Visibly Linear Dynamic Logic[☆]

Alexander Weinert*, Martin Zimmermann

Reactive Systems Group, Saarland University, 66123 Saarbrücken, Germany

Abstract

We introduce Visibly Linear Dynamic Logic (VLDL), which extends Linear Temporal Logic (LTL) by temporal operators that are guarded by visibly pushdown languages over finite words. In VLDL one can, e.g., express that a function resets a variable to its original value after its execution, even in the presence of an unbounded number of intermediate recursive calls. We prove that VLDL describes exactly the ω-visibly pushdown languages, i.e., that it is strictly more expressive than LTL and able to express recursive properties of programs with unbounded call stacks.

The main technical contribution of this work is a translation of VLDL into ω-visibly pushdown automata of exponential size via one-way alternating jumping automata. This translation yields exponential-time algorithms for satisfiability, validity, and model checking. We also show that visibly pushdown games with VLDL winning conditions are solvable in triply-exponential time. We prove all these problems to be complete for their respective complexity classes.

Keywords: Temporal Logic, Visibly Pushdown Languages, Satisfiability, Model Checking, Infinite Games

1. Introduction

Linear Temporal Logic (LTL) [2] is widely used for the specification of non-terminating systems. Its popularity is owed to its simple syntax and intuitive semantics, as well as to the so-called exponential compilation property, i.e., for each LTL formula there exists an equivalent Büchi automaton of exponential size. Due to the latter property, there exist algorithms for model checking in polynomial space and for solving infinite games in doubly-exponential time.

While LTL suffices to express properties of circuits and non-recursive programs with bounded memory, its application to real-life programs is hindered by its inability to express recursive properties. In fact, LTL is too weak to even express all ω-regular properties. There are several approaches to address the latter shortcoming by augmenting LTL, e.g., with regular expressions [3, 4],

[☆]Supported by the projects "TriCS" (ZI 1516/1-1) and "AVACS" (SFB/TR 14) of the German Research Foundation (DFG).
This is an extended and revised version of work first presented at FSTTCS '16. [1]

*Corresponding Author

Email addresses: weinert@react.uni-saarland.de (Alexander Weinert), zimmermann@react.uni-saarland.de (Martin Zimmermann)

finite automata on infinite words [5], and right-linear grammars [6]. We concentrate on the approach of Linear Dynamic Logic (LDL) [4], which guards the globally- and eventually-operators of LTL with regular expressions. While the LTL-formula $\mathbf{F}\psi$ simply means "Either now, or at some point in the future, ψ holds", the corresponding LDL operator $\langle r \rangle \psi$ means "Either now, or at some point in the future, ψ holds and the infix between these two points matches r".

The logic LDL captures the ω-regular languages. In spite of its greater expressive power, LDL still enjoys the exponential compilation property, hence there exist algorithms for model checking and solving infinite games in polynomial space and doubly-exponential time, respectively.

While the expressive power of LDL is sufficient for many specifications, it is still not sufficient to reason about recursive properties of systems. In order to address this shortcoming, we replace the regular expressions guarding the temporal operators with visibly pushdown languages (VPLs) [7] specified by visibly pushdown automata (VPAs) [7].

A VPA is a pushdown automaton that operates over a fixed partition of the input alphabet into calls, returns, and local actions. In contrast to classical pushdown automata, VPAs may only push symbols onto the stack when reading calls and may only pop symbols off the stack when reading returns. Moreover, they may not even inspect the topmost symbol of the stack when not reading returns. Thus, the height of the stack after reading a word is known a priori for all VPAs using the same partition of the input alphabet. Due to this, VPAs are closed under union and intersection, as well as complementation. The class of languages accepted by VPAs is known as visibly pushdown languages.

The class of such languages over infinite words, i.e., ω-visibly pushdown languages, are known to allow for the specification of many important properties in program verification such as "there are infinitely many positions at which at most two functions are active", which may, e.g., express repeated returns to a main-loop, or "every time the program enters a module m while p holds true, p holds true upon exiting m" [8]. The extension of VPAs to their variant operating on infinite words is, however, not well-suited to the specification of such properties in practice, as Boolean operations on such automata do not preserve the logical structure of the original automata. By guarding the temporal operators introduced in LDL with VPAs, VLDL allows for the modular specification of recursive properties while capturing ω-VPAs.

1.1. Our contributions

We begin with an introduction of VLDL and give examples of its use. We then provide translations from VLDL to VPAs over infinite words, so-called ω-VPAs, and vice versa. For the direction from logic to automata we translate VLDL formulas into one-way alternating jumping automata (1-AJA), which are known to be translatable into ω-VPAs of exponential size due to Bozzelli [9]. For the direction from automata to logic we use a translation of ω-VPAs into deterministic parity stair automata by Löding et al. [10], which we then translate into VLDL formulas. Afterwards, we compare and contrast VLDL and Visibly Linear Temporal Logic (VLTL), another logic capturing visibly pushdown languages. The logics VLDL and VLTL share the basic mechanism of guarding temporal operators with languages of finite words.

Secondly, we prove the satisfiability problem and the validity problem for VLDL to be ExpTime-complete. Membership in ExpTime follows from the

previously mentioned constructions, while we show ExpTime-hardness of both problems by a reduction from the word problem for polynomially space-bounded alternating Turing machines adapting a similar reduction by Bouajjani et al. [11].

As a third result, we show that model checking visibly pushdown systems against VLDL specifications is ExpTime-complete as well. Membership in ExpTime follows from ExpTime-membership of the model checking problem for 1-AJAs against visibly pushdown systems. ExpTime-hardness follows from ExpTime-hardness of the validity problem for VLDL.

Moreover, solving visibly pushdown games with VLDL winning conditions is proven to be 3ExpTime-complete. Membership in 3ExpTime follows from the exponential translation of VLDL formulas into ω-VPAs and the membership of solving pushdown games against ω-VPA winning conditions in 2ExpTime due to Löding et al. [10]. 3ExpTime-hardness is due to a reduction from solving pushdown games against LTL specifications, again due to Löding et al. [10].

Finally, we show that replacing the visibly pushdown automata used as guards in VLDL by deterministic pushdown automata yields a logic with an undecidable satisfiability problem.

Our results show that VLDL allows for the concise specification of important properties in a logic with intuitive semantics. In the case of satisfiability and model checking, the complexity jumps from PSpace-completeness for LDL to ExpTime-completeness. For solving infinite games, the complexity gains an exponent moving from 2ExpTime-completeness to 3ExpTime-completeness.

We choose VPAs for the specification of guards in order to simplify arguing about the expressive power of VLDL. In order to simplify the modeling of ω-VPLs, other formalisms that capture VPLs over finite words may be used. We discuss one such formalism in the conclusion.

1.2. Related Work

The need for specification languages able to express recursive properties has been identified before and there exist other approaches to using visibly pushdown languages over infinite words for specifications, most notably VLTL [12] and CaRet [8]. While VLTL captures the class of ω-visibly pushdown languages, CaRet captures only a strict subset of it. For both logics there exist exponential translations into ω-VPAs. In this work, we provide exponential translations from VLDL to ω-VPAs and vice versa. Hence, CaRet is strictly less powerful than VLDL, but every CaRet formula can be translated into an equivalent VLDL formula, albeit with a doubly-exponential blowup. Similarly, every VLTL formula can be translated into an equivalent VLDL formula and vice versa, with doubly-exponential blowup in both directions. For a fragment of VLDL, however, a translation with only exponential blowup exists. This fragment retains the expressiveness of the full logic. We discuss the connections between VLDL and VLTL in more detail in Section 6.

Other logical characterizations of visibly pushdown languages include characterizations by a fixed-point logic [9] and by monadic second order logic augmented with a binary matching predicate (MSO_μ) [7]. Even though these logics also capture the class of visibly pushdown languages, they feature neither an intuitive syntax nor intuitive semantics and thus are less applicable than VLDL in a practical setting.

Moreover, the algorithm for checking satisfiability of VLDL formulas presented in this work relies on a translation of these formulas into a variant of

alternating automata. The emptiness problem for these automata is ultimately solved via a reduction to the emptiness problem for pushdown systems. To the best of our knowledge, no efficient solvers for this problem exist. In order to alleviate this shortcoming of the existing algorithm, an alternative translation of 1-AJA into nondeterministic tree automata that preserves emptiness has been presented [13]. By subsequently reducing the problem of checking tree automata emptiness to that of solving Büchi games, this algorithm achieves asymptotically optimal runtime and reduces the problem of 1-AJA emptiness to one with mature tool support.

2. Preliminaries

In this section we introduce the basic notions used in the remainder of this work. A pushdown alphabet $\widetilde{\Sigma} = (\Sigma_c, \Sigma_r, \Sigma_l)$ is a finite set Σ that is partitioned into calls Σ_c, returns Σ_r, and local actions Σ_l. We write $w = w_0 \cdots w_n$ and $\alpha = \alpha_0\alpha_1\alpha_2 \cdots$ for finite and infinite words, respectively. The stack height $sh(w)$ reached after reading w is defined inductively 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 $l \in \Sigma_l$. We say that a call $c \in \Sigma_c$ at some position k of a word w is matched if there exists a $k' > k$ with $w_{k'} \in \Sigma_r$ and $sh(w_0 \cdots w_k) - 1 = sh(w_0 \cdots w_{k'})$. The return at the earliest such position k' is called the matching return of c. We define $steps(\alpha) := \{k \in \mathbb{N} \mid \forall k' \geq k.\ sh(\alpha_0 \cdots \alpha_{k'}) \geq sh(\alpha_0 \cdots \alpha_k)\}$ as the positions reaching a lower bound on the stack height along the remaining suffix. Note that we have $0 \in steps(\alpha)$ and that $steps(\alpha)$ is infinite for infinite words α.

Visibly Pushdown Systems A visibly pushdown system (VPS) $\mathcal{S} = (Q, \widetilde{\Sigma}, \Gamma, \Delta)$ consists of a finite set Q of states, a pushdown alphabet $\widetilde{\Sigma}$, a stack alphabet Γ, which contains a stack-bottom marker $\bot$, and a transition relation

$$\Delta \subseteq (Q \times \Sigma_c \times Q \times (\Gamma \setminus \{\bot\})) \cup (Q \times \Sigma_r \times \Gamma \times Q) \cup (Q \times \Sigma_l \times Q) \ .$$

The order of the elements of the relation induces a functional view of the transition relation: When processing calls, the automaton outputs a stack symbol to be placed on top of the stack, while when processing a return, it takes the symbol on top of the stack as input and only outputs the state to move to.

A configuration (q, γ) of $\mathcal{S}$ is a pair of a state $q \in Q$ and a stack content $\gamma \in Cont_\Gamma = (\Gamma \setminus \{\bot\})^* \cdot \bot$. The VPS $\mathcal{S}$ induces the configuration graph $G_\mathcal{S} = (Q \times Cont_\Gamma, E)$ with $E \subseteq ((Q \times Cont_\Gamma) \times \Sigma \times (Q \times Cont_\Gamma))$ and $((q, \gamma), a, (q', \gamma')) \in E$ if, and only if, either

1. $a \in \Sigma_c$, $(q, a, q', A) \in \Delta$, and $A\gamma = \gamma'$,
2. $a \in \Sigma_r$, $(q, a, \bot, q') \in \Delta$, and $\gamma = \gamma' = \bot$,
3. $a \in \Sigma_r$, $(q, a, A, q') \in \Delta$, $A \neq \bot$, and $\gamma = A\gamma'$, or
4. $a \in \Sigma_l$, $(q, a, q') \in \Delta$, and $\gamma = \gamma'$.

For an edge $e = ((q, \gamma), a, (q', \gamma'))$, we call a the label of e. A (finite or infinite) run $\pi = (q_0, \gamma_0)(q_1, \gamma_1)(q_2, \gamma_2) \cdots$ of $\mathcal{S}$ on $w = w_0 w_1 w_2 \cdots$ is a sequence of configurations where $((q_i, \gamma_i), w_i, (q_{i+1}, \gamma_{i+1})) \in E$ in $G_\mathcal{S}$ for all $i \in [0; |\pi|)$ or for all $i \in \mathbb{N}$ in the case of infinite runs. The VPS $\mathcal{S}$ is deterministic if for each vertex (q, γ) in $G_\mathcal{S}$ and each $a \in \Sigma$ there exists at most one outgoing a-labeled edge from (q, γ). In figures, we write $\downarrow A$, $\uparrow A$ and $\rightarrow$ to denote pushing A onto the stack, popping A off the stack, and local actions, respectively.

(Büchi) Visibly Pushdown Automata A visibly pushdown automaton (VPA) [7] is a six-tuple $\mathfrak{A} = (Q, \widetilde{\Sigma}, \Gamma, \Delta, I, F)$, where $\mathcal{S} = (Q, \widetilde{\Sigma}, \Gamma, \Delta)$ is a VPS and $I, F \subseteq Q$ are sets of initial and final states. A run $(q_0, \gamma_0)(q_1, \gamma_1)(q_2, \gamma_2) \cdots$ of $\mathfrak{A}$ is a run of $\mathcal{S}$, which we call initial if $(q_0, \gamma_0) = (q_I, \bot)$ for some $q_I \in I$. A finite run $\pi = (q_0, \gamma_0) \cdots (q_n, \gamma_n)$ is accepting if $q_n \in F$. A VPA $\mathfrak{A}$ accepts a finite word w if there exists an initial accepting run of $\mathfrak{A}$ on w. We denote the family of languages accepted by VPA by VPL.

A Büchi VPA (BVPA) is syntactically identical to a VPA, but we only consider runs over infinite words. An infinite run is Büchi-accepting if it visits states in F infinitely often. A BVPA $\mathfrak{A}$ accepts an infinite word α if there exists an initial Büchi-accepting run of $\mathfrak{A}$ on α. We denote the family of languages accepted by BVPA by ω-VPL.

Finally, we define the size of a VPA or a BVPA $\mathfrak{A}$ as $|\mathfrak{A}| = |Q| + |\Gamma|$.

3. Visibly Linear Dynamic Logic

We fix a finite set P of atomic propositions and a partition $\widetilde{\Sigma} = (\Sigma_c, \Sigma_r, \Sigma_l)$ of 2^P throughout this work. The syntax of VLDL is defined by the grammar

$$\varphi := p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \langle \mathfrak{A} \rangle \varphi \mid [\mathfrak{A}]\varphi \ ,^{1}$$

where $p \in P$ and where $\mathfrak{A}$ ranges over testing visibly pushdown automata (TVPA) over $\widetilde{\Sigma}$. We define a TVPA $\mathfrak{A} = (Q, \widetilde{\Sigma}, \Gamma, \Delta, I, F, t)$ as consisting of a VPA $(Q, \widetilde{\Sigma}, \Gamma, \Delta, I, F)$ and a partial function t mapping states to VLDL formulas over $\widetilde{\Sigma}$.[2] Intuitively, such an automaton accepts an infix $\alpha_i \cdots \alpha_j$ of an infinite word $\alpha_0\alpha_1\alpha_2 \cdots$ if the embedded VPA has an initial accepting run $(q_i, \gamma_i) \cdots (q_{j+1}, \gamma_{j+1})$ on $\alpha_i \cdots \alpha_j$ such that, if q_{i+k} is marked with φ by t, then $\alpha_{i+k}\alpha_{i+k+1}\alpha_{i+k+2} \cdots$ satisfies φ.

We define the size of φ as the sum of the number of subformulas (including those contained as tests in automata and their subformulas) and of the sizes of the automata contained in φ. As shorthands, we use $\mathtt{tt} := p \vee \neg p$ and $\mathtt{ff} := p \wedge \neg p$ for some atomic proposition p. Even though the testing function t is defined as a partial function, we generally assume it is total by setting $t \colon q \mapsto \mathtt{tt}$ if q is not in the domain of t. If t labels each state of $\mathfrak{A}$ with $\mathtt{tt}$, we say that $\mathfrak{A}$ is test-free.

Let $\alpha = \alpha_0\alpha_1\alpha_2 \cdots$ be an infinite word over 2^P and let $k \in \mathbb{N}$ be a position in α. We define the semantics of VLDL inductively via

- $(\alpha, k) \models p$ if, and only if, $p \in \alpha_k$,
- $(\alpha, k) \models \neg\varphi$ if, and only if, $(\alpha, k) \not\models \varphi$,
- $(\alpha, k) \models \varphi_0 \wedge \varphi_1$ if, and only if, $(\alpha, k) \models \varphi_0$ and $(\alpha, k) \models \varphi_1$, and dually for $\varphi_0 \vee \varphi_1$,
- $(\alpha, k) \models \langle \mathfrak{A} \rangle \varphi$ if, and only if, there exists $l \geq k$ s.t. $(k, l) \in \mathcal{R}_{\mathfrak{A}}(\alpha)$ and $(\alpha, l) \models \varphi$,

[1] The logic VLDL does not feature the standard temporal operators **X** and **U** from LTL. We show that these operators can be added without changing the expressiveness or complexity of the logic in Section 6.

[2] Obviously, there are some restrictions on the nesting of tests into automata. More formally, we require the subformula relation to be acyclic as usual.

- $(\alpha, k) \models [\mathfrak{A}]\varphi$ if, and only if, for all $l \geq k$, $(k, l) \in \mathcal{R}_{\mathfrak{A}}(\alpha)$ implies $(\alpha, l) \models \varphi$,

where $\mathcal{R}_{\mathfrak{A}}(\alpha)$ contains all pairs of positions (k, l) such that $\mathfrak{A}$ accepts $\alpha_k \cdots \alpha_{l-1}$. Formally, we define

$$\begin{aligned}\mathcal{R}_{\mathfrak{A}}(\alpha) := \{(k,l) \in \mathbb{N} \times \mathbb{N} \mid \\ \text{there exists an initial accepting run } (q_k, \gamma_k) \cdots (q_l, \gamma_l) \text{ of } \mathfrak{A} \text{ on } \alpha_k \cdots \alpha_{l-1} \\ \text{and } \forall m \in \{k, \ldots, l\}.\ (\alpha, m) \models t(q_m)\}.\end{aligned}$$

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 $L(\varphi) := \{\alpha \in (2^P)^\omega \mid \alpha \models \varphi\}$. As usual, disjunction and conjunction are dual, as well as the $\langle\mathfrak{A}\rangle$-operator and the $[\mathfrak{A}]$-operator, which can be dualized using De Morgan's law and the logical identity $[\mathfrak{A}]\varphi \equiv \neg\langle\mathfrak{A}\rangle\neg\varphi$, respectively. Note that the latter identity only dualizes the temporal operator, but does not require complementation of the automaton guarding the operator. We additionally allow the use of derived boolean operators such as $\rightarrow$ and $\leftrightarrow$, as they can easily be reduced to the basic operators $\wedge$, $\vee$ and $\neg$.

The logic VLDL combines the expressive power of visibly pushdown automata with the intuitive temporal operators of LDL. Thus, it allows for concise and intuitive specifications of many important properties in program verification [7]. In particular, VLDL allows for the specification of recursive properties, which makes it more expressive than both LDL [4] and LTL [2]. In fact, we can embed LDL in VLDL in linear time.

Lemma 1. *For any LDL formula ψ over P we can effectively construct a VLDL formula φ over $\widetilde{\Sigma} := (\emptyset, \emptyset, 2^P)$ in linear time such that $L(\psi) = L(\varphi)$.*

Proof. We define φ by structural induction over ψ. Recall that the syntax and semantics of LDL is very similar to that of VLDL, but uses regular expressions with tests as guards instead of visibly pushdown automata. The only interesting case of the induction is $\psi = \langle r \rangle \psi'$, where r is a regular expression containing tests, since all other cases follow from closure properties and duality. We obtain the VLDL formula φ' over $\widetilde{\Sigma}$ equivalent to ψ' by induction and construct the finite automaton $\mathfrak{A}_r$ from r using the construction of Faymonville and Zimmermann [14]. This construction yields an automaton equivalent to r by adapting the well-known Thompson construction [15] in order to account for the tests occurring in r. The automaton $\mathfrak{A}_r$ contains tests, but is not equipped with a stack. Since $\widetilde{\Sigma} = (\emptyset, \emptyset, 2^P)$ only contains local actions, we can interpret $\mathfrak{A}_r$ as a TVPA without changing the language it recognizes. We call the TVPA $\mathfrak{A}'_r$ and define $\varphi = \langle \mathfrak{A}'_r \rangle \varphi'$. □

Since LTL can be in turn embedded in LDL in linear time, Lemma 1 directly implies the embeddability of LTL in VLDL in linear time. Note that this proof motivates the use of TVPAs instead of VPAs without tests as guards in order to obtain a concise formalism. We later show that removing tests from these automata does not change the expressiveness of VLDL. It is, however, open whether it is possible to translate even LTL formulas into VLDL formulas without tests in polynomial time.

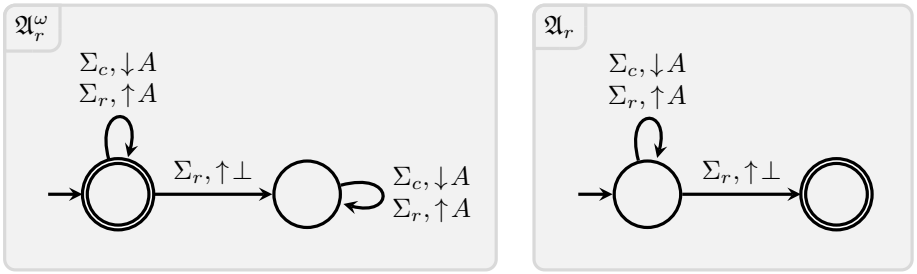

Figure 1: The automata $\mathfrak{A}_r^\omega$ and $\mathfrak{A}_r$ used in Example 2. We draw the accepting states with a double circle.

4. Examples of VLDL Specifications

As we will show in Section 5, VLDL captures the visibly pushdown languages and thus, it is strictly more expressive than traditional Büchi automata. In fact, VLDL allows for concise formulations of a number of important properties of recursive programs in program verification. We give some examples of such properties and their formalization in this section.

Example 2. Assume that we have a program that may call some module and that has the observable atomic propositions $P := \{c, r\}$, where c and r denote function calls and returns, respectively. Our aim is to specify a very basic assumption on the consistency of the trace: "During the run of a program, no unmatched returns may occur." Although each actual trace of a program satisfies this property, it is instructive to specify it formally.

For the sake of readability, we assume that the program emits either $\{c\}$ or $\{r\}$ in each step. Since we want to keep track of the calls and returns occurring in the program using the stack, we choose the pushdown alphabet $\widetilde{\Sigma} = (\Sigma_c, \Sigma_r, \Sigma_l)$ with $\Sigma_c = \{c\}$, $\Sigma_r = \{r\}$, and $\Sigma_l = \emptyset$.

The ω-VPA $\mathfrak{A}_r^\omega$ shown in Figure 1 recognizes the traces satisfying this requirement. When specifying the property as a VLDL formula, we only require minute modifications to $\mathfrak{A}_r^\omega$ in order to transform it into an automaton that accepts any prefix of a word that ends in an unmatched return. We show this automaton $\mathfrak{A}_r$ in Figure 1. Using this automaton, we can specify the property above with the VLDL formula $\varphi = \neg\langle\mathfrak{A}_r\rangle\mathtt{tt}$.

Example 3. Now assume that we have a program that may call some module and that has the observable atomic propositions $P := \{c, r, p, q\}$, where c and r denote calls to and returns from the module, and p and q are arbitrary propositions.

We now construct a VLDL formula that describes the condition "If p holds true immediately after entering the module, it shall hold immediately after the corresponding return from the module as well" [8]. Under the assumption that the module is able to call itself recursively this property is not ω-regular. For the sake of readability, we make similar assumptions on the program as in the previous example, i.e., we assume that the program never emits both c and r in the same step and that it emits at least one atomic proposition in each step. Since we want to count the calls and returns occurring in the program using

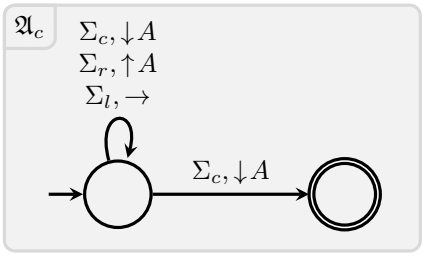

Figure 2: The automaton $\mathfrak{A}_c$ used in Example 3.

the stack, we pick the pushdown alphabet $\widetilde{\Sigma} = (\Sigma_c, \Sigma_r, \Sigma_l)$ such that $P' \subseteq P$ is in Σ_c if $c \in P'$, P' is in Σ_r if $r \in P'$, but $c \notin P'$, and P' is in Σ_l otherwise.

The formula $\varphi := [\mathfrak{A}_c](p \to \langle \mathfrak{A}_r \rangle p)$ captures the condition above, where Figure 2 shows $\mathfrak{A}_c$, whereas $\mathfrak{A}_r$ is the automaton from Example 2 shown in Figure 1. The automaton $\mathfrak{A}_c$ accepts all finite words ending with a call to the module, whereas the automaton $\mathfrak{A}_r$ accepts all words ending with a single unmatched return.

Figure 3 shows a BVPA describing the same specification as φ. For the sake of readability, we use $\Sigma_x^p = \{P' \in \Sigma_x \mid p \in P'\}$ and $\Sigma_x^{\neg p} = \{P' \in \Sigma_x \mid p \notin P'\}$ for $x \in \{c, r, l\}$. In contrast to φ, which uses only a single stack symbol, namely A, the BVPA has to rely on the two stack symbols P and $\bar{P}$ to track whether or not p held true after entering the module m. Moreover, there is no direct correlation between the logical structure of the specification and the structure of the BVPA, which exemplifies the difficulty of maintaining specifications given as BVPAs.

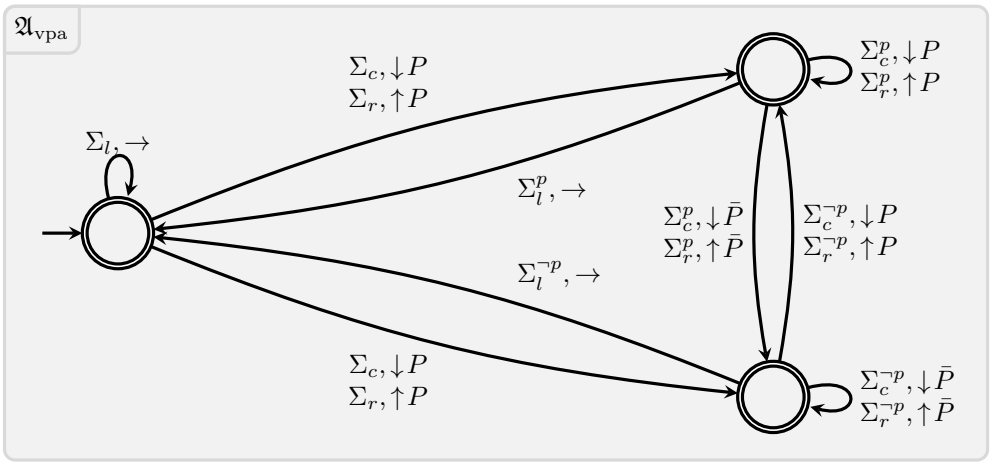

Figure 3: The BVPA $\mathfrak{A}_{\text{vpa}}$ specifying the same language as φ from Example 3.

Finally, one could also specify the property above using the formalism of VLTL [12], which relies on augmented regular expressions instead of automata. Since VLTL supports the guarded existential modality as well, it suffices to translate the VPAs $\mathfrak{A}_c$ and $\mathfrak{A}_r$ into VREs. Using the VREs $r_c = \Sigma^* r$ and $r_r = ((cl^* r)^* l^*)^{\circlearrowleft_l} r$, which specify the languages recognized by $\mathfrak{A}_c$ and $\mathfrak{A}_r$, respec-

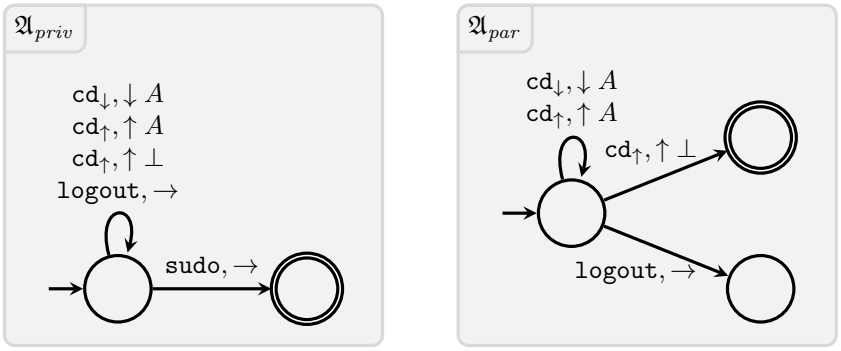

Figure 4: The automata $\mathfrak{A}_{priv}$ and $\mathfrak{A}_{par}$ used in Example 4.

tively, as well as the duality $\langle\mathfrak{A}\rangle\varphi \equiv \neg[\mathfrak{A}]\neg\varphi$ we obtain the VLTL formula

$$\psi = \neg(r_c; \neg(p \rightarrow (r_r; p)))$$

which specifies the property given above. We compare the two logics VLDL and VLTL more in-depth in Section 6.

In contrast to these two alternative formal specifications, VLDL offers a well readable and intuitive formalism that combines the well-known standard acceptors for visibly pushdown languages with guarded versions of the widely used temporal operators of LTL and the readability of classical logical operators.

Note that the stack is simply used as a counter in Example 3. This technique suffices for the specification of other properties as well, such as tracking the path through a directory structure instead of a call stack.

Example 4. We consider a simplified system model, in which a user can move through directories and obtain and relinquish superuser rights. To this end, we consider the set of atomic propositions $P = \{\texttt{cd}_{\downarrow}, \texttt{cd}_{\uparrow}, \texttt{sudo}, \texttt{logout}\}$, where $\texttt{cd}_{\downarrow}$ denotes moving into a subdirectory of the current working directory, $\texttt{cd}_{\uparrow}$ denotes moving to the parent directory, `sudo` denotes the acquisition of elevated privileges, and `logout` denotes relinquishing them. For readability, we only define the pushdown alphabet for singleton subsets of P and pick $\widetilde{\Sigma} := (\{\texttt{cd}_{\downarrow}\}, \{\texttt{cd}_{\uparrow}\}, \{\texttt{sudo}, \texttt{logout}\})$ in order to formalize the property "If the program acquires elevated privileges, it has to relinquish them before moving out of its current directory" [16].

We use the stack as a counter using the stack alphabet $\Gamma := \{\bot, A\}$. Then the formula $\varphi := [\mathfrak{A}_{priv}]\neg\langle\mathfrak{A}_{par}\rangle\texttt{tt}$, specifies the property above, where $\mathfrak{A}_{priv}$ accepts all prefixes of runs of the program that end with the acquisition of elevated privileges, and $\mathfrak{A}_{par}$ tracks the depth of the current working directory. We depict the automata $\mathfrak{A}_{priv}$ and $\mathfrak{A}_{par}$ in Figure 4.

While the previous example shows how to handle programs that can simply request a single set of elevated rights, in actual systems the situation is more complicated. In reality, a program may request the rights of any user of the system by logging in as that user. When logging out, the rights revert to those of the previously logged in user. In the following example we use the stack to keep track of the currently logged in user and ensure that system calls are not executed with elevated privileges.

Example 5. We remove some of the simplifications of the previous example and model the login mechanism of an actual system more precisely. To this end, let $P = \{\texttt{exec}, \texttt{login}_s, \texttt{login}_u, \texttt{logout}\}$, where `exec` denotes the execution of a system call, $\texttt{login}_s$ and $\texttt{login}_u$ denote the login as the superuser and some other user, respectively, and `logout` denotes logging the current user out and reverting to the previous user. The pushdown alphabet $\widetilde{\Sigma} := (\{\texttt{login}_s, \texttt{login}_u\}, \{\texttt{logout}\}, \{\texttt{exec}\})$ allows us to keep track of the stack of logged in users. We want to specify the property "No system calls shall be executed while the user has obtained elevated privileges."

Recall that visibly pushdown automata are not allowed to inspect the top of the stack. Thus, in order to correctly trace the currently logged in user, we need to store both the current user and the previously logged in user on the stack. The automaton $\mathfrak{A}_{user}$ performs this bookkeeping using the stack alphabet $\Gamma := \{(c, p) \mid c, p \in \{s, u\}\}$, where c denotes the currently logged in user, and p denotes the previously logged in user. It moves to the state u when a normal user is logged in and to the state s when a superuser is logged in.

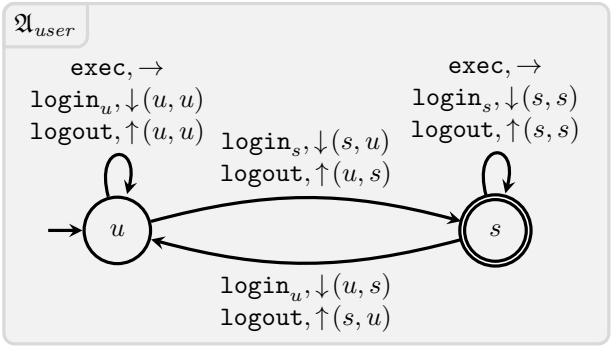

Figure 5: The automaton $\mathfrak{A}_{user}$, which keeps track of the status of the currently logged in user.

Since the only action available to the program in this example apart from logging users in or out is to execute system calls, we do not need an additional automaton to capture the undesired behavior, but can simply use the atomic proposition `exec` in the formula. Hence, the formula $\varphi := [\mathfrak{A}_{user}]\neg\texttt{exec}$ defines the desired behavior.

Due to the modular nature of VLDL, we can easily reuse existing automata and subformulas. Consider, e.g., a setting similar to that of Examples 4 and 5 with the added constraint that we want to make sure that superusers neither execute system calls, nor leave the directory they were in when they acquired superuser-privileges. Using some simple modifications to $\mathfrak{A}_{user}$ and $\mathfrak{A}_{par}$ to work over an extended set of atomic propositions, we can specify the conjunction of the previously defined behaviors without having to construct new automata from scratch.

5. VLDL Captures ω-VPL

In this section we show that VLDL captures ω-VPL. Recall that a language is in ω-VPL if, and only if, there exists a BVPA recognizing it. We provide

effective constructions transforming BVPAs into equivalent VLDL formulas and vice versa.

Theorem 6. *For any language of infinite words $L \subseteq \Sigma^\omega$ there exists a BVPA $\mathfrak{A}$ with $L(\mathfrak{A}) = L$ if, and only if, there exists a VLDL formula φ with $L(\varphi) = L$. There exist effective translations for both directions.*

In Section 5.1 we show the construction of VLDL formulas from BVPAs via deterministic parity stair automata. In Section 5.2 we construct one-way alternating jumping automata from VLDL formulas. These automata are known to be translatable into equivalent BVPAs. Both constructions incur an exponential blowup in size. We show this blowup to be unavoidable in the construction of BVPAs from VLDL formulas. It remains open whether the blowup can be avoided in the construction for the other direction.

5.1. From Stair Automata to VLDL

In this section we construct a VLDL formula of exponential size that is equivalent to a given BVPA $\mathfrak{A}$. To this end, we first transform $\mathfrak{A}$ into an equivalent deterministic parity stair automaton (DPSA) [10] in order to simplify the translation. A parity stair automaton (PSA) $\mathfrak{A} = (Q, \widetilde{\Sigma}, \Gamma, \Delta, I, \Omega)$ consists of a VPS $\mathcal{S} = (Q, \widetilde{\Sigma}, \Gamma, \Delta)$, a set of initial states I, and a coloring $\Omega\colon Q \to \mathbb{N}$. The automaton $\mathfrak{A}$ is deterministic if $\mathcal{S}$ is deterministic and if $|I| = 1$. The size of a PSA is the size of its underlying VPS.

A (finite or infinite) run of $\mathfrak{A}$ on a word α is a (finite or infinite) run of the underlying VPS $\mathcal{S}$ on α. Recall that a step of α is a position at which the stack height reaches a lower bound for the remainder of the word. A stair automaton only evaluates the parity condition at the steps of the word. Recall that every word α over $\widetilde{\Sigma}$ has infinitely many steps. Let $k_0 < k_1 < k_2 \cdots$ be the ordered enumeration of these steps. Now, the unique initial run $\rho_\alpha = (q_0, \sigma_0)(q_1, \sigma_1)(q_2, \sigma_2)\cdots$ of $\mathfrak{A}$ on α induces a sequence of colors $\Omega(\rho_\alpha) := \Omega(q_{k_0})\Omega(q_{k_1})\Omega(q_{k_2})\cdots$. A DPSA $\mathfrak{A}$ accepts an infinite word α if the largest color appearing infinitely often in the color sequence $\Omega(\rho_\alpha)$ induced by the unique initial run ρ_α of $\mathfrak{A}$ on α is even. The language $L(\mathfrak{A})$ of a parity stair automaton $\mathfrak{A}$ is the set of all words α that are accepted by $\mathfrak{A}$.

Lemma 7 ([10])**.** *For each BVPA $\mathfrak{A}$ there exists an effectively constructible equivalent DPSA $\mathfrak{A}_{st}$ with $|\mathfrak{A}_{st}| \in \mathcal{O}(2^{|\mathfrak{A}|})$.*

Since the stair automaton $\mathfrak{A}_{st}$ equivalent to a BVPA $\mathfrak{A}$ is deterministic, the acceptance condition collapses to the requirement that the unique run of $\mathfrak{A}_{st}$ on α must be accepting. Another important observation is that every time $\mathfrak{A}_{st}$ reaches a step of α, the stack may be cleared: Indeed, since the topmost element of the stack will never be popped after reaching a step, and since VPAs cannot inspect the top of the stack, neither this symbol, nor the ones below it have any influence on the remainder of the run.

Thus, the formula equivalent to $\mathfrak{A}_{st}$ has to specify the following constraints on the unique run of $\mathfrak{A}_{st}$ on a given input:

- There must exist some state q of even color such that the stair automaton visits q at a step,

- afterwards the automaton may never visit a higher color again at a step, and
- each visit to q at a step must be followed by another visit to q at a step.

All of these conditions can be specified by VLDL formulas in a straightforward way, since $\mathfrak{A}_{st}$ is deterministic and since there is only a finite number of colors in $\mathfrak{A}_{st}$.

Lemma 8. *For each DPSA $\mathfrak{A}$ there exists an effectively constructible equivalent VLDL formula $\varphi_{\mathfrak{A}}$ with $|\varphi_{\mathfrak{A}}| \in \mathcal{O}(|\mathfrak{A}|^2)$.*

Proof. We first construct a formula φ_{st} such that, for each word α, we have $(\alpha, k) \models \varphi_{st}$ if, and only if, $k \in \mathit{steps}(\alpha)$: Let $\mathfrak{A}_{st}$ be a VPA that accepts upon reading an unmatched return, constructed similarly to $\mathfrak{A}_r$ from Example 2. Then we can define $\varphi_{st} := \neg\langle\mathfrak{A}_{st}\rangle\mathtt{tt}$, i.e., we demand that the stack height never drops below the current level by disallowing $\mathfrak{A}_{st}$ to accept any prefix.

Let Q and Ω be the state set and the coloring function of $\mathfrak{A}$, respectively. In the remainder of this proof, we write ${}_{I'}\mathfrak{A}_{F'}$ to denote the TVPA that we obtain from combining the VPS of $\mathfrak{A}$ with the sets I' and F' of initial and final states. Additionally, we require that ${}_{I'}\mathfrak{A}_{F'}$ does not accept the empty word. This is trivially true if the intersection of I' and F' is empty, and easily achieved by adding a new initial state if it is not. Furthermore, we define $Q_{even} := \{q \in Q \mid \Omega(q) \text{ is even}\}$ and $Q_{>q} := \{q' \in Q \mid \Omega(q') > \Omega(q)\}$.

Recall that $\mathfrak{A}$ accepts a word α if the largest color seen infinitely often at a step during the unique run of $\mathfrak{A}$ on α is even. This is equivalent to the existence of a state q as characterized above. These conditions are formalized as

$$\varphi_1(q) := \langle {}_I\mathfrak{A}_{\{q\}}\rangle(\varphi_{st} \wedge [{}_{\{q\}}\mathfrak{A}_{Q_{>q}}]\neg\varphi_{st})$$

and

$$\varphi_2(q) := [{}_I\mathfrak{A}_{\{q\}}](\varphi_{st} \rightarrow \langle {}_{\{q\}}\mathfrak{A}_{\{q\}}\rangle\varphi_{st}) \ ,$$

respectively. We obtain $\varphi_{\mathfrak{A}} := \bigvee_{q \in Q_{even}} (\varphi_1(q) \wedge \varphi_2(q))$. Clearly, $\varphi_{\mathfrak{A}}$ is of quadratic size in the number of states of $\mathfrak{A}$.

The construction of $\varphi_2(q)$ relies heavily on the determinism of the DPSA $\mathfrak{A}$. If $\mathfrak{A}$ were not deterministic, the universal quantification over all runs ending in q at a step would also capture eventually rejecting partial runs. Since there only exists a single run of $\mathfrak{A}$ on the input word, however, $\varphi_{\mathfrak{A}}$ has the intended meaning. Furthermore, both $\varphi_1(q)$ and $\varphi_2(q)$ use the observation that we are able to clear the stack every time that we reach a step. Thus, although the stack contents are not carried over between the different automata, combining the automata in the formula does not change the resulting run. Hence, we have $\alpha \in L(\mathfrak{A})$ if, and only if, $(\alpha, 0) \models \varphi_{\mathfrak{A}}$ and thus $L(\mathfrak{A}) = L(\varphi_{\mathfrak{A}})$. □

Combining Lemmas 7 and 8 yields that VLDL is at least as expressive as BVPA. The construction inherits an exponential blowup from the construction of DPSAs from BVPAs and proves one direction of Theorem 6.

In the next section we show that each VLDL formula can be transformed into an equivalent VPA of exponential size. Thus, the construction from the proof of Lemma 8 yields a normal form for VLDL formulas. In particular, formulas in this normal form do not use tests and only use temporal operators up to nesting depth three.

Proposition 9. *Let φ be a VLDL formula. There exists an equivalent formula $\varphi' = \bigvee_{i=1}^{n}(\langle\mathfrak{A}_i^1\rangle(\varphi_{st} \wedge [\mathfrak{A}_i^2]\neg\varphi_{st}) \wedge [\mathfrak{A}_i^1](\varphi_{st} \rightarrow \langle\mathfrak{A}_i^3\rangle\varphi_{st}))$, for some n that is doubly-exponential in $|\varphi|$, where all $\mathfrak{A}_i^j$ share the same underlying VPS, φ_{st} is fixed over all φ, and neither the $\mathfrak{A}_i^j$ nor φ_{st} contain tests.*

Proposition 9 shows that tests are not essential for the expressiveness of VLDL. However, removing them incurs a doubly-exponential blowup. It remains open whether this blowup can be avoided.

5.2. From VLDL to 1-AJA

We now construct a BVPA equivalent to a given VLDL formula. A direct construction would incur a non-elementary blowup due to the unavoidable exponential blowup of complementing BVPAs. Moreover, it would be difficult to handle runs of the VPAs over finite words and their embedded tests, which run in parallel. Thus, we extend a construction by Faymonville and Zimmermann [14], where a similar challenge was addressed using alternating automata. Instead of alternating visibly pushdown automata [9], however, we use one-way alternating jumping automata (1-AJA), which can be translated into equivalent BVPAs of exponential size [9].

A 1-AJA $\mathfrak{A} = (Q, \widetilde{\Sigma}, \delta, I, \Omega)$ consists of a finite state set Q, a visibly pushdown alphabet $\widetilde{\Sigma}$, a transition function $\delta\colon Q \times \Sigma \rightarrow \mathcal{B}^+(\mathit{Comms}_Q)$, where $\mathit{Comms}_Q := \{\rightarrow, \rightarrow_a\} \times Q \times Q$, with $\mathcal{B}^+(\mathit{Comms}_Q)$ denoting the set of positive Boolean formulas over Comms_Q, a set $I \subseteq Q$ of initial states, and a coloring $\Omega\colon Q \rightarrow \mathbb{N}$. We define $|\mathfrak{A}| = |Q|$. Intuitively, when the automaton is in state q at position k of the word $\alpha = \alpha_0\alpha_1\alpha_2\cdots$, it guesses a set of commands $R \subseteq \mathit{Comms}_Q$ that is a model of $\delta(q, \alpha_k)$. It then spawns one copy of itself for each command $(d, q, q') \in R$ and executes the command with that copy. If $d = \rightarrow_a$ and if α_k is a matched call, the copy jumps to the position of the matching return of α_k and transitions to state q'. Otherwise, i.e., if $d = \rightarrow$, or if α_k is not a matched call, the automaton advances to position $k+1$ and transitions to state q. All copies of $\mathfrak{A}$ proceed in parallel. A single copy of $\mathfrak{A}$ accepts if the highest color visited infinitely often is even. A 1-AJA accepts α if all of its copies accept.

Example 10. Consider the property "During the run of a program, no unmatched returns may occur", which we already formalized as a VPA and as a VLDL formula in Example 2. In order to demonstrate the concepts used by 1-AJAs, we construct a 1-AJA recognizing this property.

Recall that 1-AJAs can deterministically decide whether or not a call is matched upon reading it. Hence, in order to obtain an idiomatic 1-AJA, we leverage the fact that a trace satisfies the above property if it either contains an unmatched call or if each call at stack height zero is matched.

Using this observation, we construct a 1-AJA with three states q, $q_\bot$, and $q_\top$, where $q_\bot$ and $q_\top$ serve as a rejecting and an accepting sink, respectively. The automaton "jumps" along the steps of stack height zero in state q. If it encounters an unmatched call in state q, it moves to state $q_\top$, while it moves to state $q_\bot$ upon processing a return in state q.

Formally, we define the 1-AJA $\mathfrak{A} = (\{q, q_\bot, q_\top\}, \widetilde{\Sigma}, \delta, \{q\}, \Omega)$ with

- $\delta(q, x) = \begin{cases} (\rightarrow_a, q_\bot, q) & \text{if } x \in \Sigma_c \\ (\rightarrow, q, q) & \text{if } x \in \Sigma_l \\ (\rightarrow, q_\bot, q_\bot) & \text{if } x \in \Sigma_r \end{cases}$
- $\delta(q_\bot, x) = (\rightarrow, q_\bot, q_\bot)$ for $x \in \Sigma$
- $\delta(q_\top, x) = (\rightarrow, q_\top, q_\top)$ for $x \in \Sigma$

and with

- $\Omega(q) = 0$
- $\Omega(q_\bot) = 1$
- $\Omega(q_\top) = 0$.

As argued previously, the automaton $\mathfrak{A}$ accepts those traces that satisfy the property given above.

It is known that 1-AJAs have the same expressiveness as VPAs, i.e., they characterize the class ω-VPL.

Lemma 11 ([9])**.** *For each 1-AJA $\mathfrak{A}$ there exists an effectively constructible equivalent BVPA $\mathfrak{A}_{vp}$ with $|\mathfrak{A}_{vp}| \in \mathcal{O}(2^{|\mathfrak{A}|})$.*

For a given VLDL formula φ we now inductively construct a 1-AJA that recognizes the same language as φ. The main difficulty lies in the translation of formulas of the form $\langle \mathfrak{A} \rangle \varphi$, since these require us to translate TVPAs over finite words into 1-AJAs over infinite words. We do so by adapting the idea for the translation from BVPAs to 1-AJAs by Bozzelli [9] and by combining it with the bottom-up translation from LDL into alternating automata by Faymonville and Zimmermann [14].

Lemma 12. *For each VLDL formula φ there exists an effectively constructible equivalent 1-AJA $\mathfrak{A}_\varphi$ with $|\mathfrak{A}_\varphi| \in \mathcal{O}(|\varphi|^2)$.*

Proof. We construct the automaton inductively over the structure of φ. The case $\varphi = p$ is trivial. For Boolean operations, we obtain $\mathfrak{A}_\varphi$ by closure of 1-AJAs under these operations [9]. If $\varphi = [\mathfrak{A}]\varphi'$ we use the identity $[\mathfrak{A}]\varphi' \equiv \neg\langle \mathfrak{A} \rangle \neg\varphi'$ and construct $\mathfrak{A}_{\neg\langle \mathfrak{A} \rangle \neg\varphi'}$ instead.

We now consider $\varphi = \langle \mathfrak{A} \rangle \varphi'$, where $\mathfrak{A}$ is some TVPA and construct a 1-AJA $\mathfrak{A}_\varphi$. By induction we obtain a 1-AJA $\mathfrak{A}'$ equivalent to φ'. $\mathfrak{A}_\varphi$ simulates a run of $\mathfrak{A}$ on a prefix of α and, upon acceptance, nondeterministically transitions into $\mathfrak{A}'$.

Consider an initial run of $\mathfrak{A}$ on a prefix w. Since w is finite, $steps(w)$ is finite as well. Hence, each stack height may only be encountered finitely often at a step. At the last visit to a step of a given height, $\mathfrak{A}$ either accepts, or it reads a call action. The symbol pushed onto the stack in that case does not influence the remainder of the run. We show such a run on the word *clcrrclrll* in Figure 6, where c is a call, r is a return, and l is a local action.

The idea for the simulation of the run of $\mathfrak{A}$ by $\mathfrak{A}_\varphi$ is to have a main copy of $\mathfrak{A}_\varphi$ that jumps along the steps of the input word. When $\mathfrak{A}_\varphi$ encounters a call $c \in \Sigma_c$ it guesses whether or not $\mathfrak{A}$ encounters the current stack height

α		c	l	c	r	r	c	c	l	r	l	l		$\cdots$
q	q_0	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9	q_{10}	q_{11}		$\cdots$
γ				B				B	B					
		A	A	A	A		A	A	A	A	A	A		$\ldots$
	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$		

Figure 6: Run of a VPA on the word *clcrrcclrll*.

again. If it does, then $\mathfrak{A}_\varphi$ guesses $q', q'' \in Q$ and $A \in \Gamma$ such that (q, c, q', A) is a transition of $\mathfrak{A}$, it jumps to the matching return of c with state q'' and it spawns a copy that verifies that $\mathfrak{A}$ can go from the configuration (q', A) to the configuration $(q'', \bot)$. If $\mathfrak{A}$ supposedly never returns to the current stack height, then $\mathfrak{A}_\varphi$ only guesses $q' \in Q$ and $A \in \Gamma$ such that (q, c, q', A) is a transition of $\mathfrak{A}$, moves to state q', and stores in its state space that it may not read any returns anymore. This is repeated until the main copy guesses that $\mathfrak{A}'$ accepts the prefix read so far.

This construction is not able to leverage the ability of 1-AJAs to distinguish matched and unmatched calls due to the different scope of the words processed by the simulated VPA and the simulating 1-AJA. Consider, for example, the prefix $w = clc$ of the word $\alpha = clcrrcclrll$. While the first two calls of α are clearly matched, they are unmatched in the prefix w. Hence, if we relied on this built-in mechanism of 1-AJAs, the automaton resulting from our construction would be unable to simulate acceptance of the prefix w, as it is only able to accept with its main copy.

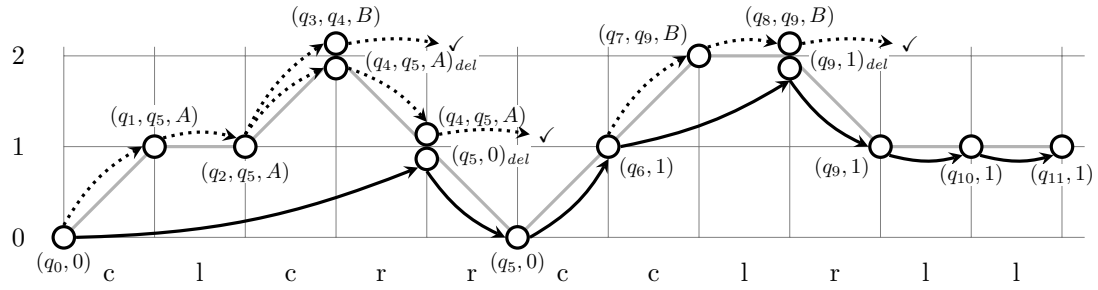

Figure 7: Simulation of the run from Figure 6 by a 1-AJA.

Figure 7 shows the run of such a 1-AJA corresponding to the run of $\mathfrak{A}$ shown in Figure 6. The gray line indicates the stack height, while the solid and dashed black paths denote the run of the main automaton and those of the verifying automata, respectively. Dotted lines indicate spawning a verifying automaton. For readability, the figure does not include copies of the automata that are spawned to verify that the tests of $\mathfrak{A}$ hold true. The main copy of the automaton uses states of the form $(q, 0)$ if it has not yet ignored any call actions, and states of the form $(q, 1)$ if it has done so. Additionally, since 1-AJAs jump to the position of a matching return instead of the succeeding position, the main copy additionally uses states of the form q_{del}, i.e., we delay further processing of the input for an additional step. The states (q, q', A) denote verification copies that verify $\mathfrak{A}$'s capability to move from the configuration (q, A) to the configuration $(q', \bot)$. The verification automata work similarly to the main

automaton, except that they assume all pushed symbols to be eventually popped and reject if they encounter an unmatched call. We now construct the 1-AJA $\mathfrak{A}_\varphi$ equivalent to $\langle\mathfrak{A}\rangle\varphi'$ formally.

Let $\mathfrak{A} = (Q^{\mathfrak{A}}, \widetilde{\Sigma}, \Gamma^{\mathfrak{A}}, \Delta^{\mathfrak{A}}, I^{\mathfrak{A}}, F^{\mathfrak{A}}, t^{\mathfrak{A}})$, let $\mathfrak{A}' = (Q', \widetilde{\Sigma}, \delta', I', \Omega')$ be the 1-AJA equivalent to φ' and, for each $\varphi_i \in \mathit{range}(t^{\mathfrak{A}})$, let $\mathfrak{A}_i = (Q^i, \widetilde{\Sigma}, \delta^i, I^i, \Omega^i)$ be a 1-AJA equivalent to φ_i. The automata $\mathfrak{A}'$ and $\mathfrak{A}_i$ are obtained by induction.

We use the set of states

$$Q := \{q, q_{del} \mid q \in (Q^{\mathfrak{A}} \times \{0,1\}) \cup (Q^{\mathfrak{A}} \times Q^{\mathfrak{A}} \times \Gamma)\} \cup \{\mathit{rej}\} \cup Q' \cup \bigcup\nolimits_{\varphi_i \in \mathit{range}(t)} Q^i \ ,$$

where the state *rej* is a rejecting sink. The states from $Q^{\mathfrak{A}} \times \{0,1\}$ are used to simulate the original automaton at steps with stack height zero ($Q^{\mathfrak{A}} \times \{0\}$) and stack height at least one ($Q^{\mathfrak{A}} \times \{1\}$), respectively.

For the sake of readability, we define the transition function for the different components of the automaton separately. We also write $(\rightarrow, q)$ and $(\rightarrow_a, q)$ as shorthands for $(\rightarrow, q, \mathit{rej})$ and $(\rightarrow_a, \mathit{rej}, q)$, respectively. Intuitively, $(\rightarrow, q)$ denotes that we demand the current symbol to be a return, a local action, or an unmatched call. Dually, the transition $(\rightarrow_a, q)$ denotes that we demand the current symbol to be a matched call.

The easiest parts of the transition function are those which control the delay states and the rejecting sink *rej*, which are defined as $\delta_{del}(q_{del}, a) = q$ and as $\delta_{sink}(\mathit{rej}, a) := (\rightarrow, \mathit{rej})$ for all $q \in (Q^{\mathfrak{A}} \times \{0,1\}) \cup (Q^{\mathfrak{A}} \times Q^{\mathfrak{A}} \times \Gamma)$ and all $a \in \Sigma$.

When encountering a final state of $\mathfrak{A}$, we need to be able to move to the successors of the initial states of $\mathfrak{A}'$ in order to model acceptance of $\mathfrak{A}$ on the finite prefix read so far. To achieve a uniform presentation, we define the auxiliary formula $\chi^f(q,a) := \bigvee_{q'_I \in I'} \delta'(q'_I, a)$ if $q \in F^{\mathfrak{A}}$ and $\chi^f(q,a) := (\rightarrow, \mathit{rej})$ otherwise.

Moreover, we need notation to denote transitions into the automata $\mathfrak{A}_i$ implementing the tests of $\mathfrak{A}$. More precisely, since we only transition into these automata upon leaving the states labeled with the respective test, we need to transition into the successors of one of the initial states of the implementing automata. To this end, we define the auxiliary formula $\theta^a_q := \bigvee_{q_I \in I^i} \delta^i(q_I, a)$, where $t(q) = \varphi_i$.

For local actions the main copy of the automaton can simply simulate the behavior of $\mathfrak{A}$ on the input word. Hence we have

$$\delta_{main}((q,b),l) := \Big[\chi^f(q,l) \vee \bigvee\nolimits_{(q,l,q') \in \Delta} (\rightarrow, (q', b))\Big] \wedge \theta^l_q \quad \text{for } l \in \Sigma_l, b \in \{0,1\}$$

When reading a call, the automaton nondeterministically guesses whether it jumps to the matching return or whether it simulates the state transition while ignoring the effects on the stack. In the former case, it guesses a transition $(q, c, q', A) \in \Delta$ and a state $q'' \in Q$, spawns a verification automaton verifying that it is possible to go from q' to q'' by popping A off the stack in the final transition, and continues at the matching return in state q''. In the latter case

it ignores the effects on the stack.

$$\delta_{main}((q,b),c) := \Big[\chi^f(q,c) \vee \bigvee_{(q,c,q',A)\in\Delta, q''\in Q} \Big[(\rightarrow,(q',q'',A)) \wedge (\rightarrow_a,(q'',b)_{del})\Big] \vee \bigvee_{(q,c,q',A)\in\Delta} (\rightarrow,(q',1))\Big] \wedge \theta_q^c \quad \text{for } c\in\Sigma_c, b\in\{0,1\}$$

The main automaton may only handle returns as long as it has not skipped any calls. If it encounters a return after having skipped a push action, it rejects the input word, since the return falsifies its earlier guess of an unmatched call.

$$\delta_{main}((q,0),r) := \Big[\chi^f(q,r) \vee \bigvee_{(q,r,\bot,q')\in\Delta} (\rightarrow,(q',0))\Big] \wedge \theta_q^r \quad \text{for } r\in\Sigma_r$$
$$\delta_{main}((q,1),r) := (\rightarrow, rej) \quad \text{for } r\in\Sigma_r$$

The transition function δ_{main} determines the behavior of the main automaton. It remains to define the behavior of the verifying automata. These behave similarly to the main automaton on reading local actions and calls. The main difference in handling calls is that these automata do not need to guess whether or not a call is matched: Since they are only spawned on reading supposedly matched calls and accept upon reading the matching return, all calls they encounter must be matched as well. Additionally, they never transition to the automaton $\mathfrak{A}'$, but merely to the automaton implementing the test of the current state upon having verified their guess.

$$\delta_{ver}((q,q',A),l) := \Big[\bigvee_{(q,l,q'')\in\Delta} (\rightarrow,(q'',q',A))\Big] \wedge \theta_q^l \quad \text{if } l\in\Sigma_l$$
$$\delta_{ver}((q,q',A),c) := \Big[\bigvee_{(q,c,q'',A')\in\Delta, q'''\in Q} (\rightarrow,(q'',q''',A')) \wedge (\rightarrow_a,(q''',q',A)_{del})\Big] \wedge \theta_q^c \quad \text{if } c\in\Sigma_c$$
$$\delta_{ver}((q,q',A),r) := \theta_q^r \quad \text{if } r\in\Sigma_r, (q,r,A,q')\in\Delta$$
$$\delta_{ver}((q,q',A),r) := (\rightarrow, rej) \quad \text{if } r\in\Sigma_r, (q,r,A,q')\notin\Delta$$

We then define the complete transition function δ of $\mathfrak{A}_\varphi$ as the union of the previously defined partial transition functions. Since their domains are pairwise disjoint, this union is well-defined.

$$\delta := \delta_{sink} \cup \delta_{del} \cup \delta' \cup \bigcup_{\varphi_i\in range(t)} \delta^i \cup \delta_{main} \cup \delta_{ver}$$

The coloring of $\mathfrak{A}_\varphi$ is obtained by copying the coloring of $\mathfrak{A}'$ and the $\mathfrak{A}^i$ and by coloring all states resulting from the translation of $\mathfrak{A}$ with 1. Thus, we force every path of the run of $\mathfrak{A}$ to eventually leave this part of the automaton, since this automaton only accepts a finite prefix of the input word. The 1-AJA

$$\mathfrak{A}_\varphi := (Q, \widetilde{\Sigma}, \delta, I^{\mathfrak{A}} \times \{0\}, \Omega \cup \Omega' \cup \bigcup_{\varphi_i\in range(t^{\mathfrak{A}})} \Omega^i)$$

then recognizes the language of $\varphi = \langle\mathfrak{A}\rangle\varphi'$, where $\Omega : q \mapsto 1$ for all $q \in \{q, q_{del} \mid q \in (Q^{\mathfrak{A}} \times \{0,1\}) \cup (Q^{\mathfrak{A}} \times Q^{\mathfrak{A}} \times \Gamma)\} \cup \{rej\}$. □

By combining Lemmas 11 and 12 we see that BVPAs are at least as expressive as VLDL. This proves the direction from logic to automata of Theorem 6. The construction via 1-AJAs yields automata of exponential size in the number of states. This blowup is unavoidable, which we show by relying on the analogous lower bound for translating LTL into Büchi automata. This lower bound is obtained by encoding an exponentially bounded counter in LTL.

Lemma 13. *There exists a pushdown alphabet $\widetilde{\Sigma}$ such that for all $n \in \mathbb{N}$ there exists a language L_n that is defined by a VLDL formula over $\widetilde{\Sigma}$ of polynomial size in n, but every BVPA over $\widetilde{\Sigma}$ recognizing L_n has at least exponentially many states in n.*

Proof. We use the pushdown alphabet $\widetilde{\Sigma} = (\Sigma_c, \Sigma_r, \Sigma_l) = (\emptyset, \emptyset, \{0, 1, \#\})$. For any $n \in \mathbb{N}$ and any $i \in [0; 2^n - 1]$ we write $\langle i \rangle_n$ to denote the binary encoding of i using n bits. Moreover, we define the language $L_n := \{\#\langle 0\rangle_n \# \cdots \# \langle 2^n - 1\rangle_n \#^\omega\}$, which only contains a single word encoding an n-bit counter. It is known that there exists an LTL formula of polynomial length in n that defines L_n. Thus, there also exists a VLDL formula of polynomial length defining this language due to Lemma 1.

Furthermore, since all symbols are local actions, any BVPA recognizing L_n cannot use its stack and thus has to work like a traditional finite automaton with Büchi acceptance. Again, it is known that all Büchi automata recognizing L_n have at least exponentially many states in n. Consequently, all BVPAs recognizing L_n have at least exponentially many states in n. □

After having shown that VLDL has the same expressiveness as BVPAs, we now turn our attention to several decision problems for this logic. Namely, we study the satisfiability and the validity problem, as well as the model checking problem. Moreover, we consider the problem of solving visibly pushdown games with VLDL winning conditions.

Before studying these decision problems, however, we contrast and compare our logic with the logic VLTL. This logic has the same expressiveness as VLDL and features a similar technical core, i.e., it contains temporal operators that are guarded with visibly pushdown languages.

6. Comparison between VLDL and VLTL

The logic VLTL, introduced by Bozzelli and Sánchez [12], follows the same basic idea as VLDL, i.e., that of augmenting temporal modalities with visibly pushdown languages in order to obtain a logic that captures the ω-visibly pushdown languages. The main difference between the two logics lies in the choice of temporal operators as well as in the specification of the guards.

Since both VLDL and VLTL capture the class of ω-visibly pushdown languages, for each VLDL formula there exists an equivalent VLTL formula and vice versa. Moreover, as there exist effective translations to and from VPA for VLTL [17], these equivalent formulas can be effectively constructed. However, both VLDL and VLTL incur an exponential blowup when translating to and from VPAs. Hence, these naïve translations between the two logics incur a doubly exponential blowup in both directions. In this section we investigate translations with a smaller blowup between fragments of the two logics.

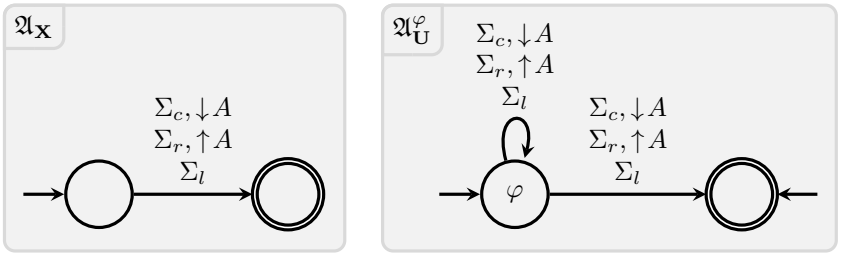

Figure 8: The automata $\mathfrak{A}_{\mathbf{X}}$ and $\mathfrak{A}^{\varphi}_{\mathbf{U}}$ used for translating the formulas $\mathbf{X}\psi$ and $\varphi\mathbf{U}\psi$ into VLDL.

Our logic only uses guarded variants of the operators **F** and **G** of LTL. Similarly to LTL, these operators are dual due to the presence of negation in VLDL: Removing either of them from the logic does not change its expressiveness or the computational complexity of the decision problems for this logic. Moreover, due to the identities $\mathbf{X}\varphi \equiv [\mathfrak{A}_{\mathbf{X}}]\varphi$ and $\varphi\mathbf{U}\psi \equiv \langle\mathfrak{A}^{\varphi}_{\mathbf{U}}\rangle\psi$, with the automata $\mathfrak{A}_{\mathbf{X}}$ and $\mathfrak{A}^{\varphi}_{\mathbf{U}}$ as shown in Figure 8, the next- and the until-operator can be added to VLDL without changing its expressiveness or the complexity of its associated decision problems.

In contrast, the logic VLTL has a much richer suite of temporal operators. In addition to an analogue of the guarded eventually operator, called the sequencing operator, it contains guarded versions of the until and the weak until of LTL, called the power operator and the weak power operator, respectively. Moreover, VLTL allows for the use of past versions of these operators. A more in-depth introduction to the syntax and semantics of VLTL can be found in the work of Bozzelli and Sánchez [12].

The guards of VLDL are specified as nondeterministic visibly pushdown automata with tests. These automata extend VPAs, the canonical acceptor for visibly pushdown languages, with tests in order to enable concise rewritings of LTL formulas into VLDL. In contrast, the guards in VLTL are specified via visibly rational expressions (VREs) [17], an extension of regular expressions that captures visibly pushdown languages.

It is known that VPAs without tests can be transformed into VREs and vice versa with an exponential blowup in both directions [17]. Due to this transformation, we are able to directly translate the test-free fragment of VLDL into VLTL without taking a detour via VPAs.

Theorem 14. *For each VLDL formula φ in which all automata are test-free there exists an effectively constructible equivalent VLTL formula ψ with $|\psi| \in \mathcal{O}(2^{|\varphi|})$.*

Proof. Since both logics feature the standard Boolean connectives, it suffices to describe how to translate the guarded eventually operator due to the equivalence $[\mathfrak{A}]\varphi' \equiv \neg\langle\mathfrak{A}\rangle\neg\varphi'$. Due to the similar semantics, this operator can easily be translated into an application of the sequencing operator. This transformation does, however, incur an exponential blowup inherited from the blowup in the transformation of VPAs into VREs. The resulting VLTL formula ψ is thus of exponential size in the size of φ, contains only the (future) sequencing operator and defines the same language as φ. □

For the other direction, i.e., the translation from VLTL into VLDL, it is trivial to translate the sequencing operator of VLTL into the guarded eventually operator of VLTL. Moreover, it is straightforward to translate the power operator into VLDL by slightly extending the above translation of the **U** operator from LTL. The greatest hurdle for a translation is, however, presented by the weak power operator of VLTL. This operator is the guarded analogue of the weak until operator of LTL and allows for an infinite sequence of finite words satisfying the guard. Thus, intuitively, the automaton representing the guard is required to restart infinitely often after visiting an accepting state. This Büchi condition over a nondeterministic automaton is not directly expressible in VLDL and it remains open how to concisely translate the weak power operator of VLTL into VLDL.

Both VLDL and VLTL capture the class of ω-VPLs. In fact, Proposition 9 yields that even the test-free fragment of VLDL suffices to characterize this class. Hence, we obtain that a small fragment of VLTL indeed already suffices to capture the ω-visibly pushdown languages.

Corollary 15. *For each ω-VPL L there exists a VLTL formula ψ that only uses the future sequencing operator with $L(\psi) = L$.*

Previous results have already shown that the past modalities of VLTL do not contribute to its expressiveness [12]. Corollary 15 extends this result by showing that the future power and future weak power operator are also not required for the expressiveness of VLTL. It does, however, remain open whether or not these operators contribute to the conciseness of VLTL.

7. Satisfiability and Validity are ExpTime-complete

We say that a VLDL formula φ is satisfiable if it has a model. Dually, we say that φ is valid if all words are models of φ. Instances of the satisfiability and validity problem consist of a VLDL formula φ and ask whether φ is satisfiable and valid, respectively. Both problems are decidable in exponential time. We also show both problems to be ExpTime-hard.

We obtain this result by modifying a proof by Bouajjani et al. [11], with which they showed model checking pushdown systems against LTL specifications to be ExpTime-hard. We adapt this proof for showing ExpTime-hardness of the satisfiability problem by encoding the transition system into the resulting formula.

Theorem 16. *The satisfiability problem and the validity problem for VLDL are* ExpTime*-complete.*

Proof. Due to duality of the problems, we only show ExpTime-completeness of the satisfiability problem. Membership follows from the 1-AJA-emptiness-problem being in ExpTime [9] and Lemma 12.

It remains to show ExpTime-hardness, which we prove by providing a reduction from the word problem for polynomially space-bounded alternating Turing machines. This problem asks whether a given word is accepted by a given alternating Turing machine which only uses polynomial space in the size of the input. Since a terminating run of an alternating Turing machine is a finite tree,

it can be serialized as a word, where the subtrees are delimited by special symbols. Such a word can then be checked for correctly encoding some tree using the stack of a VPA. Adherence to the transition relation, as well as the property that the tree describes an accepting run of the Turing machine can be checked mostly locally without using the stack. These constraints can be expressed in VLDL, such that their conjunction is satisfiable if, and only if, there exists an accepting run of the Turing machine on the word, i.e., if the Turing machine accepts the word.

Formally, an alternating Turing machine (ATM) [18] $\mathcal{T} = (Q_\exists, Q_\forall, \Gamma, q_I, \Delta, F)$ consists of two finite disjoint sets $Q_\exists$ and $Q_\forall$ of states, which are called existential and universal states, respectively, for which we write $Q := Q_\exists \cup Q_\forall$, a tape alphabet Γ containing a blank symbol B, an initial state $q_I \in Q \setminus F$, a transition relation $\Delta \subseteq Q \times \Gamma \times Q \times \Gamma \times \{L, R\}$, and a set of final states $F \subseteq Q$.

Let $p(n)$ be some polynomial. A configuration c of a $p(n)$-bounded ATM $\mathcal{T}$ on an input word w is a word of length $p(|w|) + 1$ over the alphabet $\Gamma \cup Q$ that contains exactly one symbol from Q. Let $\mathit{Conf} := \Gamma^* Q \Gamma^* \cap (Q \cup \Gamma)^{p(|w|)+1}$ denote the set of such configurations. If $c \in \mathit{Conf}$ contains a symbol from $Q_\exists$ ($Q_\forall$), we call c existential (universal). Analogously, if c contains a symbol from F, we call c accepting. Furthermore, a transition $(q, a, q', a', D) \in \Delta$ with $D \in \{L, R\}$ is existential (universal), if $q \in Q_\exists$ ($q \in Q_\forall$). We assume w.l.o.g. that every configuration has exactly two applicable transitions and that the initial state is not final.

A run of a $p(n)$-bounded ATM $\mathcal{T}$ on w is a finite tree that is labeled with configurations of $\mathcal{T}$ on w. Each non-terminal vertex has either one or two successors, depending on whether it is labeled with an existential or a universal configuration. Each successor is labeled with a successor configuration. These successors have to be labeled by one or two successor configurations. A run is accepting if all terminal vertices are labeled with accepting configurations. An ATM $\mathcal{T}$ accepts a word w if there exists an accepting run of $\mathcal{T}$ on w.

An instance of the word problem consists of a $p(n)$-space-bounded ATM $\mathcal{T}$ and a word w and asks whether or not $\mathcal{T}$ accepts w. This problem is ExpTime-hard [18].

We encode runs of $\mathcal{T}$ by linearizing them as words using tags of the form $<_\tau^i$ and $>_\tau^i$ for $i \in \{1, 2\}$ to delimit the encoding of the first and second subtree of a vertex (recall that we assume that every configuration has at most two successors). Here, τ denotes the transition that is applied to obtain the configuration of the root of this subtree. Moreover, we use the tags $<_\ell$ and $>_\ell$ to denote leaves.

Formally, we define the pushdown alphabet $\widetilde{\Sigma} = (\Sigma_c, \Sigma_r, \Sigma_l)$ with

- $\Sigma_c = ((Q \cup \Gamma) \times \{\downarrow\}) \cup \{<_\tau^1 \mid \tau \in \Delta\} \cup \{<_\ell\}$,
- $\Sigma_r = ((Q \cup \Gamma) \times \{\uparrow\}) \cup \{>_\tau^1 \mid \tau \text{ existential}\} \cup \{>_\tau^2 \mid \tau \text{ universal}\} \cup \{>_\ell\}$, and
- $\Sigma_l = \{>_\tau^1, <_\tau^2 \mid \tau \text{ universal}\} \cup \{\#\}$.

Let $\mathit{Tags} = \{<_\tau^1, >_\tau^1 \mid \tau \text{ existential}\} \cup \{<_\tau^1, >_\tau^1, <_\tau^2, >_\tau^2 \mid \tau \text{ universal}\} \cup \{<_\ell, >_\ell\}$.

For $C = c_0 \cdots c_n \in \mathit{Conf}^*$ and $d \in \{\downarrow, \uparrow\}$, let $(C, d) := (c_0, d) \cdots (c_n, d)$, which we lift to languages in the straightforward way. Furthermore, for an arbitrary word $w = w_0 \cdots w_n$, let $w^r := w_n \cdots w_0$. Let $c \in \mathit{Conf}$. We define $\mathit{push}(c) := (c, \downarrow)$ and $\mathit{pop}(c) := (c^r, \uparrow)$

Using this, we encode a run of $\mathcal{T}$ by recursively iterating over its vertices v as follows:

- $enc(v) := <_\ell \cdot push(c) \cdot >_\ell \cdot pop(c)$, if v is a leaf labeled with the configuration c.
- $enc(v) := <^1_\tau \cdot push(c) \cdot enc(v_1) \cdot >^1_\tau \cdot pop(c)$, if v has a single child v_1, v is labeled by the (existential) configuration c, and τ is the transition that is applied to c to obtain the label of v_1.
- $enc(v) := <^1_{\tau_1} \cdot push(c) \cdot enc(v_1) \cdot >^1_{\tau_1} \cdot pop(c) \cdot <^2_{\tau_2} \cdot push(c) \cdot enc(v_2) \cdot >^2_{\tau_2} \cdot pop(c)$, if v has two children v_1 and v_2, v is labeled by the (universal) configuration c, and τ_i, for $i \in \{1, 2\}$, is the transition that is applied to c to obtain the label of v_i.

Thus, a complete run with root v is encoded by $enc(v) \cdot \#^\omega$. Our goal is to construct a formula that is satisfied only by words that encode initial accepting runs of $\mathcal{T}$ on w. To this end, we need to formalize the following six conditions on an infinite word $\alpha \in \Sigma^\omega$:

1. $\alpha \in (Tags \cdot Conf)^+ \cdot \#^\omega$ and begins with $<^1_\tau \cdot (c_I, \downarrow)$, where c_I is the initial configuration of $\mathcal{T}$ on w and where τ is a transition that is applicable to c_I.
2. Every $<^i_\tau$, $i \in \{1, 2\}$, is directly followed by $(c, \downarrow)$ for some configuration c to which τ is applicable. Furthermore, say the stack height is n after this infix. Then, we require that this stack height is reached again at a later position, and at the first such position, the infix $>^1_\tau \cdot (c^r, \uparrow)$ starts.
3. Every $>^1_\tau$ with universal τ, which is directly followed by $(c^r, \uparrow)$ for some configuration c (assuming the previous condition is satisfied), is directly followed by $(c^r, \uparrow) \cdot <^2_{\tau'} \cdot (c, \downarrow)$, where $\tau' \neq \tau$ is the unique other transition that is applicable to c.
4. Every $<^i_\tau$, $i \in \{1, 2\}$, is directly followed by $(c, \downarrow) < (c', \downarrow)$ for some $< \in \{<^1_\tau \mid \tau \in \Delta\} \cup \{<_\ell\}$ such that τ is applicable to c and c' is the corresponding successor configuration.
5. Every $<_\ell$ is directly followed by $(c, \downarrow) >_\ell (c^r, \uparrow)$ for some accepting configuration of $\mathcal{T}$.
6. Stack height zero has to be reached after a non-empty prefix, and from the first such position onwards, only # appears.

It is straightforward to come up with polynomially-sized VLDL formulas expressing these conditions (note that only the second and sixth condition require non-trivial usage of the stack). Furthermore, α satisfies the conjunction of these properties if, and only if, it encodes an accepting run of $\mathcal{T}$ on w. Thus, as the word problem for polynomially space-bounded ATMs is ExpTime-hard, the satisfiability problem for VLDL is ExpTime-hard as well. □

8. Model Checking is ExpTime-complete

We now consider the model checking problem for VLDL. An instance of this problem consists of a VPS $\mathcal{S}$, an initial state q_I of $\mathcal{S}$, and a VLDL formula φ and asks whether $traces(\mathcal{S}, q_I) \subseteq L(\varphi)$ holds true, where $traces(\mathcal{S}, q_I)$ denotes the set obtained by mapping each run of $\mathcal{S}$ starting in q_I to the sequence of

labels of the traversed edges. This problem is decidable in exponential time due to Lemma 12 and an exponential-time model checking algorithm for 1-AJAs [9]. Moreover, the problem is ExpTime-hard, as it subsumes the validity problem.

Theorem 17. *Model checking VLDL specifications against VPS's is* ExpTime*-complete.*

Proof. Membership in ExpTime follows from Lemma 12 and the membership of the problem of checking visibly pushdown systems against 1-AJA specifications in ExpTime [9]. Moreover, since the validity problem for VLDL is ExpTime-hard and since validity of φ is equivalent to $\mathit{traces}(\mathcal{S}_{univ}) \subseteq \varphi$, where $\mathcal{S}_{univ}$ with $\mathit{traces}(\mathcal{S}_{univ}) = \Sigma^\omega$ is effectively constructible in constant time, the model checking problem for VLDL is ExpTime-hard as well. □

9. Solving VLDL Games is 3ExpTime-complete

In this section we investigate visibly pushdown games with winning conditions given by VLDL formulas. We consider games with two players, called Player 0 and Player 1, respectively.

A two-player game with VLDL winning condition $\mathcal{G} = (V_0, V_1, \Sigma, E, v_I, \ell, \varphi)$ consists of two disjoint, at most countably infinite sets V_0 and V_1 of vertices, where we define $V := V_0 \cup V_1$, a finite alphabet Σ, a set of edges $E \subseteq V \times V$, an initial state $v_I \in V$, a labeling $\ell\colon V \to \Sigma$, and a VLDL formula φ over some partition of Σ, called the winning condition. In order to avoid dealing with finite plays, we demand that for each $v \in V$ there exists a $v' \in V$ with $(v, v') \in E$.

A play $\pi = v_0 v_1 v_2 \cdots$ of $\mathcal{G}$ is an infinite sequence of vertices of $\mathcal{G}$ with $(v_i, v_{i+1}) \in E$ for all $i \geq 0$. The play π is initial if $v_0 = v_I$. It is winning for Player 0 if $\ell(v_1)\ell(v_2)\ell(v_3)\cdots$ is a model of φ.[3] Otherwise π is winning for Player 1.

A strategy for Player i is a function $\sigma\colon V^*V_i \to V$, such that $(v, \sigma(w \cdot v)) \in E$ for all $v \in V_i$, $w \in V^*$. A play $\pi = v_0 v_1 v_2 \cdots$ is consistent with σ if $\sigma(v_0 \cdots v_n) = v_{n+1}$ for all finite prefixes $v_0 \cdots v_n$ of π with $v_n \in V_i$. A strategy σ is winning for Player i if all initial plays that are consistent with σ are winning for that player. We say that Player i wins $\mathcal{G}$ if she has a winning strategy. If either player wins $\mathcal{G}$, we say that $\mathcal{G}$ is determined.

A visibly pushdown game (VPG) with a VLDL winning condition $\mathcal{H} = (\mathcal{S}, Q_0, Q_1, q_I, \varphi)$ consists of a VPS $\mathcal{S} = (Q, \widetilde{\Sigma}, \Gamma, \Delta)$, a partition of Q into Q_0 and Q_1, an initial state $q_I \in Q$, and a VLDL formula φ over $\widetilde{\Sigma}$. Recall that Cont_Γ denotes the set of stack contents over Γ. The VPG $\mathcal{H}$ then induces the two-player game $\mathcal{G}_\mathcal{H} = (V_0, V_1, \Sigma, E, v_I, \ell, \varphi)$ with $V_i := Q_i \times \mathit{Cont}_\Gamma \times \Sigma$, $v_I = (q_I, \bot, a)$ for some $a \in \Sigma$ (recall that the trace disregards the label of the initial vertex), $((q, \gamma, a), (q', \gamma', a')) \in E$ if there is an a'-labeled edge from (q, γ) to (q, γ') in the configuration graph $G_\mathcal{S}$, and $\ell\colon (q, \gamma, a) \mapsto a$. Solving a VPG $\mathcal{H}$ means deciding whether Player 0 wins $\mathcal{G}_\mathcal{H}$.

Proposition 18. *VPGs with VLDL winning conditions are determined.*

[3] For technical reasons, the sequence of labels omits the label of the first vertex.

Proof. Since each VLDL formula defines a language in ω-VPL due to Theorem 6, each VPG with VLDL winning condition is equivalent to a VPG with an ω-VPL winning condition. These are known to be determined [10]. □

We show that solving VPGs with winning conditions specified in VLDL is harder than solving VPGs with winning conditions specified by BVPAs. Indeed, the problem is 3ExpTime-complete. We obtain this result by adapting a proof by Löding et al. [10], which shows 3ExpTime-hardness of solving pushdown games with LTL winning conditions. The technical core of our proof lies in the translation of pushdown games into visibly pushdown games.

Theorem 19. *Solving VPGs with VLDL winning conditions is* 3ExpTime*-complete.*

Proof. We solve VPGs with VLDL winning conditions by first constructing a BVPA $\mathfrak{A}_\varphi$ of exponential size from the winning condition φ and by then solving the resulting visibly pushdown game with a BVPA winning condition [10]. As VPGs with BVPA winning conditions can be solved in doubly-exponential time in the size of the BVPA and in exponential time in the size of the VPS, this approach takes triply-exponential time in $|\varphi|$ and exponential time in $|\mathcal{S}|$.

We show 3ExpTime-hardness of the problem by a reduction from solving pushdown games with LTL winning conditions, which is known to be 3ExpTime-complete [10]. A pushdown game with an LTL winning condition $\mathcal{H} = (\mathcal{S}, V_0, V_1, \psi)$ is defined similarly to a VPG, except for the relaxation that $\mathcal{S}$ may now be a traditional pushdown system instead of a visibly pushdown system. Specifically, we have $\Delta \subseteq (Q \times \Gamma \times \Sigma \times Q \times \Gamma^{\leq 2})$, where $\Gamma^{\leq 2}$ denotes the set of all words over Γ of at most two letters. Stack symbols are popped off the stack using transitions of the form $(q, A, a, q', \varepsilon)$, the top of the stack can be tested and changed with transitions of the form (q, A, a, q', B), and pushes are realized with transitions of the form (q, A, a, q', BC). Additionally, the winning condition is given as an LTL formula instead of a VLDL formula. The two-player game $\mathcal{G}_\mathcal{H}$ is defined analogously to the visibly pushdown game.

Since the pushdown game admits transitions such as (q, A, a, q', BC), which pop A off the stack and push B and C onto it, we need to split such transitions into several transitions in the visibly pushdown game. We modify the original game such that every transition of the original game is modeled by three transitions in the visibly pushdown game, up to two of which may be dummy actions that do not change the stack. As each transition may perform at most three operations on the stack, we can keep track of the list of changes still to be performed in the state space. We perform these actions using dummy letters c and l, which we add to Σ and read while performing the required actions on the stack. We choose the vertices $V_i' = V_i \cup (V_i \times (\Gamma \cup \{\#\})^{\leq 2})$ for $i \in \{0, 1\}$ and the alphabet $\widetilde{\Sigma} = (\{c\}, \Sigma, \{l\})$.

We transform $\mathcal{H}$ as shown in Figure 9 and obtain the VPG $\mathcal{H}'$. Moreover, we transform the winning condition ψ of $\mathcal{H}$ into ψ' by inductively replacing each occurrence of $\mathbf{X}\psi$ by $\mathbf{X}^3\psi'$ and each occurrence of $\psi_1 \mathbf{U} \psi_2$ by $(\psi_1' \vee c \vee l)\mathbf{U}(\psi_2' \wedge \neg c \wedge \neg l)$. We subsequently translate the resulting LTL formula ψ' into an equivalent VLDL formula φ using Lemma 1. The input player wins $\mathcal{H}'$ with the winning condition φ if, and only if, he wins $\mathcal{H}$ with the winning condition ψ. Hence, solving VPGs with VLDL winning conditions is 3ExpTime-hard. □

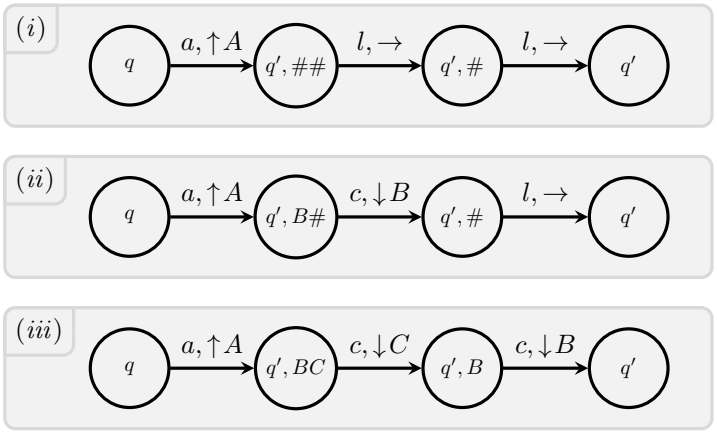

Figure 9: Construction of a VPG from a pushdown game for transitions of the forms (i) $(q, a, A, q', \varepsilon)$, (ii) (q, a, A, q', B), and (iii) (q, a, A, q', BC).

Moreover, Löding et al. have shown that in a visibly pushdown game with a winning condition given by a BVPA, Player 0, in general, requires infinite memory in order to win [10]. Thus, there is, in general, no winning strategy for her that is implemented by a finite automaton with output.[4] Such automata are sufficient, e.g., for omega-regular games on finite graphs. As we can translate BVPAs into VLDL formulas, we obtain the same lower bound for VPGs with VLDL winning conditions.

Moreover, for each VPG $\mathcal{G}$ with winning condition φ, we can easily construct the game $\mathcal{G}'$ over the same VPS by exchanging the states of Player 0 and Player 1 and obtain that Player 0 wins $\mathcal{G}$ with winning condition φ if, and only if, she loses $\mathcal{G}'$ with winning condition $\neg\varphi$. Hence, Player 1 requires, in general, infinite memory as well in order to win a VPG with VLDL winning condition.

Corollary 20. *There exists a VPG $\mathcal{G}$ with VLDL winning condition such that Player 0 wins $\mathcal{G}$, but requires infinite memory to do so. Similarly, there exists a VPG $\mathcal{G}'$ with VLDL winning condition such that Player 1 win $\mathcal{G}'$, but requires infinite memory to do so.*

10. Deterministic Pushdown Linear Dynamic Logic

In this work we extended LDL by replacing the regular languages used as guards for the temporal operators by visibly pushdown languages. We obtain an even more powerful logic by using more expressive languages as guards, e.g., deterministic pushdown languages, which have deterministic pushdown automata (DPDA) as their canonical acceptors. However, all relevant decision problems for the resulting logic called Deterministic Pushdown Linear Dynamic Logic (DPLDL) are undecidable, most importantly the satisfiability problem.

Theorem 21. *The satisfiability problem for DPLDL is undecidable.*

[4] See the work by Löding et al. for a formal definition of memory [10].

Proof. We reduce the problem of deciding nonemptiness of the intersection of two DPDA, which is known to be undecidable [19], to the satisfiability problem for DPLDL. Let $\mathfrak{A}_1$ and $\mathfrak{A}_2$ be two DPDA over a shared alphabet Σ, pick $\# \notin \Sigma$ and consider $\varphi := \langle \mathfrak{A}_1 \rangle \# \wedge \langle \mathfrak{A}_2 \rangle \#$. Then φ is satisfiable if, and only if, $L(\mathfrak{A}_1) \cap L(\mathfrak{A}_2) \neq \emptyset$. Hence satisfiability of DPLDL is undecidable. □

As the satisfiability problem reduces to model checking and to solving pushdown games with DPLDL winning conditions, both problems are also undecidable.

Corollary 22. *The validity problem and the problem of checking DPLDL specifications against VPS's as well as the problem of solving pushdown games against DPLDL winning conditions are undecidable.*

Since every DPDA is also a PDA, the extension of DPLDL by nondeterministic pushdown automata inherits these undecidability results from DPLDL. Thus, VLDL is, to the best of our knowledge, the most expressive logic that combines the temporal modalities of LDL with guards specified by languages over finite words and still has decidable decision problems, just as VLTL is the most expressive logic that combines LTL with VREs that has the same property.

11. Conclusion

We have introduced Visibly Linear Dynamic Logic (VLDL) which strengthens Linear Dynamic Logic (LDL) by replacing the regular languages used as guards in the latter logic with visibly pushdown languages. VLDL characterizes the class of ω-visibly pushdown languages. We have provided effective translations from VLDL to BVPA and vice versa with an exponential blowup in size in both directions. From automata to logic, this blowup cannot be avoided while it remains open whether or not it can be avoided in the other direction.

Figure 10 shows the formalisms that capture ω-VPL as well as CaRet and the translations between them. Our constructions are marked by solid lines, all others by dotted lines. All constructions are annotated with the blowup they incur. In that figure we include a translation from DPSA to 1-AJA. This translation can easily be obtained by adapting the "guess-and-verify"-construction used in the translation of VLDL into 1-AJAs in the proof of Lemma 12 in order to directly translate DPSA into 1-AJA without a detour via VLDL.

In particular, there exist translations between VLTL and VLDL via BVPAs that incur a doubly-exponential blowup in both directions, as shown in Figure 10. In spite of this blowup the satisfiability problem and the model checking problem for both logics are ExpTime-complete. It remains open whether there exist efficient translations between the two logics.

We showed the satisfiability and the emptiness problem for VLDL, as well as model checking visibly pushdown systems against VLDL specifications, to be ExpTime-complete. Also, we proved that solving visibly pushdown games with VLDL winning conditions is 3ExpTime-complete.

Extending VLDL by replacing the guards with a more expressive family of languages quickly yields undecidable decision problems. In fact, using deterministic pushdown languages as guards already renders all decision problems discussed in this work undecidable.

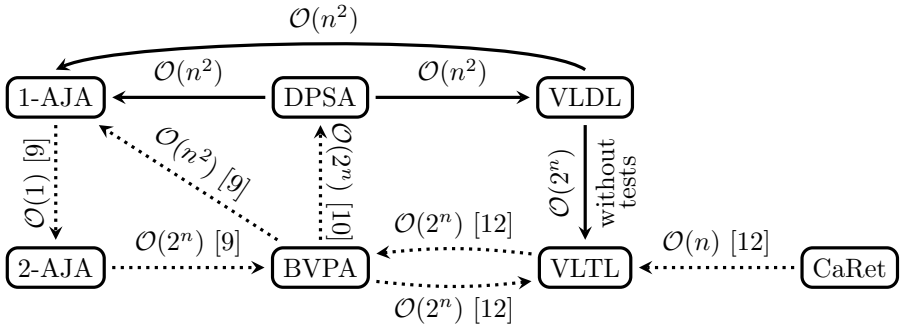

Figure 10: Formalisms capturing (subsets of) ω-VPL and translations between them.

In contrast to LDL [4] and VLTL [12], VLDL uses automata to define guards instead of regular or visibly rational expressions. We are currently investigating a variant of VLDL where the VPAs guarding the temporal operators are replaced by visibly rational expressions (with tests), which is closer in spirit to LDL and VLTL.

Finally, another algorithm for solving the satisfiability problem and the model checking problem for VLDL has recently been proposed [13]. This algorithm also first translates the given VLDL formula into a 1-AJA, but subsequently constructs an emptiness-equivalent tree automaton from the 1-AJA. Hence, this construction ultimately reduces the problem of VLDL satisfiability and VLDL model checking to solving Büchi games. This contrasts our algorithm presented in this work, in which we ultimately reduce the problems to checking emptiness of context-free grammars. We are working on an implementation of the two algorithms in order to evaluate their respective strengths and weaknesses.

Acknowledgements The authors would like to thank Laura Bozzelli for providing the full version of [9] and Christof Löding for pointing out the 3ExpTime-hardness of solving infinite games for visibly pushdown games against LTL specifications. Moreover, we would like to thank the reviewers for their valuable and constructive comments which improved the paper considerably. In particular, we thank an anonymous reviewer for the observation formalized in Corollary 15.

References

[1] A. Weinert, M. Zimmermann, Visibly linear dynamic logic, in: A. Lal, S. Akshay, S. Saurabh, S. Sen (Eds.), FSTTCS 2016, Vol. 65 of LIPIcs, Schloss Dagstuhl - LZI, 2016, pp. 28:1–28:14. doi:10.4230/LIPIcs.FSTTCS.2016.28.

[2] A. Pnueli, The temporal logic of programs, in: FOCS 1977, IEEE, 1977, pp. 46–57. doi:10.1109/SFCS.1977.32.

[3] M. Leucker, C. Sanchéz, Regular linear temporal logic, in: C. B. Jones, Z. Liu, J. Woodcock (Eds.), ICTAC 2007, no. 4711 in LNCS. doi:10.1007/978-3-540-75292-9_20.

[4] M. Vardi, The rise and fall of LTL, in: G. D'Agostino, S. L. Torre (Eds.), EPTCS 54, 2011.

[5] M. Vardi, P. Wolper, Reasoning about infinite computations, Inf. and Comp. 115 (1994) 1–37. doi:10.1006/inco.1994.1092.

[6] P. Wolper, Temporal logic can be more expressive, Inf. and Cont. 56 (1983) 72–99. doi:10.1016/S0019-9958(83)80051-5.

[7] R. Alur, P. Madhusudan, Visibly pushdown languages, in: STOC 2004, ACM, 2004, pp. 202–211. doi:10.1145/1007352.1007390.

[8] R. Alur, K. Etessami, P. Madhusudan, A temporal logic of nested calls and returns, in: TACAS 2004, Vol. 2988 of LNCS, Springer, 2004, pp. 467–481. doi:10.1007/978-3-540-24730-2_35.

[9] L. Bozzelli, Alternating automata and a temporal fixpoint calculus for visibly pushdown languages, in: L. Caires, V. T. Vasconcelos (Eds.), CONCUR 2007, Vol. 4703 of LNCS, Springer, 2007, pp. 476–491. doi:10.1007/978-3-540-74407-8_32.

[10] C. Löding, P. Madhusudan, O. Serre, Visibly pushdown games, in: L. Lodaya, M. Mahajan (Eds.), FSTTCS 2004, Vol. 3328 of LNCS, Springer, 2005, pp. 408–420. doi:10.1007/978-3-540-30538-5_34.

[11] A. Bouajjani, J. Esparza, O. Maler, Reachability analysis of pushdown automata: Application to model-checking, in: A. Mazurkiewicz, J. Winkowski (Eds.), CONCUR 1997, Vol. 1243 of LNCS, Springer, 1997, pp. 135–150, Full version available at `http://www.liafa.univ-paris-diderot.fr/~abou/BEM97.pdf`. doi:10.1007/3-540-63141-0_10.

[12] L. Bozzelli, C. Sánchez, Visibly linear temporal logic, in: IJCAR 2014, Vol. 8562 of LNCS, 2014, pp. 418–483. doi:10.1007/978-3-319-08587-6_33.

[13] A. Weinert, VLDL Satisfiability and Model Checking via Tree Automata, in: S. Lokam, R. Ramanujam (Eds.), FSTTCS 2017, Vol. 93 of LIPIcs, Schloss Dagstuhl - LZI, 2017, pp. 47:1–47:13. doi:10.4230/LIPIcs.FSTTCS.2017.47.

[14] P. Faymonville, M. Zimmermann, Parametric linear dynamic logic, Inf. Comput. 253 (2017) 237–256. doi:10.1016/j.ic.2016.07.009.

[15] K. Thompson, Programming techniques: Regular expression search algorithm, C. ACM 11 (6) (1968) 419–422.

[16] H. Chen, D. Wagner, MOPS: an infrastructure for examining security properties of software, in: V. Atluri (Ed.), CCS 2002, ACM, 2002, pp. 235–244. doi:10.1145/586110.586142.

[17] L. Bozzelli, C. Sánchez, Visibly rational expressions, Act. Inf. 51 (1) (2014) 25–49.

[18] A. Chandra, L. Stockmeyer, Alternation, in: FOCS 1976, IEEE, 1976, pp. 98–108. doi:10.1145/322234.322243.

[19] J. Hopcroft, R. Motwani, J. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, 2001.