathfrak{A}]\neg\varphi$. Note that this latter duality does not require complementation of the automaton $\mathfrak{A}$, hence it is applicable in constant time. We then construct $\mathfrak{A}_\varphi$ equivalent to φ inductively over the structure of φ.

If $\varphi = p$, $\varphi = \neg p$, or $\varphi = \varphi_1 \circ \varphi_2$ for $\circ \in \{\vee, \wedge\}$, we trivially obtain $\mathfrak{A}_\varphi$, with $|\mathfrak{A}_p| = |\mathfrak{A}_{\neg p}| \in \mathcal{O}(1)$ and $|\mathfrak{A}_{\varphi_1 \circ \varphi_2}| \in \mathcal{O}(|\mathfrak{A}_{\varphi_1}| + |\mathfrak{A}_{\varphi_2}|)$ due to closure of 1-AJA under these operations [5]. If $\varphi = \langle\mathfrak{A}\rangle\varphi'$, we follow the same intuition as in the previous construction, i.e., we construct the 1-AJA $\mathfrak{A}_\varphi$ such that a single copy of $\mathfrak{A}$ jumps along the cardinal positions of the input-word and spawns copies at every matched call in order to verify that the jumps taken correctly summarize finite runs of $\mathfrak{A}$ on the nested infix. Additionally, $\mathfrak{A}_\varphi$ spawns copies verifying that the tests annotating the states along the simulated run hold true. Finally, $\mathfrak{A}_\varphi$ nondeterministically decides to transition into $\mathfrak{A}_{\varphi'}$. The complete construction for this case can be found in the full version of our previous work [24]. Although we constructed 1-AJAs with parity acceptance condition in that work, this previous construction can easily be adapted to use Büchi acceptance by making none of the states simulating $\mathfrak{A}$ accepting in $\mathfrak{A}_\varphi$, thus forcing the simulated run to eventually transition into $\mathfrak{A}_{\varphi'}$.

If $\varphi = [\mathfrak{A}]\varphi'$, we obtain an automaton equivalent to φ via a dual construction to the previous case $\varphi = \langle\mathfrak{A}\rangle\varphi'$. We provide the detailed construction for this case in the full version of this work [23]. ◀

Having translated VLDL formulas into 1-AJAs, we now show how to transform a given 1-AJA $\mathfrak{A}$ into a tree automaton recognizing the stack trees of words from $\mathcal{L}(\mathfrak{A})$. To this end, consider a run R of a 1-AJA $\mathfrak{A}$ on some word $\alpha \in \Sigma^\omega$, as illustrated on the left-hand side of Figure 3. As stated above, the cardinal positions of the processed word serve as synchronization points in the run of $\mathfrak{A}$ on α: If i is a cardinal position of α, then there exist no positions $j, j' \in \mathbb{N}$ with $j < i < j'$ such that R contains an edge from level j to level j'. In other words, each infinite path starting in the initial vertex v_I of R contains a vertex on level i for each cardinal position i of α. Hence, we are able to decide whether or not R is accepting by considering finite paths of R starting and ending in levels i and i', respectively, where i and i' are adjacent cardinal positions of α.

More formally, we demonstrate that the breakpoint construction of Miyano and Hayashi [14] can be adapted to 1-AJAs. To this end, let $R = (V, E)$ be a run of some 1-AJA on some word. A breakpoint sequence over R is an infinite sequence of cardinal positions $0 = i_0 < i_1 < i_2 \cdots$ of α such that all finite paths in R starting on level i_j and ending on level i_{j+1} contain at least one accepting vertex. Each i_j in a breakpoint sequence is called a breakpoint.

▶ **Lemma 4.** *Let R be a run of a 1-AJA. The run R is accepting if and only if there exists a breakpoint sequence over R.*

Proof. First assume that there exists a breakpoint sequence $0 = i_0, i_1, i_2, \ldots$ over R. Then R is clearly accepting, as each infinite path π of R starting in v_I is of the form $\pi =$

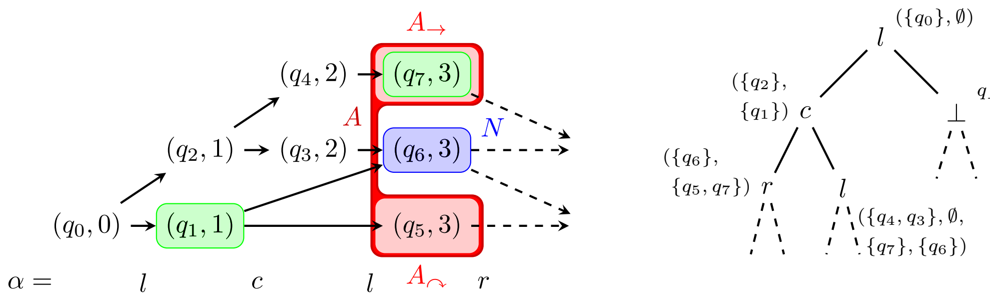


Figure 3 Encoding of a run of a 1-AJA (left) into a run of a tree automaton (right). The states q_1 and q_7 are accepting and marked in green. The positions 0, 1, and 3 are cardinal positions of α. We have $A_\to = \{q_5\}$, $A_\curvearrowright = \{q_7\}$, $A_3 = \{q_5, q_7\}$, and $N_\to = N_\curvearrowright = N = \{q_6\}$. The set A is marked in dark red, while the sets $A_\to$ and $A_\curvearrowright$ are marked in light red. The set N is marked in blue.

$v_0\pi_0v_1\pi_1v_2\pi_2\cdots$, where each $v_j\pi_jv_{j+1}$ is a path from level i_j to level i_{j+1}, hence $v_j\pi_jv_{j+1}$ contains at least one accepting vertex. Thus, π is accepting.

For the other direction, assume that R is accepting. We show the existence of a breakpoint sequence inductively and begin by defining $i_0 = 0$. Now let $i_0, \dots, i_j$ be a finite prefix of a breakpoint sequence and assume towards a contradiction that no cardinal position i_{j+1} exists such that $i_0, \dots, i_j, i_{j+1}$ is a prefix of a breakpoint sequence.

Consider the subgraph of R induced by removing those accepting vertices occurring on levels greater than i_j from R. Since for each cardinal position k, there exists a path from some vertex on level i_j to some vertex on level k that does not contain an accepting vertex, this graph is infinite. Moreover, it is finitely branching, since it is a subgraph of the finitely branching graph R. Due to König's Lemma, the induced subgraph contains an infinite path.

Hence, there also exists an infinite path starting on level i_j that does not contain an accepting vertex, which contradicts R being accepting. Thus, there exists a cardinal position i_{j+1} such that $i_0, \dots, i_j, i_{j+1}$ is a prefix of some breakpoint sequence. Hence, there exists a breakpoint sequence over R. ◀

Given some 1-AJA $\mathfrak{A}$, we now construct a tree automaton that verifies that the input tree is indeed a stack tree and, if this is the case, simulates a run of $\mathfrak{A}$ on the word represented by the input tree by keeping track of the set of states at each level. Moreover, it verifies the existence of a breakpoint sequence, visiting an accepting state on the cardinal branch of the processed tree every time the corresponding symbol is at a cardinal position of the input word that can continue the prefix of the breakpoint sequence constructed so far. In order to do so, we adapt the breakpoint construction by Miyano and Hayashi [14].

The key insight of this construction is that, given some breakpoint i, the vertices of any cardinal position $j > i$ can be partitioned into two sets A and N. The set A contains those states such that each finite path from some vertex on level i to some vertex on level j visits at least one accepting state, while N contains the remaining vertices on level j. We illustrate this partitioning on the left-hand side of Figure 3. If the set N is empty, then the position j continues the breakpoint sequence constructed so far. We adapt this technique in order to translate 1-AJA into tree automata by keeping track of the sets A and N along the cardinal branch of the stack tree. Upon encountering a matched call at position i, the tree automaton guesses the sets A and N reached at the next cardinal position j and verifies this guess when processing the nested infix of position i.

▶ **Lemma 5.** *For every 1-AJA $\mathfrak{A}$ there exists an effectively constructible tree automaton $\mathfrak{T}$ such that $\mathcal{L}(\mathfrak{T}) = \mathrm{st}(\mathcal{L}(\mathfrak{A}))$ with $|\mathfrak{T}| \in \mathcal{O}(2^{4|\mathfrak{A}|})$.*

Proof. We construct a tree automaton $\mathfrak{T}'$ such that $\mathcal{L}(\mathfrak{T}') \cap \mathrm{st}(\Sigma^\omega) = \mathrm{st}(\mathcal{L}(\mathfrak{A}))$. Recall that, due to Theorem 1, we obtain a tree automaton $\mathfrak{T}_\Sigma$ with $\mathcal{L}(\mathfrak{T}_\Sigma) = \mathrm{st}(\Sigma^\omega)$. By intersecting $\mathfrak{T}'$ with $\mathfrak{T}_\Sigma$ we subsequently obtain $\mathfrak{T}$ with the properties stated above.

We have explained the behavior of the automaton $\mathfrak{T}'$ along the cardinal branch above. It remains to take into account the effect of nested infixes on the states reached by $\mathfrak{A}$ at cardinal positions. To this end, we observe that each state reached at a cardinal position is either reached by taking a jumping transition from the previous cardinal position, or by taking a direct transition from the directly preceding position, i.e., from the last position of the nested infix. Thus, when reading a matched call at position i, the automaton $\mathfrak{T}$ guesses sets $A_\rightarrow, N_\rightarrow \subseteq Q$ that are reached eventually by copies of the automaton that process the nested infix w of position i. It moreover guesses sets $A_\curvearrowright, N_\curvearrowright \subseteq Q$ that are reached by copies of the automaton that jump over the nested infix of position i. The automaton then assumes that the states in $A = A_\rightarrow \cup A_\curvearrowright$ and $N = N_\rightarrow \cup N_\curvearrowright$ are indeed reached by processing w and by skipping over w and verifies the former guess while processing $\mathrm{st}(w)$, i.e., the right-hand subtree of the matched call. The latter guess is verified using the transition function of $\mathfrak{A}$. We show an example of this encoding of a run of $\mathfrak{A}$ as a run of $\mathfrak{T}'$ on the right-hand side of Figure 3.

We use two kinds of states in order to implement this idea. States of the form (A, N), where A and N partition a nonempty subset of Q are used along the cardinal branch of the processed stack tree, implementing the breakpoint construction. Furthermore, we use states of the form $((A, N), (A_G, N_G))$, where (A, N) as well as (A_G, N_G) are partitions of nonempty subsets of Q, in order to verify the guesses A_G and N_G about the effects of processing nested infixes. Moreover, we use a sink-state $q_\perp$ in order to process subtrees labeled exclusively with $\perp$.

The detailed construction of the tree automaton $\mathfrak{T}$ recognizing stack trees of words in $\mathcal{L}(\mathfrak{A})$ can be found in the full version [23]. Correctness of $\mathfrak{T}$ follows from Lemma 4 and the above arguments. Moreover, we indeed obtain $|\mathfrak{T}| \in \mathcal{O}(2^{4|\mathfrak{A}|})$. ◀

The proof of Theorem 2 follows from Lemma 3 and Lemma 5: Given a VLDL formula φ, we first construct the 1-AJA $\mathfrak{A}$ with $\mathcal{L}(\mathfrak{A}) = \mathcal{L}(\varphi)$ as demonstrated in the proof of Lemma 3. The automaton $\mathfrak{A}$ is of size $\mathcal{O}(|\varphi|^3)$. We then construct the tree automaton $\mathfrak{T}$ with $\mathcal{L}(\mathfrak{T}) = \mathrm{st}(\mathcal{L}(\mathfrak{A}))$ as shown in the proof of Lemma 5. The automaton $\mathfrak{T}$ recognizes $\mathrm{st}(\mathcal{L}(\mathfrak{A})) = \mathrm{st}(\mathcal{L}(\varphi))$ and is of size $\mathcal{O}(2^{4|\mathfrak{A}|}) = \mathcal{O}(2^{4|\varphi|^3})$.

5 Reducing VLDL Model Checking to Tree Automata Emptiness

In the previous section we have reduced the problem of VLDL satisfiability checking to the emptiness problem for tree automata. We now consider the problem of VLDL model checking, which is formulated as follows: "Given a VPS $\mathcal{S}$ and a VLDL formula φ, does $\mathit{traces}(\mathcal{S}) \subseteq \mathcal{L}(\varphi)$ hold true?" We now reduce this problem to the emptiness problem for tree automata similarly to the reduction of the satisfiability problem for VLDL to the same problem.

▶ **Theorem 6.** *Let $\mathcal{S}$ be a VPS with state space Q and stack alphabet Γ and let φ be a VLDL formula. There exists an effectively constructible tree automaton $\mathfrak{T}$ such that $\mathcal{L}(\mathfrak{T}) = \mathrm{st}(\mathit{traces}(\mathcal{S}) \cap \mathcal{L}(\neg\varphi))$ with $|\mathfrak{T}| \in \mathcal{O}(2^{4|\varphi|^3}|Q|^2|\Gamma|))$.*

Proof. Recall that we can effectively construct a tree automaton $\mathfrak{T}_{\neg\varphi}$ such that $\mathcal{L}(\mathfrak{T}_{\neg\varphi}) = \mathrm{st}(\mathcal{L}(\neg\varphi))$ due to Theorem 2. We now construct a tree automaton $\mathfrak{T}_\mathcal{S}$ recognizing $\mathrm{st}(\mathit{traces}(\mathcal{S}))$.

By intersecting $\mathfrak{T}_{\mathcal{S}}$ and $\mathfrak{T}_{\neg\varphi}$ we subsequently obtain the tree automaton $\mathfrak{T}$ recognizing $\mathrm{st}(\mathit{traces}(\mathcal{S})) \cap \mathrm{st}(\mathcal{L}(\neg\varphi))$. Since, due to injectivity of st, $\mathrm{st}(\mathit{traces}(\mathcal{S})) \cap \mathrm{st}(L(\neg\varphi)) = \mathrm{st}(\mathit{traces}(\mathcal{S}) \cap L(\neg\varphi))$ holds true, we obtain the stated result.

It remains to construct $\mathfrak{T}_{\mathcal{S}}$. Similarly to the proof of Lemma 5, we first construct $\mathfrak{T}'_{\mathcal{S}}$ such that $\mathcal{L}(\mathfrak{T}'_{\mathcal{S}}) \cap \mathrm{st}(\Sigma^\omega) = \mathrm{st}(\mathit{traces}(\mathcal{S}))$. By intersecting $\mathfrak{T}'_{\mathcal{S}}$ with $\mathfrak{T}_\Sigma$ recognizing $\mathrm{st}(\Sigma^\omega)$ as constructed in the proof of Theorem 1 we then obtain the required $\mathfrak{T}_{\mathcal{S}}$. The idea behind the construction of $\mathfrak{T}'_{\mathcal{S}}$ is to simulate a run of $\mathcal{S}$ along the cardinal branch of the tree. This is straightforward in the case of local actions and unmatched calls or returns. Upon encountering a matched call, $\mathfrak{T}'_{\mathcal{S}}$ guesses the state reached by $\mathcal{S}$ upon encountering the matched return and verifies that guess on the stack tree of the nested infix.

Let $\mathcal{S} = (Q, \widetilde{\Sigma}, \Gamma, \Delta, q_I)$. We define $\mathfrak{T}'_{\mathcal{S}} = (Q', \Sigma, \Delta', q_I, Q'_F)$ with $Q' = Q \cup (Q \times Q) \cup (Q \times \Gamma) \cup (Q \times \Gamma \times Q) \cup \{q_\perp\}$, $Q'_F = Q'$, and $\Delta' = \Delta_l \cup \Delta_{uc} \cup \Delta_{ur} \cup \Delta_{mc} \cup \Delta_{mr} \cup \Delta_v \cup \Delta_s$. The individual components of Δ' are defined as follows: We process local actions using transitions of the form $\Delta_l = \{(q, l, q', q_\perp), ((q, q_G), l, (q', q_G), q_\perp) \mid (q, l, q') \in \Delta, q_G \in Q\}$. Similarly, upon encountering unmatched calls or returns, we use transitions of the form $\Delta_{uc} = \{(q, c, q_\perp, q') \mid (q, c, q', A) \in \Delta\}$ and $\Delta_{ur} = \{(q, r, q', q_\perp) \mid (q, r, \perp, q') \in \Delta\}$, respectively. When encountering a matched call, we guess a state q_G reached by the automaton upon processing the matching return and verify that guess using transitions from $\Delta_{mc} = \{(q, c, (q_G, A), (q', q_G)) \mid (q, c, q', A) \in \Delta, q_G \in Q\} \cup \{((q, q_G), c, (q'_G, A, q_G), (q', q'_G)) \mid (q, c, q', A) \in \Delta, q_G, q'_G \in Q\}$. Upon encountering a matched return, we are in some state from $(Q \times \Gamma) \cup (Q \times \Gamma \times Q)$, since a matched return only occurs directly following a matched call. Hence, we use a transition from $\Delta_{mr} = \{((q, A), r, q', q_\perp) \mid (q, r, A, q') \in \Delta\} \cup \{((q, A, q_G), r, (q', q_G), q_\perp) \mid (q, r, A, q') \in \Delta\}$ in order to process that matched return. We verify that we have indeed guessed the correct state by using transitions from $\Delta_v = \{((q, q), \perp, q_\perp, q_\perp) \mid q \in Q\}$. Finally, we define $\Delta_s = \{(q_\perp, \perp, q_\perp, q_\perp)\}$ to continue the run of $\mathfrak{T}'_{\mathcal{S}}$ upon encountering the sink state $q_\perp$.

Using the intuition given above, it can easily be verified that $\mathcal{L}(\mathfrak{T}'_{\mathcal{S}}) \cap \mathrm{st}(\Sigma^\omega) = \mathrm{st}(\mathit{traces}(\mathcal{S}))$ indeed holds true. Thus, we obtain the automaton $\mathfrak{T}_{\mathcal{S},\neg\varphi}$ with the properties given in the statement of this lemma as argued above. ◀

Due to Theorem 6, we obtain a novel asymptotically optimal algorithm for VLDL model checking: Given a VPS $\mathcal{S}$ and a VLDL formula φ, we construct $\mathfrak{T}$ such that $\mathcal{L}(\mathfrak{T}) = \mathrm{st}(\mathit{traces}(\mathcal{S}) \cap \mathcal{L}(\neg\varphi))$. Recall that $\mathit{traces}(\mathcal{S}) \subseteq \mathcal{L}(\varphi)$ if and only if $\mathit{traces}(\mathcal{S}) \cap \mathcal{L}(\neg\varphi) = \emptyset$. Since $\mathrm{st}(\mathit{traces}(\mathcal{S}) \cap \mathcal{L}(\neg\varphi))$ is empty if and only if $\mathit{traces}(\mathcal{S}) \cap \mathcal{L}(\neg\varphi)$ is empty, we obtain that $\mathcal{L}(\mathfrak{T})$ is empty if and only if $\mathit{traces}(\mathcal{S}) \subseteq \mathcal{L}(\varphi)$.

The automaton $\mathfrak{T}$ can be constructed in exponential time and is of exponential size in $|\varphi|$ and of polynomial size in both $|Q|$ and $|\Gamma|$. Hence, we can check $\mathfrak{T}$ for emptiness in exponential time in $|\varphi|$ and in polynomial time in both $|Q|$ and $|\Gamma|$. Since the problem of VLDL model checking is ExpTime-complete [24], this algorithm is asymptotically optimal.

6 Conclusion

In this work we have presented a correspondence between infinite words over a pushdown alphabet and infinite binary trees. Moreover, we demonstrated a construction translating VLDL formulas into tree automata that are language-equivalent with respect to the above correspondence. This construction yields novel algorithms for satisfiability and model checking of VLDL formulas that reduce the problem to the emptiness problem for tree automata. Thus, this construction leverages the strong connection between visibly pushdown languages and regular tree languages that was already exhibited by Alur and Madhusudan in their seminal work on the former family of languages [2]. Moreover, the construction demonstrates that the well-known breakpoint construction by Miyano and Hayashi [14], which is routinely used to remove alternation from stack-free automata, can easily be adapted to transform alternating automata over visibly pushdown words into corresponding alternation-free automata over trees representing such words.

In future work, we plan to empirically evaluate and compare the algorithms presented in this work as well as those presented in earlier work [24], which reduce the satisfiability and model checking problems for VLDL to the emptiness problem for visibly pushdown automata. Recall that our novel algorithm reduces both problems to the emptiness problem for tree automata, which in turn reduces to the problem of solving a two-player Büchi game. The latter problem is well-studied due to its important applications, e.g., in program verification [1, 22] and program synthesis [12]. Hence, there exist efficient algorithms [17] for solving them as well as mature solvers [9, 10]. Thus, we expect our novel algorithm to outperform the existing approach [24] to the above problems.

Moreover, in previous work we investigated the problem of solving two-player games on a visibly pushdown arena in which the winning condition is given by a VLDL formula and determined this problem to be 3ExpTime-complete [24]. We showed membership of this problem in 3ExpTime by reducing it to the problem of solving visibly pushdown games against a winning condition given by visibly pushdown automata. Currently, we are investigating whether the approach presented in this work can be lifted to the setting of games.

Acknowledgments. The author would like to thank Martin Zimmermann for multiple fruitful discussions.

References

1 Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-time temporal logic. *J. ACM*, 49(5):672–713, 2002. doi:10.1145/585265.585270.

2 Rajeev Alur and P. Madhusudan. Visibly pushdown languages. In László Babai, editor, *Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004*, pages 202–211. ACM, 2004. doi:10.1145/1007352.1007390.

3 Thomas Ball and Sriram K. Rajamani. Bebop: A symbolic model checker for boolean programs. In Klaus Havelund, John Penix, and Willem Visser, editors, *SPIN Model Checking and Software Verification, 7th International SPIN Workshop, Stanford, CA, USA, August 30 - September 1, 2000, Proceedings*, volume 1885 of *Lecture Notes in Computer Science*, pages 113–130. Springer, 2000. doi:10.1007/10722468_7.

4 Thomas Ball and Sriram K. Rajamani. Bebop: a path-sensitive interprocedural dataflow engine. In John Field and Gregor Snelting, editors, *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'01, Snowbird, Utah, USA, June 18-19, 2001*, pages 97–103. ACM, 2001. doi:10.1145/379605.379690.

5 Laura Bozzelli. Alternating automata and a temporal fixpoint calculus for visibly pushdown languages. In Luís Caires and Vasco Thudichum Vasconcelos, editors, *CONCUR 2007 - Concurrency Theory, 18th International Conference, CONCUR 2007, Lisbon, Portugal, September 3-8, 2007, Proceedings*, volume 4703 of *Lecture Notes in Computer Science*, pages 476–491. Springer, 2007. doi:10.1007/978-3-540-74407-8_32.

6 Laura Bozzelli and César Sánchez. Visibly Linear Temporal Logic. *J. Aut. Reas.*, 2018. In Press. doi:10.1007/s10817-017-9410-z.

7 Krishnendu Chatterjee, Thomas A. Henzinger, and Nir Piterman. Algorithms for büchi games. *CoRR*, abs/0805.2620, 2008. `arXiv:0805.2620`.

8 Nathanaël Fijalkow, Sophie Pinchinat, and Olivier Serre. Emptiness of alternating tree automata using games with imperfect information. In Anil Seth and Nisheeth K. Vishnoi, editors, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2013, December 12-14, 2013, Guwahati, India*, volume 24 of *LIPIcs*, pages 299–311. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013. `doi:10.4230/LIPIcs.FSTTCS.2013.299`.

9 Oliver Friedmann and Martin Lange. The PGSolver Collection of Parity Game Solvers. *University of Munich*, 2009. Available at `https://github.com/tcsprojects/pgsolver/blob/b88e86e31f2fe02ebcabdccd51ee73f2692ac884/doc/pgsolver.pdf`.

10 Jeroen Keiren. An experimental study of algorithms and optimisations for parity games, with an application to Boolean Equation Systems. Master's thesis, Eindhoven University of Technology, 2009. Available at `http://www.jeroenkeiren.nl/publications`.

11 Orna Kupferman and Moshe Y. Vardi. Weak alternating automata and tree automata emptiness. In Jeffrey Scott Vitter, editor, *Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing, Dallas, Texas, USA, May 23-26, 1998*, pages 224–233. ACM, 1998. `doi:10.1145/276698.276748`.

12 Orna Kupferman and Moshe Y. Vardi. From linear time to branching time. *ACM Trans. Comput. Log.*, 6(2):273–294, 2005. `doi:10.1145/1055686.1055689`.

13 Christof Löding, Parthasarathy Madhusudan, and Olivier Serre. Visibly Pushdown Games. In L. Lodaya and M. Mahajan, editors, *FSTTCS 2004*, volume 3328 of *LNCS*, pages 408–420. Springer, 2005. `doi:10.1007/978-3-540-30538-5\_34`.

14 Satoru Miyano and Takeshi Hayashi. Alternating finite automata on omega-words. *Theor. Comput. Sci.*, 32:321–330, 1984. `doi:10.1016/0304-3975(84)90049-5`.

15 Frank Nießner. Nondeterministic tree automata. In Erich Grädel, Wolfgang Thomas, and Thomas Wilke, editors, *Automata, Logics, and Infinite Games: A Guide to Current Research [outcome of a Dagstuhl seminar, February 2001]*, volume 2500 of *Lecture Notes in Computer Science*, pages 135–152. Springer, 2001. `doi:10.1007/3-540-36387-4_8`.

16 Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57. IEEE Computer Society, 1977. `doi:10.1109/SFCS.1977.32`.

17 Yuval Rabani, editor. *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012*. SIAM, 2012. `doi:10.1137/1.9781611973099`.

18 Stefan Schwoon. *Model checking pushdown systems*. PhD thesis, Technical University Munich, Germany, 2002. URL: `http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2002/schwoon.html`.

19 Dejvuth Suwimonteerabuth, Stefan Schwoon, and Javier Esparza. jmoped: A java byte-code checker based on moped. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3440 of *Lecture Notes in Computer Science*, pages 541–545. Springer, 2005. `doi:10.1007/978-3-540-31980-1_35`.

20 Wolfgang Thomas. Automata on Infinite Objects. *Handbook of theoretical computer science, Volume B*, pages 133–191, 1990.

21 Wolfgang Thomas. Languages, Automata, and Logic. In *Handbook of formal languages*, pages 389–455. Springer, 1997.

22 Moshe Y. Vardi. Automata-theoretic model checking revisited. In Hana Chockler and Alan J. Hu, editors, *Hardware and Software: Verification and Testing, 4th International Haifa Verification Conference, HVC 2008, Haifa, Israel, October 27-30, 2008. Proceedings*, volume 5394 of *Lecture Notes in Computer Science*, page 2. Springer, 2008. `doi:10.1007/978-3-642-01702-5_2`.

23 Alexander Weinert. VLDL satisfiability and model checking via tree automata. *CoRR*, abs/1708.00699, 2017. `arXiv:1708.00699`.

24 Alexander Weinert and Martin Zimmermann. Visibly linear dynamic logic. In Akash Lal, S. Akshay, Saket Saurabh, and Sandeep Sen, editors, *36th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2016, December 13-15, 2016, Chennai, India*, volume 65 of *LIPIcs*, pages 28:1–28:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. `doi:10.4230/LIPIcs.FSTTCS.2016.28`.

25 Wieslaw Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theor. Comput. Sci.*, 200(1-2):135–183, 1998. `doi:10.1016/S0304-3975(98)00009-7`.